Multidimensional Point Data [1]

Hanan Samet
Computer Science Department
University of Maryland
College Park, Maryland 20742

September 10, 1998

Multidimensional Point Data

The representation of multidimensional point data is a central issue in database design, as well as applications in many other fields that include computer graphics, computer vision, computational geometry, image processing, geographic information systems (GIS), pattern recognition, VLSI design, and others. These points can represent locations and objects in space as well as more general records. As an example of a general record, consider an employee record which has attributes corresponding to the employee's name, address, sex, age, height, weight, and social security number. Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute or key[2]) albeit the different dimensions have different type units (i.e., name and address are strings of characters, sex is binary; while age, height, weight, and social security number are numbers).

Formally speaking, a database is a collection of records, termed a *file*. There is one record per data point, and each record contains several attributes or keys. In order to facilitate retrieval of a record based on some of its attribute values, we assume the existence of an ordering for the range of values of each of these attributes. In the case of locational or numeric attributes, such an ordering is quite obvious as the values of these attributes are numbers. In the case of alphanumeric attributes, the ordering is usually based on the alphabetic sequence of the characters making up the attribute value. Other data such as color could be ordered by the characters making up the name of the color or possibly the color's wavelength. It should be clear that finding an ordering for the range of values of an attribute is generally not an issue; the only issue is what ordering to use!

The representation that is ultimately chosen for this collection depends, in part, on answers to the following questions:

1. What operations are to be performed on the data?
2. Should we organize the data or the embedding space from which the data is drawn?
3. Is the database static or dynamic (i.e., can the number of data points grow and shrink at will)?
4. Can we assume that the volume of data is sufficiently small so that it can all fit in core, or should we make provisions for accessing disk-resident data?

Note that we have already made use of the distinction made in 2 for one-dimensional data in our discussion of searching in Chapter **??** where we differentiated between tree-based and trie-based methods. Recall that a trie [41] is a branching structure in which the value of a data item or key is treated as a sequence of j characters, where each character has M possible values and the first character is at position 0 while the last character is at position $j-1$. A node at depth k in the trie represents an M-way branch depending on the value of the k^{th} character. For example, in one dimension, the binary search tree [73] is an example of a tree-based method since the boundaries of different regions in the search space are determined by the data being stored. On the other hand, address computation methods such as radix searching [73] (also known as digital searching) are examples of trie-based methods, since region boundaries are chosen from among locations that are fixed regardless of the content of the file.

The extension of a trie to multidimensional data is relatively straightforward and proceeds as follows. Assume d-dimensional data with attribute i $(1 \leq i \leq d)$ being a sequence of j_i characters where each character has M_i possible values and the first character is at position 0 while the last character is at position $j_i - 1$. For the sake of this discussion, without loss of generality, let j_i be a constant c for all i $(1 \leq i \leq d)$. A node at depth k in the multidimensional trie represents an $\prod_{i=1}^{d} M_i$-way branch depending on the value of the k^{th} character of each of the d attribute values[3]. For attributes whose value is of type numeric which is treated as a

[2] We use the terms *key* and *attribute* (as well as *field*, *dimension*, *coordinate*, and *axis*) interchangeably in this chapter. The choice that we make is based on the context of the discussion while attempting to be consistent with their use in the literature.

[3] When j_i differs for the different attributes, there is no branching for attribute i once we have examined the ${j_i}^{th}$ character position in the value of attribute i.

sequence of binary digits, this process is usually a halving process in one dimension, quartering in two dimensions, etc. and is known as *regular decomposition*. We use this characterization frequently in our discussion of multidimensional data when all of the attributes are locational.

Disk-resident data implies grouping the data (either the underlying space based on the volume — that is, the amount — of the data it contains or the points, hopefully, by the proximity of their values) into sets (termed *buckets*) corresponding to physical storage units (i.e., pages). This leads to questions about their size, and how they are to be accessed, including:

1. Do we require a constant time to retrieve a record from a file or is a logarithmic function of the number of records in the file adequate? This is equivalent to asking if the access is via a directory in the form of an array (i.e., direct access) or in the form of a tree (i.e., a branching structure)?
2. How large can the directories be allowed to grow before it is better to rebuild them?
3. How should the buckets be laid out on the disk?

In this chapter we examine a number of representations that address these questions. Our primary focus is on dynamic data and we concentrate on the following queries:

1. Point queries — that is, if point p is present. If yes, then the result is the address corresponding to the record in which p is stored.
2. Range queries (e.g., region search) which yield a set of data points whose specified keys have specific values or values within given ranges. These queries include the partially specified query, also known as partial match and partial range, in which case unspecified keys take on the range of the key as their domain.
3. Boolean combinations of 1 and 2 using the Boolean operations `AND`, `OR`, `NOT`.

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes have the same unit, which is distance in space[4]. In this case, we can combine the attributes and pose queries that involve proximity. Assuming a Euclidean distance metric, for example, we may wish to find all cities within 50 miles of Chicago. This query is a special case of the range query which would seek all cities within 50 miles of the latitude position of Chicago and within 50 miles of the longitude position of Chicago[5]. A related query is one that seeks to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn. We do not deal with such queries in this chapter (but see [59]).

In contrast, proximity queries are not very meaningful when the attributes do not have the same type or units. For example, it is not customary to seek all people with age-weight combination less than 50 year-pounds (year-kilograms) of that of John Jones, or the person with age-weight combination closest to John Jones as we do not have a commonly accepted unit of year-pounds (year-kilograms) or definition thereof[6]. It should be clear that we are not speaking of queries involving Boolean combinations of the different attributes (e.g., range queries), which are quite common.

The representations that we describe are applicable to data with an arbitrary number of attributes d. The attributes need not be locational or numeric, although all of our examples and explanations assume that all of

[4] The requirement that the attribute value be a unit of distance in space is stronger than one that merely requires the unit to be a number. For example, if one attribute is the length of a pair of pants and the other is the width of the waist, then although the two attribute values are numbers, the two attributes are not locational.

[5] The difference between these two formulations of the query is that the former admits a circular search region while the latter admits a rectangular search region. In particular, the latter query is applicable to both locational and nonlocational data while the former is only applicable to locational data.

[6] This query is further complicated by the need to define the distance metric. We have assumed a Euclidean distance metric. However, other distance metrics such as the cityblock (also known as Manhattan) and the maximum value (also known as Chessboard) could also be used.

NAME	x	y
Chicago	35	42
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

Figure 1: Sample list of cities with their x and y coordinate values.

the attributes are locational or numeric. The concluding section contains a discussion of how these representations can be used to handle records that also contain nonlocational attributes. Moreover, in order to be able to visualize the representations we let $d = 2$. All of our examples make use of the simple database of records corresponding to eight cities given in Figure 1[7]. Each record has three attributes `NAME`, `x`, and `y`. We assume that our queries only retrieve on the basis of the values of the `x` and `y` attributes. Thus $d = 2$, and no ordering is assumed on the `NAME` field. In particular, the `NAME` field is just used to aid us in referring to the actual locations and is not really part of the record when we describe the various representations in this chapter.

1 Introduction

There are many possible representations for a file of multidimensional point data. The feasibility of these representations depends, in part, on the number of attributes and their domains. At one extreme is a pure bitmap representation of the attribute space with one bit reserved for each possible record (i.e., combination of attribute values) in the multidimensional point space whether or not it is present in the file. This representation is useful when the number of attributes d is small (i.e., $d \leq 2$) and the domains of the attributes are discrete and small (e.g., binary). Unfortunately, most applications have continuous domains and the number of attributes exceeds 2. Thus we must look for other solutions.

The simplest way to store point data is in a sequential list. In this case, no ordering is assumed for any of the attributes. Given N records and d attributes to search on, searching such a data structure is an $O(N \cdot d)$ process since each record must be examined. As an example, consider the set of eight cities and their x and y coordinate values as shown in Figure 1.

Another very common technique is the inverted list method [73], in which for each key a sorted list is maintained of the records in the file. Figure 2 is the inverted list representation of the data in Figure 1. There are two sorted lists; one for the x coordinate value and one for the y coordinate value. Note that the data stored in the lists are pointers into Figure 1. This enables pruning the search with respect to one key. In essence, the endpoints of the desired range for one key can be located very efficiently by using its corresponding sorted list. For example, this could be done by using a binary search with the aid of data structures such as the range tree [10, 13] as discussed in Section 2. This resulting list is then searched by brute force. The average search has been shown in [48] to be of $O(N^{1-1/d})$, under certain assumptions.

It should be clear that the inverted list is not particularly useful for range searches as it can only speed up the search for one of the attributes (termed the *primary* attribute). A number of solutions have been proposed. These solutions can be decomposed into two classes. One class of solutions enhances the range tree corresponding to the inverted list to include information about the remaining attributes in its internal nodes. For example, the multidimensional range tree stores ranges trees for the remaining dimensions in the internal

[7] Note that the correspondence between coordinate values and city names is not geographically accurate. We took this liberty so that the same example could be used throughout the chapter to illustrate a variety of concepts.

x	y
Denver	Miami
Omaha	Mobile
Chicago	Atlanta
Mobile	Omaha
Toronto	Chicago
Buffalo	Denver
Atlanta	Buffalo
Miami	Toronto

Figure 2: Inverted lists corresponding to the data of Figure 1.

nodes of the range tree. This is discussed in Section 2. The priority search tree [92] forms the basis of another solution known as the range priority tree [28] which is also related to the range tree. The priority search tree makes use of a binary search tree whose leaf nodes form a range tree for one dimension while its internal nodes form a heap for the remaining dimension. Such methods are discussed in Section 3.

A second class of solutions, and the one we focus on in the remaining sections of this chapter, modifies the bitmap representation to form d-dimensional cells of ranges of attribute values so that all points whose attribute values fall within the range associated with cell c are associated with c. This is analogous to a compressed bitmap representation.

The compressed bitmap representation forms the basis of the fixed-grid method [12, 73]. It partitions the space from which the data is drawn into rectangular cells by overlaying it with a grid. Each grid cell c contains a pointer to another structure (e.g., a list) which contains the set of points that lie in c. Associated with the grid is an access structure to enable the determination of the grid cell associated with a particular point p. This access structure acts like a directory and is usually in the form of a d-dimensional array with one entry per grid cell, or a tree with one leaf node per grid cell.

There are two ways to build a fixed grid. We can either subdivide the space into equal-sized grid cells or place the subdivision lines at arbitrary positions that are dependent on the underlying data. In essence, the distinction is between organizing the data to be stored and organizing the embedding space from which the data is drawn [99]. In particular, when the grid cells are equal-sized (termed a *uniform grid*), use of an array access structure is quite simple and has the desirable property that the grid cell associated with point p can be determined in constant time.

Figure 3 is an example of a uniform-grid representation corresponding to the data of Figure 1 with each grid cell having size 20×20. Assuming a 100×100 coordinate space, we have 25 squares of equal size. We adopt the convention that each square is open with respect to its upper and right boundaries and closed with respect to its lower and left boundaries (e.g., the point `(60,75)` is contained in the square centered at `(70,70)`). Also, all data points located at grid intersection points are said to be contained in the cell for which they serve as the `SW` corner (e.g., the point `(20,20)` is contained in the square centered at `(30,30)`).

When the width of each grid cell is twice the search radius for a rectangular range query[8], then the average search time is $O(F \cdot 2^d)$ where F is the number of points that have been found [16]. The factor 2^d is the maximum number of cells that must be accessed when the search rectangle is permitted to overlap more than one cell. For example, to locate all cities within a 20×20 square centered at `(32,37)` we must examine the cells centered at `(30,30)`, `(50,30)`, `(30,50)`, and `(50,50)`. Thus, four cells are examined and, for our example database, the query returns `Chicago` and `Omaha`.

Use of an array access structure when the grid cells are not equal-sized requires us to have a way of keeping track of their size so that we can determine the entry of the array access structure corresponding to the grid cell

[8] This method can also be described as yielding an *adaptive uniform grid* where the qualifier "adaptive" serves to emphasize that the cell size is a function of some property of the points (e.g., [39, 40]).

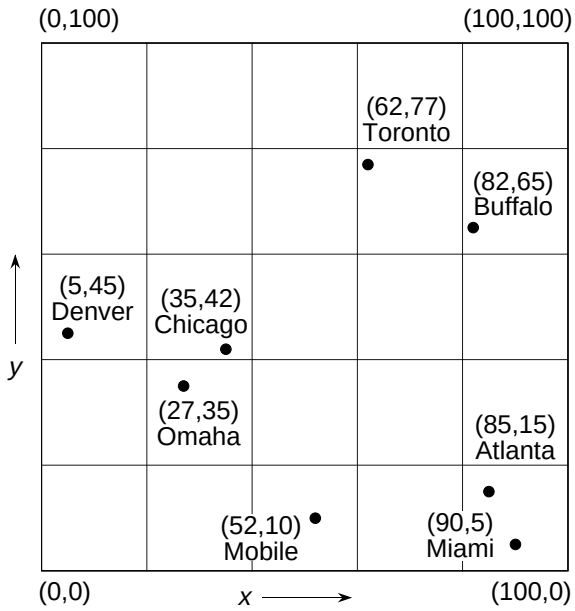


Figure 3: Uniform-grid representation corresponding to the data of Figure 1 with a search radius of 20.

associated with point p. One way to do this is to make use of what are termed *linear scales* which indicate the positions of the grid lines (or partitioning hyperplanes in $d > 2$ dimensions). Given a point p, we determine the grid cell in which p lies by finding the "coordinates" of the appropriate grid cell. The linear scales are usually implemented as one-dimensional trees containing ranges of values.

The use of an array access structure is fine as long as the data is static. When the data is dynamic, it is likely that some of the grid cells become too full while other grid cells are empty. This means that we need to rebuild the grid (i.e., further partition the grid or reposition the grid partition lines or planes) so that the various grid cells are not too full. However, this creates many more empty grid cells as a result of repartitioning the grid (i.e., empty grid cells are split into more empty grid cells). In this case, we have two alternatives.

The first alternative is to merge spatially-adjacent empty grid cells into larger empty grid cells, while splitting grid cells that are too full, thereby making the grid adaptive. The result is that we no longer make use of an array access structure to retrieve the grid cell that contains query point p. Instead, we make use of a tree access structure in the form of a k-ary tree where k is usually 2^d. Thus what we have done is marry a k-ary tree with the fixed-grid. This is the basis of the point quadtree [37] as well as trie-based representations such as the MX quadtree [122] and the PR quadtree [103, 124] which are all multidimensional generalizations of binary trees. They are discussed in Section 4.

As the dimensionality of the space increases, each level of decomposition of the quadtree results in many new cells since the fanout value of the tree is high (i.e., 2^d). This is alleviated by making use of a *k-d tree* [9]. This is a binary tree where at each level of the tree, we subdivide along just one of the attributes. This is discussed in Section 5.

The second alternative is to assign an ordering to all the grid cells and to impose a tree access structure on the elements of the ordering that correspond to nonempty grid cells. The effect of this alternative is analogous to using a mapping from d dimensions to one dimension and then applying one of the one-dimensional access structures such as a B-tree, a balanced binary tree, etc. discussed in Chapter **??** to the result of the mapping[9]. This is the subject of Section 6. Note that this alternative is applicable regardless of whether or not the grid

[9] These mappings have been investigated primarily for purely locational or numeric multidimensional point data in which case they can also be applied directly to the key values. They cannot be applied directly to the key values for nonlocational point data in which they can only be applied to the grid cell locations.

cells are equal-sized. Of course, if the grid cells are not equal-sized (not discussed here), then we must also record their size in the element of the access structure.

At times, in the dynamic situation, the data volume becomes so large that a conventional tree access structure (i.e., with the limited fanout values described above) is inefficient. In particular, the grid cells can become so numerous that they cannot all fit into memory thereby causing them to be grouped into sets (which we termed *buckets*) corresponding to physical storage units (i.e., pages) in secondary storage. This is the subject of Section 7. In the case of a tree access structure, each bucket usually corresponds to a contiguous set of grid cells that forms a hyper-rectangle in the underlying space. The problem is that, depending on the implementation of the tree access structure, each time we must follow a pointer, we may need to make a disk access.

This problem is resolved in two ways. The first way is to retain the use of the tree access structure, say T, and also aggregate the internal (i.e., nonleaf) nodes of T into buckets thereby forming a multiway tree in the spirit of a B-tree [22] (see Section **??** in Chapter **??**). The buckets whose contents are internal nodes of T correspond to subtrees of T and their fanout value corresponds to the sum of the number of possible branches at the deepest level of each of the subtrees. The resulting structure is now a tree of buckets with a large fanout value, and is frequently referred to as a *tree directory*. Examples of such structures are the R-tree and the R^+-tree, which are discussed in great detail in Sections **??** and sec-disjoint-object-based-hierarchical of Chapter **??**, respectively, as well as the k-d-B-tree and LSD tree which are discussed in Section 7.1.

The second way is to return to the use of an array access structure. The difference from the array used with the static fixed-grid method described earlier is that now the array access structure (termed a *grid directory*) may be so large (e.g., when d gets large) that it resides on disk as well (although this is not the case for all of the representations that we present), and the fact that the structure of the grid directory can be changed as the data volume grows or contracts. Each grid cell (i.e., an element of the grid directory) contains the address of a bucket (i.e., page) where the points associated with the grid cell are stored. Notice that a bucket can correspond to more than one grid cell. Thus any page can be accessed by two disk operations: one to access the grid cell and one more to access the actual bucket. The mechanics of the management of the growth of the access structure is the interesting issue here and is the subject of Section 7.2.

Section 8 contains a comparison of the various representations that we present. It also discusses briefly how to adapt the representations to handle nonlocational attributes. In general, the representations that we have presented are good for range searching (e.g., finding all cities within 80 miles of Atlanta) as they act as pruning devices on the amount of search that will be performed. In particular, many points will not be examined since their containing grid cells lie outside the query range. Most of these representations are generally very easy to implement and have good expected execution times, although the execution times of algorithms that use some of them are often quite difficult to analyze from a mathematical standpoint. However, their worst cases, despite being rare, can be quite bad. This is is especially true for the second class of solutions that were based on modifying the bitmap representation. These worst cases can be avoided by making use of members of the first class of solutions that were variants of range trees and priority search trees.

Exercises

1. Given a file containing N records with d keys apiece, implemented as an inverted list, prove that the average search for a particular record is $O(N^{1-1/d})$.

2. Given a file containing records with d keys apiece, implemented using the uniform grid method such that each cell has a width equal to twice the search radius of a rectangular range query, prove that the average time for the range query is $O(F \cdot 2^d)$ where F is the number of records found and 2^d is the number of cells that must be accessed.

3. Assume a file containing records with d keys apiece and implemented using the fixed-grid method such that each cell has width equal to the search radius of a rectangular range query. Prove that the average time for the range query is $O(F \cdot 3^d)$ where F is the number of records found and 3^d is the number of cells that must be accessed.

2 Range Trees

The multidimensional *range tree* of Bentley and Maurer [10, 13] is an asymptotically faster search structure compared to the point quadtree and the k-d tree; however, it has significantly higher storage requirements. It stores points and is designed to detect all points that lie in a given range.

The range tree is best understood by first examining the one-dimensional range searching problem. A one-dimensional range tree is a balanced binary search tree where the data points are stored in the leaf nodes and the leaf nodes are linked in sorted order by use of a doubly-linked list. The nonleaf nodes contain midrange values that enable discriminating between the left and right subtrees. For example, Figure 4 is the range tree for the data of Figure 1 when they are sorted according to values of their x coordinate. A range search for $[B : E]$ is performed by procedure `RANGE_SEARCH`. It searches the tree and finds the node with either the largest value $\leq B$ or the smallest value $\geq B$, and then follows the links until reaching a leaf node with a value greater than E. For N points, this process takes $O(\log_2 N + F)$ time and uses $O(N)$ storage. F is the number of points found.

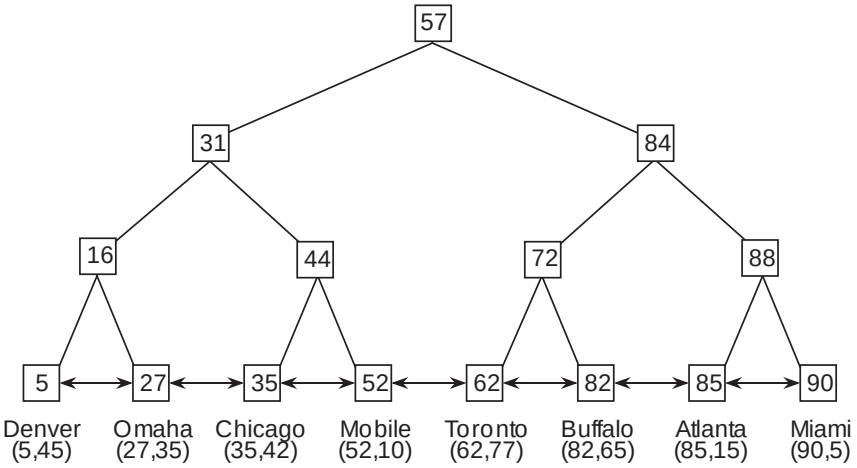


Figure 4: Range tree for the x coordinate values of the data of Figure 1.

Procedure `RANGE_SEARCH` assumes that each node has six fields, `LEFT`, `RIGHT`, `VALUE`, `PREV`, `NEXT`, and `MIDRANGE`. `LEFT`(P) and `RIGHT`(P) denote the left and right sons, respectively, of nonleaf node P (they are null in the case of a leaf node). `VALUE` is an integer indicating the value stored in the leaf node. `PREV`(P) and `NEXT`(P) are only meaningful for leaf nodes in which case they are used for the doubly-linked list of leaf nodes sorted in nondecreasing order. In particular, `PREV`(P) points to a node with value less than or equal to `VALUE`(P), while `NEXT`(P) points to a node with value greater than or equal to `VALUE`(P). `MIDRANGE`(P) is a variant of a discriminator between the left and right subtrees — that is, it is greater than or equal to the values stored in the left subtree, and less than or equal to the values stored in the right subtree (see Exercise 2). The `MIDRANGE` field is only meaningful for nonleaf nodes. Note that by making use of a `NODETYPE` field to distinguish between leaf and nonleaf nodes, we can use the `LEFT`, `RIGHT`, and `MIDRANGE` fields to indicate the information currently represented by the `PREV`, `NEXT`, and `VALUE` fields, respectively, thereby making them unnecessary.

```
procedure RANGE_SEARCH(B,E,T);
/* Perform a range search for the one-dimensional interval [B:E] in the one-dimensional range tree rooted at
   T. */
begin
   value integer B,E;
   value pointer node T;
   if NULL(T) then return;
   while NOT(LEAF(T)) do
```

```
    T←if B≤MIDRANGE(T) then LEFT(T)
        else RIGHT(T);
  if NOT(NULL(T)) and VALUE(T)<B then T←NEXT(T);
  while NOT(NULL(T)) and VALUE(T)≤E do
    begin
      output(VALUE(T));
      T←NEXT(T);
    end;
end;
```

For example, suppose we want to perform a range search for `[28:62]` on the range tree in Figure 4. In this example, we assume that the `VALUE` field of the leaf nodes only contains the x coordinate values. We first descend the tree to locate, in this case, the node with the largest value $\leq$`28` (i.e., `(25,35)`). Next, following the `NEXT` links, we report the points `(35,40)`, `(50,10)`, and `(60,75)`. We stop when encountering `(80,65)`.

A two-dimensional range tree is a binary tree of binary trees. It is formed in the following manner. First, sort all of the points along one of the attributes, say x, and store them in the leaf nodes of a balanced binary search tree (i.e., a range tree), say T. With each node of T, say I, associate a one-dimensional range tree, say T_I, of the points in the subtree rooted at I where now these points are sorted along the other attribute, say y[10]. For example, Figure 5 is the two-dimensional range tree for the data of Figure 1 where the x coordinate serves as the primary sort attribute. Notice the use of double lines to indicate a link to the one-dimensional range tree associated with each nonleaf node, thereby distinguishing them from the single lines used to indicate links to left and right sons in the two trees. For N points, the two-dimensional range tree uses $O(N \cdot \log_2 N)$ storage (see Exercise 7), and requires $O(N \cdot \log_2 N)$ time to build (see Exercise 8).

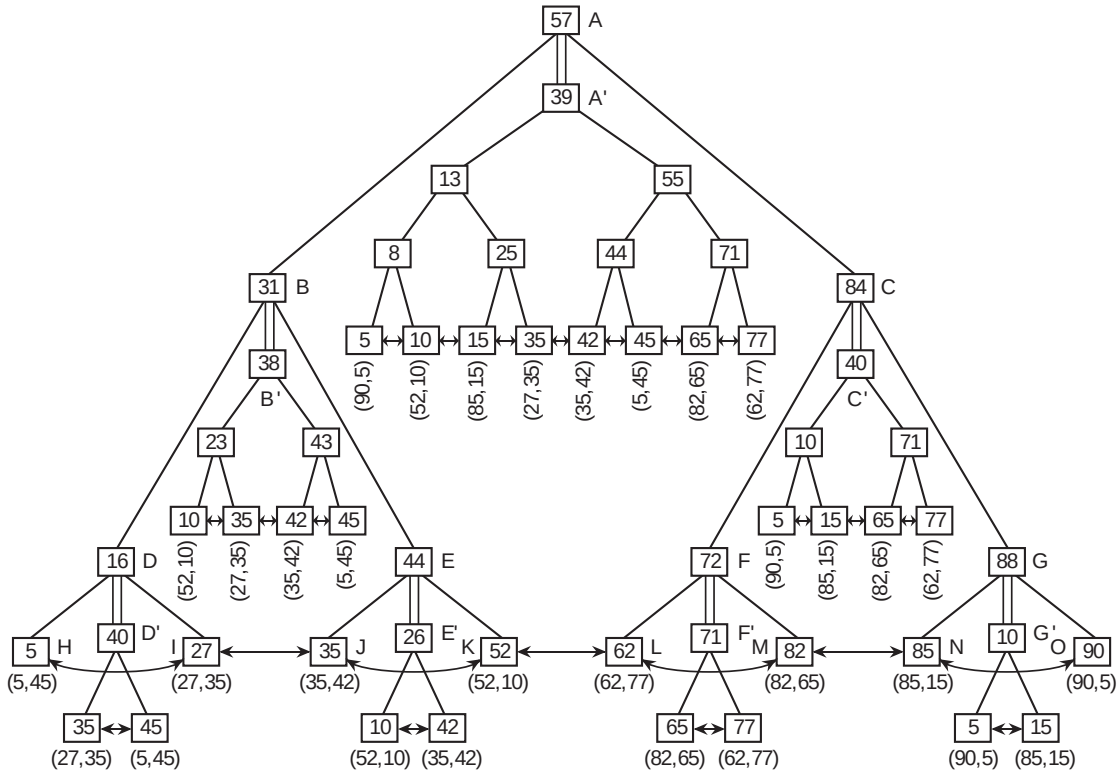


Figure 5: 2-d range tree for the data of Figure 1.

[10] Actually, there is no need for the one-dimensional range trees at the root and its two sons (see Exercise 6). Also, there is no need for the one-element one-dimensional range trees at the leaf nodes as the algorithms can make a special check for this case and use the data that is already stored in the leaf nodes (but see Exercise 5).

A range search for $([B_x : E_x], [B_y : E_y])$ is performed by procedure `2D_SEARCH`, given below. It makes use of procedure `1D_SEARCH`, a variant of procedure `RANGE_SEARCH` not given here.[11] Procedure `2D_SEARCH` starts by descending the tree in search of the closest nonleaf node Q (in terms of the number of links that are descended) to the root whose midrange value lies between B_x and E_x. Let L_x and R_x be the leaf nodes in the two-dimensional range tree that are reported by a search for B_x and E_x, respectively, regardless of whether or not B_x and E_x are actually stored there. L_x and R_x are termed *boundary nodes*. For example, using Figure 5 if B_x is `24` and E_x is `74`, then L_x and R_x are the leaf nodes containing `(25,35)` and `(80,65)`, respectively. Q is known as the *nearest common ancestor* of L_x and R_x in T (and hence the furthest from the root in terms of node links). Let $\{L_i\}$ and $\{R_i\}$ denote the sequences of nodes (excluding Q) that form the paths in T from Q to L_x and from Q to R_x, respectively.

Procedure `2D_SEARCH` assumes that each node has five fields, `LEFT`, `RIGHT`, `MIDRANGE`, `RANGE_TREE`, and `POINT`. `LEFT`, `RIGHT`, and `MIDRANGE` have the same meaning as in the one-dimensional range tree with the exception that `MIDRANGE` discriminates on the values of the x coordinate of the points represented by the tree. `RANGE_TREE(`P`)` is the one-dimensional range tree for y that is stored at node P. `POINT` is a pointer to a record of type *point* which stores the point that is associated with a leaf node. A *point* record has two fields, `XCOORD` and `YCOORD`, that contain the values of the x and y coordinates, respectively, of the point. The `POINT` field is not used for nonleaf nodes in the tree. Note that by making `2D_SEARCH` a bit more complicated, we can get by with just one of the `POINT` and `RANGE_TREE` fields (see Exercise 5).

For each node P that is an element of $\{L_i\}$ such that `LEFT(`P`)` is also in $\{L_i\}$, `2D_SEARCH` performs a one-dimensional range search for $[B_y : E_y]$ in the one-dimensional range tree for y associated with node `RIGHT(`P`)`. For each P that is an element of $\{R_i\}$ such that `RIGHT(`P`)` is also in $\{R_i\}$, `2D_SEARCH` performs a one-dimensional range search for $[B_y : E_y]$ in the one-dimensional range tree for y associated with node `LEFT(`P`)`. A final step checks (using procedure `IN_RANGE` not given here) if the points associated with the boundary nodes (i.e., leafs) L_x and R_x are in the two-dimensional range.

```
procedure 2D_SEARCH(BX,EX,BY,EY,T);
/* Perform a range search for the two-dimensional interval ([BX:EX],[BY:EY]) in the two-dimensional range
   tree rooted at T. */
begin
   value integer BX,EX,BY,EY;
   value pointer node T;
   pointer node Q;
   if NULL(T) then return;
   while NOT(LEAF(T)) do
      begin /* Find nearest common ancestor */
         if EX<MIDRANGE(T) then T←LEFT(T)
         else if MIDRANGE(T)<BX then T←RIGHT(T)
         else exit_while_loop; /* Found nearest common ancestor */
      end;
   if LEAF(T) then IN_RANGE(T,BX,EX,BY,EY)
   else
      begin /* Nonleaf node and must process subtrees */
         Q←T; /* Save value to process other subtree */
         T←LEFT(T);
         while NOT(LEAF(T)) do
            begin /* Process the left subtree */
               if BX≤MIDRANGE(T) then
                  begin
                     1D_SEARCH(BY,EY,RANGE_TREE(RIGHT(T)));
                     T←LEFT(T);
                  end
```

[11] The difference is that in `1D_SEARCH`, the leaf nodes are points in two-dimensional space, while in `RANGE_SEARCH`, the leaf nodes are points in one-dimensional space.

```
        else T←RIGHT(T);
      end;
    IN_RANGE(T,BX,EX,BY,EY);
    T←RIGHT(Q);
    while NOT(LEAF(T)) do
      begin /* Process the right subtree */
        if MIDRANGE(T)≤EX then
          begin
            1D_SEARCH(BY,EY,RANGE_TREE(LEFT(T)));
            T←RIGHT(T);
          end
        else T←LEFT(T);
      end;
    IN_RANGE(T,BX,EX,BY,EY);
  end;
end;
```

For example, the desired closest common ancestor of $\mathtt{L_x}$ and $\mathtt{R_x}$ in Figure 6 is Q. One-dimensional range searches would be performed in the one-dimensional range trees rooted at nodes A, B, D, E, F, and H since $\{L_i\} = \{\mathtt{L_1}, \mathtt{L_2}, \mathtt{L_3}, \mathtt{L_4}, \mathtt{L_x}\}$ and $\{R_i\} = \{\mathtt{R_1}, \mathtt{R_2}, \mathtt{R_3}, \mathtt{R_4}, \mathtt{R_x}\}$. For N points, procedure 2D_SEARCH takes $O(\log_2^2 N + F)$ time where F is the number of points found (see Exercise 9).

As a more concrete example, suppose we want to perform a range search for ([25:85],[8:16]) on the two-dimensional range tree in Figure 5. We first find the nearest common ancestor of 25 and 85 which is node A. The paths $\{L_i\}$ and $\{R_i\}$ are given by {B,D,I} and {C,G,N}, respectively. Since B and B's left son (i.e., D) are in the path to 25, we search the range tree of B's right son (i.e., E) and report (50,10) as in the range. Similarly, since C and C's right son (i.e., G) are in the path to 85, we search the range tree of C's left son (i.e., F) but do not report any results as neither (60,75) nor (80,65) are in the range. Finally, we check if the boundary nodes (25,35) and (85,15) are in the range, and report (85,15) as in the range.

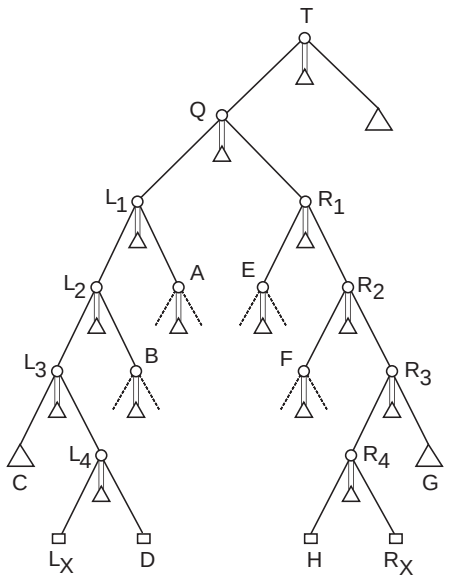


Figure 6: Example 2-d range tree to illustrate 2-d range searching.

The range tree also can be adapted easily to handle k-dimensional data. In such a case, for N points, a k-dimensional range search takes $O(\log_2^k N + F)$ time where F is the number of points found. The k-dimensional range tree uses $O(N \cdot \log_2^{k-1} N)$ storage (see Exercise 10), and requires $O(N \cdot \log_2^{k-1} N)$ time to build (see Exercise 11).

Exercises

1. Is there a difference between a balanced binary search tree where all the data is stored in the leaf nodes and a one-dimensional range tree?

2. The `MIDRANGE` field in the range tree was defined as a variant of a discriminator between the left and right subtrees in the sense that it is greater than or equal to the values stored in the left subtree, and less than or equal to the values stored in the right subtree. Why not use a simpler definition such as one that that stipulates that the `MIDRANGE` value is greater than all values in the left subtree and less than or equal to all values in the right subtree?

3. Why does procedure `2D_SEARCH` provide the desired result?

4. Prove that no point is reported more than once by the algorithm for executing a range query in a two-dimensional range tree.

5. Procedure `2D_SEARCH` makes use of a representation of the two-dimensional range tree where each node has a `POINT` and `RANGE_TREE` field. In fact, the `POINT` field is only defined for leaf nodes, while for leaf nodes, the `RANGE_TREE` field points to a one-dimensional range tree of just one node. Rewrite `2D_SEARCH`, `1D_SEARCH`, and `IN_RANGE` so that they do not use a `POINT` field, and hence interpret the `RANGE_TREE` field appropriately.

6. Show that the one-dimensional range trees at the first two levels (i.e., at the root and the two sons of the root) of a two-dimensional range tree are never used in procedure `2D_SEARCH`.

7. Show that $O(N \cdot \log_2 N)$ storage suffices for a two-dimensional range tree for N points.

8. Show that a two-dimensional range tree can be built in $O(N \cdot \log_2 N)$ time for N points.

9. Given a two-dimensional range tree containing N points, prove that a two-dimensional range query takes $O(\log_2^2 N + F)$ time, where F is the number of points found.

10. Given a k-dimensional range tree containing N points, prove that a k-dimensional range query takes $O(\log_2^k N + F)$ time, where F is the number of points found. Also, show that $O(N \cdot \log_2^{k-1} N)$ storage is sufficient.

11. Show that a k-dimensional range tree can be built in $O(N \cdot \log_2^{k-1} N)$ time for N points.

12. Write a procedure to construct a two-dimensional range tree.

3 Priority Search Trees

The *priority search tree* is a data structure that is designed for solving queries involving semi-infinite ranges in two-dimensional space. A typical query has a range of the form $([B_x : E_x], [B_y : \infty])$. It is built in the following manner. Sort all the points along the x coordinate, and store them in the leaf nodes of a balanced binary search tree, say T. We use a range tree in our formulation[12] although in some applications (e.g., the rectangle intersection problem as discussed in Section **??** of Chapter **??**) the requirement of being able to insert and delete points in $O(\log_2 N)$ time, while still requiring the $O(N \cdot \log_2 N + F)$ search behavior, causes the balanced binary search tree to be implemented using more complicated structures such as a 'red-black' balanced binary tree [54] (see Section **??** of Chapter **??**). Store midrange x coordinate values in the nonleaf nodes. Next, proceed from the root node towards the leaf nodes. Associate with each node I of T the point in the subtree rooted at I with the maximum value for its y coordinate that has not already been associated with a node at a shallower depth in the tree. If no such point exists, then leave the node empty. For N points, this structure uses $O(N)$ storage (see Exercise 1), and requires $O(N \cdot \log_2 N)$ time to build (see Exercise 6).

[12] A Cartesian tree [142] is an alternative. See Exercises 17 and 18.

For example, Figure 7 is the priority search tree for the data of Figure 1. In the figure, square boxes contain the relevant x coordinate values (e.g., the midrange values in the nonleaf nodes), while circular boxes contain the relevant y coordinate values (e.g., the maximums). Notice that the leaf nodes contain the points and are shown as linked together in ascending order of the x coordinate value. The links for the x coordinate values are only used if we conduct a search for all points within a given range of x coordinate values. They are not used in the execution of the semi-infinite range query and thus they are frequently omitted from the implementation of the priority search tree.

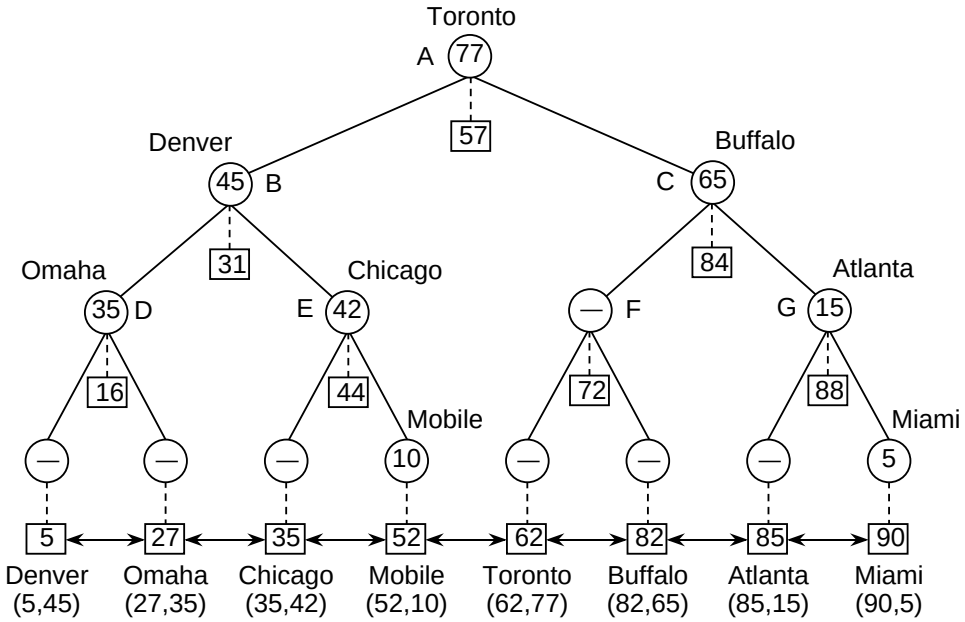


Figure 7: Priority search tree for the data of Figure 1. Each leaf node contains the value of its x coordinate in a square box. Each nonleaf node contains the appropriate x coordinate midrange value in a square box using a link drawn with a broken line. Circular boxes indicate the value of the y coordinate of the point in the corresponding subtree with the maximum value for its y coordinate that has not already been associated with a node at a shallower depth in the tree.

It is not easy to perform a two-dimensional range query of the form $([B_x : E_x], [B_y : E_y])$ with a priority search tree. The problem is that only the values of the x coordinate are sorted. In other words, given a leaf node C that stores point (x_C, y_C), we know that the values of the x coordinate of all nodes to the left of C are smaller than or equal to x_C and the values of all those to the right of C are greater than or equal to x_C. On the other hand, with respect to the values of the y coordinate, we only know that all nodes below nonleaf node D with value y_D have values less than or equal to y_D; the y coordinate values associated with the remaining nodes in the tree that are not ancestors of D may be larger or smaller than y_D.

This is not surprising because a priority search tree is really a variant of a range tree in x (see Exercise 1 in Section 2) and a heap (i.e., priority queue) [71] in y. A heap (see Section **??** in Chapter **??**) enables finding the maximum (minimum) value in $O(1)$ time. More generally the largest (smallest) F out of N values can be determined in $O(F)$ time, but the F values are not sorted. Whether the heap is used to find the largest or smallest values depends on how it is constructed. The way in which we specify the priority search tree here enables finding the largest values.

Nevertheless, despite the difficulty in using priority search trees to perform a two-dimensional range query, priority search trees make it very easy to perform a semi-infinite range query of the form $([B_x : E_x], [B_y : \infty])$. The control structure is quite similar to that of procedure `2D_SEARCH`. Descend the tree looking for the nearest common ancestor of B_x and E_x, say Q. Apply the following tests during this descent, letting P be the point associated with the examined node, say T. If no such P exists, then we are finished with the entire

subtree rooted at T as all points in T's subtrees have already been examined and/or reported. Examine the y coordinate value of P, say P_y. If $P_y < B_y$, then we are finished with the entire subtree rooted at T since P is the point with the maximum y coordinate value in T. Otherwise (i.e., $P_y \geq B_y$), check if the x coordinate value of P (i.e., P_x) is in the range $[B_x : E_x]$, and if yes, then output P as satisfying the query. At this point, determine if T is the nearest common ancestor by checking if B_x and E_x lie in T's left and right subtrees, respectively. If T is not the nearest common ancestor, then continue the descent in the left (right) subtree of T if E_x (B_x) is in the left (right) subtree of T.

Once Q has been found, process the left and right subtrees of Q using the appropriate step below where T denotes the node currently being examined. In both cases, we apply the same sequence of tests described above to determine if an exit can be made due to having processed all points in the subtrees or if all remaining points are not in the y range.

1. Left subtree of Q: check if B_x lies in the left subtree of T. If yes, then all of the points in the right subtree of T are in the x range and we just check if they are in the y range, while recursively applying this step to the left subtree of T. Otherwise, recursively apply this step to the right subtree of T.

2. Right subtree of Q: check if E_x lies in the right subtree of T. If yes, then all of the points in the left subtree of T are in the x range and we just check if they are in the y range, while recursively applying this step to the right subtree of T. Otherwise, recursively apply this step to the left subtree of T.

The actual search process is given by procedure `PRIORITY_SEARCH`. It assumes that each node has five fields, `POINT`, `MIDRANGE`, `LEFT`, `RIGHT`, and `HEAP_MAX`. `POINT`, `MIDRANGE`, `LEFT`, and `RIGHT` have the same meaning as in the two-dimensional range tree. `HEAP_MAX` is defined for all nodes. In particular, for any node *I*, `HEAP_MAX(`I`)` points at the node *C* in the subtree *S* rooted at *I* such that `YCOORD(`C`)` is greater than or equal to `YCOORD(`D`)` for any other node *D* in *S*, such that neither *C* nor *D* are associated with a node at a depth in the tree that is shallower than the depth of *I*. `PRIORITY_SEARCH` makes use of two auxiliary procedures `OUT_OF_RANGE_Y` and `Y_SEARCH`. `OUT_OF_RANGE_Y` determines if all points in a subtree have already been output or if they are all outside the y range. `Y_SEARCH` performs a one-dimensional range search for the semi-infinite interval with respect to the y range.

```
procedure PRIORITY_SEARCH(BX,EX,BY,T);
/* Perform a semi-infinite range search for the two-dimensional interval ([BX:EX],[BY:∞]) in the priority
   search tree rooted at T. */
begin
  value integer BX,EX,BY;
  value pointer node T;
  if null(T) then return;
  while true do
    begin /* Find nearest common ancestor */
      if OUT_OF_RANGE_Y(HEAP_MAX(T),BY) then return
      else if BX≤XCOORD(HEAP_MAX(T)) and XCOORD(HEAP_MAX(T))≤EX then
        output(HEAP_MAX(T));
      if LEAF(T) then return
      else if EX<MIDRANGE(T) then T←LEFT(T)
      else if MIDRANGE(T)<BX then T←RIGHT(T)
      else exit_while_loop; /* Found nearest common ancestor */
    end;
  /* Nonleaf node and must process subtrees */
  Q←T; /* Save value to process other subtree */
  T←LEFT(T);
  while true do
    begin /* Process the left subtree */
      if OUT_OF_RANGE_Y(HEAP_MAX(T),BY) then exit_while_loop
```

```
      else if BX≤XCOORD(HEAP_MAX(T)) then output(HEAP_MAX(T));
      if LEAF(T) then exit_while_loop
      else if BX≤MIDRANGE(T) then
        begin
          Y_SEARCH(RIGHT(T),BY);
          T←LEFT(T);
        end
      else T←RIGHT(T);
    end;
  T←RIGHT(Q);
  while true do
    begin /* Process the right subtree */
      if OUT_OF_RANGE_Y(HEAP_MAX(T),BY) then return
      else if XCOORD(HEAP_MAX(T))≤EX then output(HEAP_MAX(T));
      if LEAF(T) then return
      else if MIDRANGE(T)≤EX then
        begin
          Y_SEARCH(LEFT(T),BY);
          T←RIGHT(T);
        end
      else T←LEFT(T);
    end;
end;

Boolean procedure OUT_OF_RANGE_Y(P,BY);
/* Check if all relevant points have already been output or if the y coordinate values of all points in the subtrees
   are too small. */
begin
  return(if null(P) then 'true'
         else YCOORD(P)<BY);
end;

recursive procedure Y_SEARCH(T,BY);
/* Perform a one-dimensional range search for the semi-infinite interval [BY:∞] in the priority search tree
   rooted at T. */
begin
  value pointer node T;
  value integer BY;
  if null(T) then return
  else if OUT_OF_RANGE(HEAP_MAX(T),BY) then return
  else
    begin
      output(HEAP_MAX(T));
      Y_SEARCH(LEFT(T),BY);
      Y_SEARCH(RIGHT(T),BY);
    end;
end;
```

For N points, performing a semi-infinite range query in this way takes $O(\log_2 N + F)$ time. F is the number of points found (see Exercise 2). To answer the two-dimensional range query $([B_x : E_x], [B_y : E_y])$, perform the semi-infinite range query $([B_x : E_x], [B_y : \infty])$ and discard all points (x, y) such that $y > E_y$.

As an example of the execution of a semi-infinite range query, suppose that we want to search for ([35:80],[50:∞]) in the priority search tree in Figure 7. We use Figure 8a to illustrate the space decomposition induced by the

priority search tree of Figure 7. In the figure, horizontal lines correspond to the partitions caused by the HEAP_MAX field values in the tree. The vertical lines correspond to the partitions caused by the MIDRANGE field values associated with the nonleaf nodes whose HEAP_MAX field is not NIL.

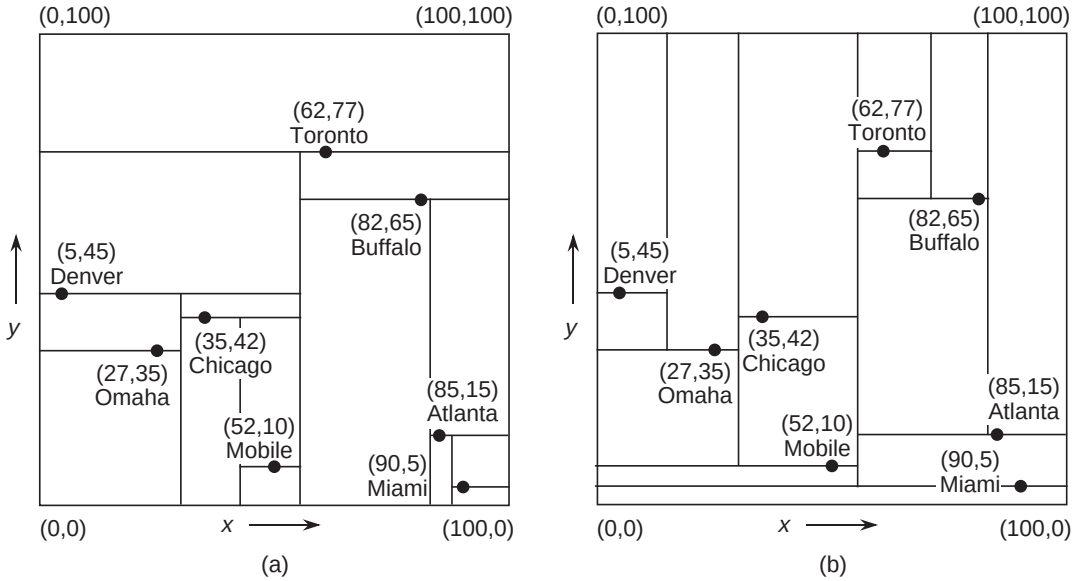


Figure 8: Space decomposition induced by (a) the priority search tree of Figure 7 and (b) the inverse priority search tree of Figure 9. Horizontal lines correspond to the partitions caused by the HEAP_MAX field values in the tree. Vertical lines correspond to the partitions caused by the MIDRANGE field values associated with the nonleaf nodes whose HEAP_MAX field is not NIL.

The search proceeds as follows. We descend the tree starting at the root (i.e., A). The point associated with A is Toronto with a y coordinate value of 75 which is in the y range. The x coordinate value of Toronto is 60 which is in the x range and thus we output Toronto as in the two-dimensional range. Next, we find that A is indeed the nearest common ancestor since the midrange value is 55 which is between the x range values of 35 and 80. Thus we prepare to descend both of A's left and right sons.

We immediately cease processing the left subtree of A (i.e., B) since the maximum of the y coordinate values in B's subtrees is 45 which is less than the lower bound of the semi-infinite range of y (i.e., 50). Next, we process the root of the right subtree of A (i.e., C). The point associated with C is Buffalo with a y coordinate value of 65 which is in the y range. The x coordinate value of Buffalo is 80 which is in the x range and thus we output Buffalo as in the two-dimensional range. Next, we examine the midrange value of C which is 83. This means that we only need to descend the left subtree of C (i.e., rooted at F) since the right subtree of C (i.e., rooted at G) is out of the x range. However, since there is no point associated with F, processing ceases as this means that all nodes in F's subtrees have already been examined. Thus the result is that points (60,75) and (80,65) are in the range ([35:80],[50:∞]).

Edelsbrunner [28] introduces a variation on the priority search tree, which we term a *range priority tree*, to obtain an $O(\log_2 N + F)$ algorithm for range searching in a two-dimensional space. Define an *inverse priority search tree* to be a priority search tree S such that with each node of S, say I, we associate the point in the subtree rooted at I with the minimum (instead of the maximum!) value for its y coordinate that has not already been stored at a shallower depth in the tree. For example, Figure 9 is the inverse priority search tree corresponding to Figure 7, and Figure 8b is the space decomposition induced by it. The range priority tree is a balanced binary search tree (i.e., a range tree), say T, where all the data points are stored in the leaf nodes and are sorted by their y coordinate values. With each nonleaf node of T, say I, which is a left son of its father, we store a priority search tree of the points in the subtree rooted at I. With each nonleaf node of T, say I, which is a right son of its father we store an inverse priority search tree of the points in the subtree rooted at I. For

N points, the range priority tree uses $O(N \cdot \log_2 N)$ storage (see Exercise 11), and requires $O(N \cdot \log_2 N)$ time to build (see Exercise 12).

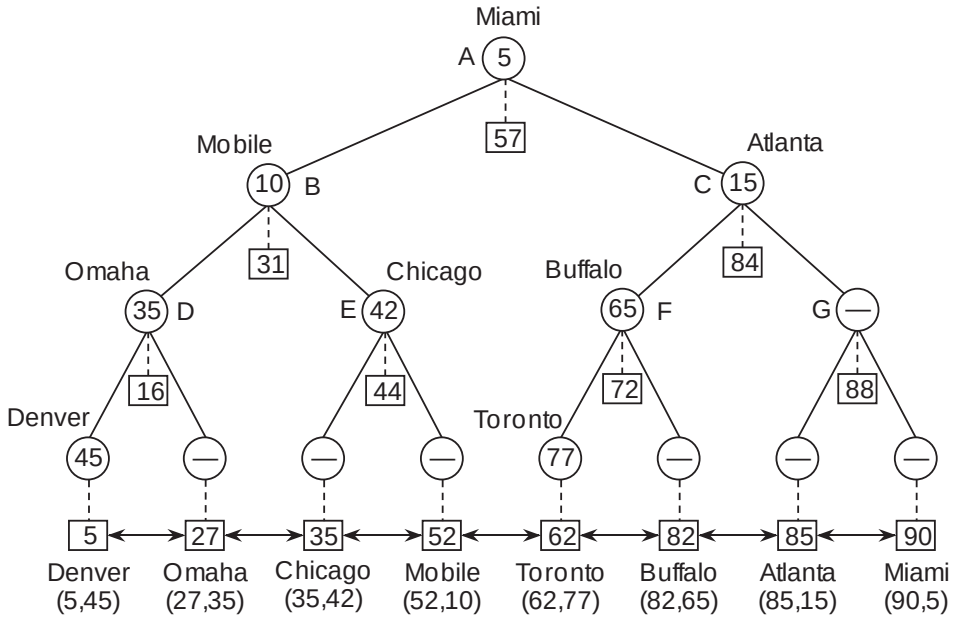


Figure 9: Inverse priority search tree for the data of Figure 1. Each leaf node contains the value of its x coordinate in a square box. Each nonleaf node contains the appropriate x coordinate midrange value in a box using a link drawn with a broken line. Circular boxes indicate the value of the y coordinate of the point in corresponding subtree with the minimum value for its y coordinate that has not already been associated with a node at a shallower depth in the tree.

For example, Figure 10 is the range priority tree for the data of Figure 1 where the y coordinate serves as the primary sort attribute. Notice the use of double lines to indicate a link to the priority and inverse priority search trees associated with each nonleaf node, thereby distinguishing them from the single lines used to indicate links to left and right sons in the two trees. In the figure, square boxes contain the relevant range tree values (e.g., the midrange values in the nonleaf nodes), while circular boxes contain the relevant heap values (i.e., maximums for the priority search tree and minimums for the inverse priority search tree). Notice that, as in the priority search tree, the leaf nodes contain the points and are shown as linked together in ascending order of the x coordinate value. Again, as in the priority search tree, the links for the x coordinate values are only used if we conduct a search for all points within a given range of x coordinate values. They are not used in the execution of the two-dimensional range query and thus they are frequently omitted from the implementation of the range priority tree.

Performing a range query for $([B_x : E_x], [B_y : E_y])$ using a range priority tree is done in the following manner. We descend the tree looking for the nearest common ancestor of B_y and E_y, say Q. The values of the y coordinate of all points in the left son of Q are less than or equal to E_y. We want to retrieve just the ones that are greater than or equal to B_y. We can obtain them with the semi-infinite range query $([B_x : E_x], [B_y : \infty])$. This can be done by using the priority tree associated with the left son of Q. The priority tree is good for retrieving all points with a specific lower bound as it stores an upper bound and hence irrelevant values can be easily pruned.

Similarly, the values of the y coordinate of all points in the right son of Q are greater than or equal to B_y. We want to retrieve just the ones that are less than or equal to E_y. We can obtain them with the semi-infinite range query $([B_x : E_x], [-\infty : E_y])$. This can be done by using the inverse priority search tree associated with the right son of Q. The inverse priority tree is good for retrieving all points with a specific upper bound as it stores a lower bound and hence irrelevant values can be easily pruned. Thus, for N points the range query

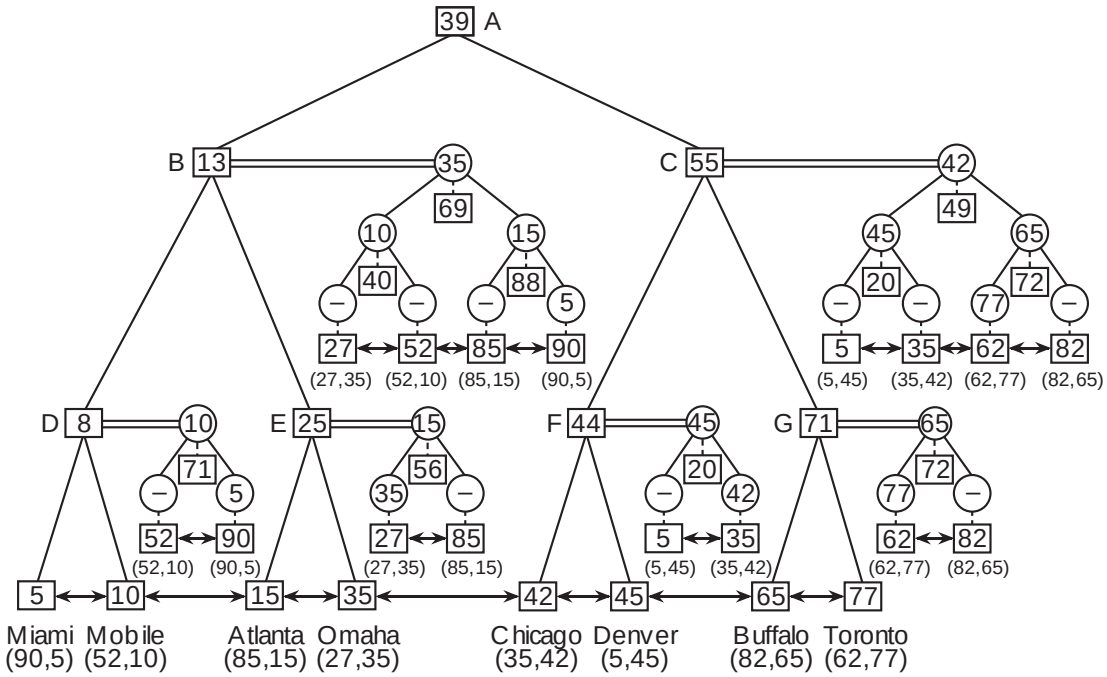


Figure 10: Range priority tree for the data of Figure 1. Square boxes contain the relevant range tree values (midrange values in the nonleaf nodes), and circular boxes contain the relevant heap values. The square boxes containing the midrange values in the priority and inverse priority search trees are linked to the appropriate nonleaf node by a broken line. Double lines indicate a link to the priority and inverse priority search trees associated with each nonleaf node, thereby distinguishing them from the single lines used to indicate links to left and right sons in the two trees.

takes $O(\log_2 N + F)$ time where F is the number of points found (see Exercise 13).

As an example, suppose we want to perform a range search for `([25:60],[15:45])` on the range priority tree in Figure 10. We first find the nearest common ancestor of `15` and `45` which is node `A`. Second, we search for `([25:60],[15:`∞`])` in the priority tree attached to left son of `A` (i.e., `B`), say p. The point `(25,35)` is associated with the root of p and it is in the range. Hence it is output. The left subtree of p is rejected as the y coordinate value associated with its root (i.e., `10`) is less than the lower limit of the search range in y (i.e., `15`). The elements of the right subtree of p are rejected as their x coordinate values are outside of the search range. Third, we search for `([25:60],[-`∞`:45])` in the inverse priority tree attached to the right son of `A` (i.e., `C`), say q. The point `(35,40)` is associated with the root of q and it is in the range. Hence it is output. The elements of the left subtree of q that have not been reported already (i.e., `(5,45)`) are rejected as their x coordinate values are outside of the search range. The right subtree of q is rejected as the y coordinate value associated with its root (i.e., `65`) is greater than the upper limit of the search range in y (i.e., `45`).

Exercises

1. Prove that a priority search tree for N points uses $O(N)$ space.

2. Prove that for a priority search tree with N points, a semi-infinite range query can be performed in $O(\log_2 N+F)$ time, where F is the number of points found.

3. Procedure `PRIORITY_SEARCH` can be made considerably shorter by changing its structure slightly as given below by procedure `PRIORITY_SEARCH_2`. The difference is that procedure `PRIORITY_SEARCH_2` does not make explicit use of the result of finding the nearest common ancestor. This forces the re-

peated comparison of B_x and E_x with the MIDRANGE field value of the node even after the nearest common ancestor has been determined. However, it does enable procedure PRIORITY_SEARCH_2 to make use of recursion as there is no longer a need to distinguish between processing the right and left sons of the nearest common ancestor, as well as testing if the node and its left son are on the path from the nearest common ancestor to B_x, or testing if the node and its right son are on the path from the nearest common ancestor to E_x. Prove that for N points, procedure PRIORITY_SEARCH_2 executes the semi-infinite range query in $O(\log_2 N + F)$ time where F is the number of points found. Which of PRIORITY_SEARCH and PRIORITY_SEARCH_2 performs fewer comparison operations?

```
recursive procedure PRIORITY_SEARCH_2(BX,EX,BY,T);
/* Perform a semi-infinite range search for the two-dimensional interval ([BX:EX],[BY:∞]) in the
   priority search tree rooted at T. */
begin
  value integer BX,EX,BY;
  value pointer node T;
  if null(T) then return
  else if null(HEAP_MAX(T)) then return
    /* All relevant points have already been output */
  else if YCOORD(HEAP_MAX(T))<BY then return
    /* y coordinate values of all points in the subtrees are too small */
  else if BX≤XCOORD(HEAP_MAX(T)) and XCOORD(HEAP_MAX(T))≤EX then
    output(HEAP_MAX(T));
  if BX≤MIDRANGE(T) then
    if MIDRANGE(T)≤EX then
      begin
        PRIORITY_SEARCH_2(BX,EX,BY,LEFT(T));
        PRIORITY_SEARCH_2(BX,EX,BY,RIGHT(T));
      end
    else PRIORITY_SEARCH_2(BX,EX,BY,LEFT(T))
  else PRIORITY_SEARCH_2(BX,EX,BY,RIGHT(T));
end;
```

4. In the original presentation of the priority search tree, two variants are defined [92]. The first variant, which is the one described in the text, makes use of a range tree on the x coordinate value and stores midrange x coordinate values in the nonleaf nodes. The second variant does not store the midrange values. Instead, it is based on a trie search where the domain of the x coordinate value is repeatedly halved. Thus the x coordinate value of all points in the left subtree are less than the partition value, while those of all points in the right subtree are greater than or equal to the partition value. This means that the midrange values are implicit to the tree. Prove that in this case, no two data points in the priority search tree can have the same x coordinate value.

5. Suppose that you are using the second variant of the priority search tree as described in Exercise 4. Thus by the time we reach the final partition there is just one value left. How can you get around the restriction that no two data points in a priority search tree have the same x coordinate value?

6. Prove that the construction of a priority search tree takes $O(N \cdot \log_2 N)$ time for N points.

7. Write a procedure to construct a priority search tree.

8. Write a procedure to construct an inverse priority search tree.

9. Modify procedure PRIORITY_SEARCH to deal with an inverse priority search tree.

10. Can you extend the priority search tree to handle k-dimensional data? If yes, show how you would do it.

11. Prove that a range priority tree for N points uses $O(N \cdot \log_2 N)$ space.

12. Prove that the construction of a range priority tree takes $O(N \cdot \log_2 N)$ time for N points.

13. Prove that for a range priority tree with N points, a two-dimensional range query can be performed in $O(\log_2 N + F)$ time, where F is the number of points found.

14. Can you extend the range priority tree to handle k-dimensional range queries? What is the order of the execution time of the query? How much space does it use?

15. Write a procedure to construct a range priority tree.

16. Write a procedure to search a range priority tree for a rectangular search region.

17. In the text, we characterized a priority search tree as variant of a range tree in x and a heap (i.e., priority queue) in y. The priority search tree is somewhat wasteful of space as the nonleaf nodes duplicate some of the information stored in the leaf nodes. Vuillemin [142] defines a *Cartesian tree* which is a binary search tree in x and a heap in y. In this case, each point (a, b) is stored at just one node — that is, both nonleaf and leaf nodes contain data. Construct a Cartesian tree for the data of Figure 1.

18. Is the Cartesian tree defined in Exercise 17 unique? In other words, can you apply rotation operations to it to make it balanced?

4 Quadtrees

Recall from Section 1 that the quadtree is the result of imposing a tree access structure on a grid so that spatially-adjacent empty grid cells are merged into larger empty grid cells. In essence, it is equivalent to marrying a k-ary tree, where $k = 2^d$, with the fixed grid. There are two types of quadtrees. The first is a point quadtree [37] where the subdivision lines are based on the values of the data points while the second is trie-based and forms a decomposition of the embedding space from which the data points are drawn.

In the rest of this section, we elaborate further on the differences between the point quadtree and the trie-based quadtree by showing how they are updated (i.e., node insertion and deletion) and how they are used for region searching. In particular, Section 4.1 presents the point quadtree. Section 4.2 discusses trie-based quadtrees such as the PR and MX quadtrees. Section 4.3 contains a brief comparison of the point, PR, and MX quadtrees.

4.1 Point Quadtrees

In two dimensions, the point quadtree, invented by Finkel and Bentley [37], is just a two-dimensional binary search tree. The first point that is inserted serves as the root, while the second point is inserted into the relevant quadrant of the tree rooted at the first point. Clearly, the shape of the tree depends on the order in which the points were inserted. For example, Figure 11 is the point quadtree corresponding to the data of Figure 3.

The rest of this section is organized as follows. Section 4.1.1 shows how to insert a point into a point quadtree. Section 4.1.2 discusses how to delete a point from a point quadtree. Section 4.1.3 explains how to do region searching in a point quadtree.

4.1.1 Insertion

Inserting record r with key values (a, b) into a point quadtree is very simple. The process is analogous to that for a binary search tree. First, if the tree is empty, then allocate a new node containing r and return a tree with it as its only node. Otherwise, search the tree for a node h with a record having key values (a, b). If h exists, then r replaces the record associated with h. Else, we are at a `NIL` node which is a son of type s (i.e., `NW`, `NE`, `SW`, or `SE`, as appropriate) of node c. This is where we make the insertion by allocating a new node

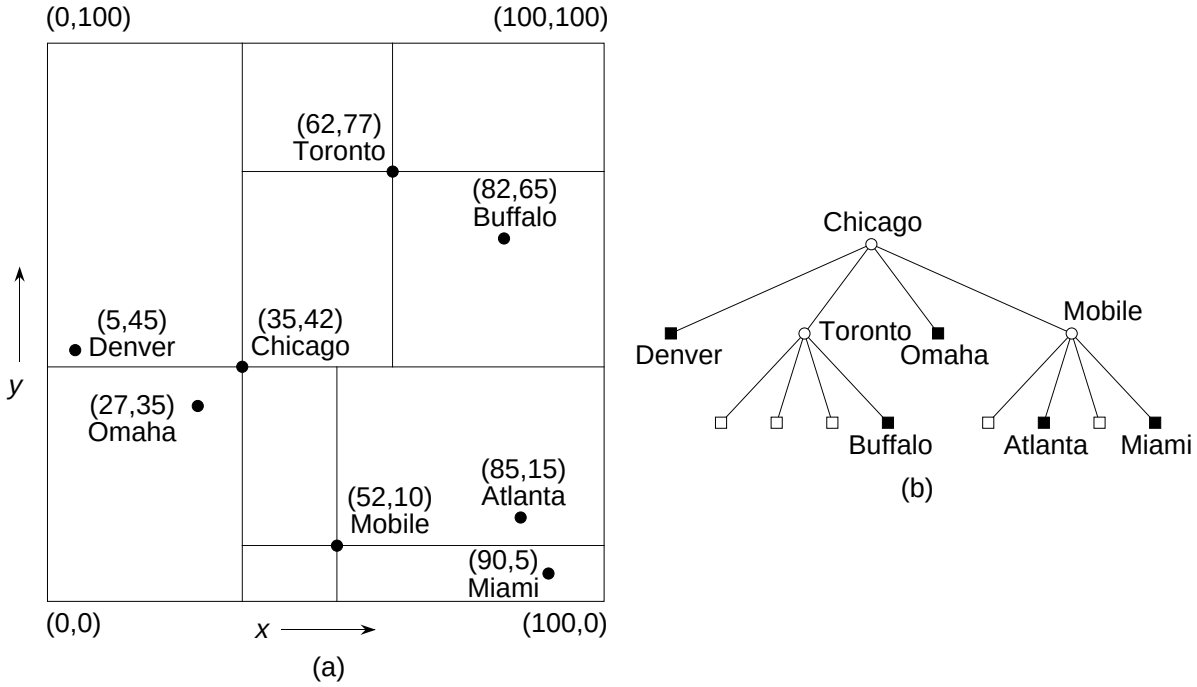


Figure 11: A point quadtree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

t containing r and make t an s son of node c. The only difference from the binary search tree is the need to perform four-way comparisons at each node of the tree. In particular, at each node q with key values (e, f) we must determine the quadrant of the subtree rooted at q in which r lies. This is done by determining the position of point (a, b) with respect to (e, f). For example, the tree in Figure 11 was built for the sequence `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`. Figure 12 shows how the tree was constructed in an incremental fashion for these cities by giving the appropriate block decompositions. In particular, Figure 12a corresponds to the tree after insertion of `Chicago` and `Mobile`, Figure 12b to the tree after insertion of `Toronto` and `Buffalo`, Figure 12c to the tree after insertion of `Denver` and `Omaha`, and Figure 12d to the tree after insertion of `Atlanta`.

To cope with data points that lie directly on one of the quadrant lines emanating from a data point, say P, we adopt the same conventions that were used for the grid method: the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. For example, in Figure 11, insertion of `Memphis` with coordinate values `(35,20)` would lead to its placement somewhere in quadrant `SE` of the tree rooted at `Chicago` (i.e., at `(35,42)`).

The amount of work expended in building a point quadtree is equal to the total path length (TPL) [72] of the tree as it reflects the cost of searching for all of the elements. Finkel and Bentley [37] have shown empirically that the TPL of a point quadtree under random insertion is roughly proportional to $N \cdot \log_4 N$, which yields an average cost $O(\log_4 N)$ of inserting, as well as searching for (i.e., a point query), a point. The extreme case is much worse (see Exercise 9) and is a function of the shape of the resulting point quadtree. This is dependent on the order in which nodes are inserted into it. The worst case arises when each successive node is the son of the currently deepest node in the tree. Consequently, there has been some interest in reducing the TPL. Two techniques for achieving this reduction are described below.

Finkel and Bentley [37] propose one approach that assumes that all the nodes are known a priori. They define an optimized point quadtree so that given a node A, no subtree of A accounts for more than one half of the nodes in the tree rooted at A. Building an optimized point quadtree from a file requires that the records in the file be sorted primarily by one key and secondarily by the other key. The root of the tree is set to the

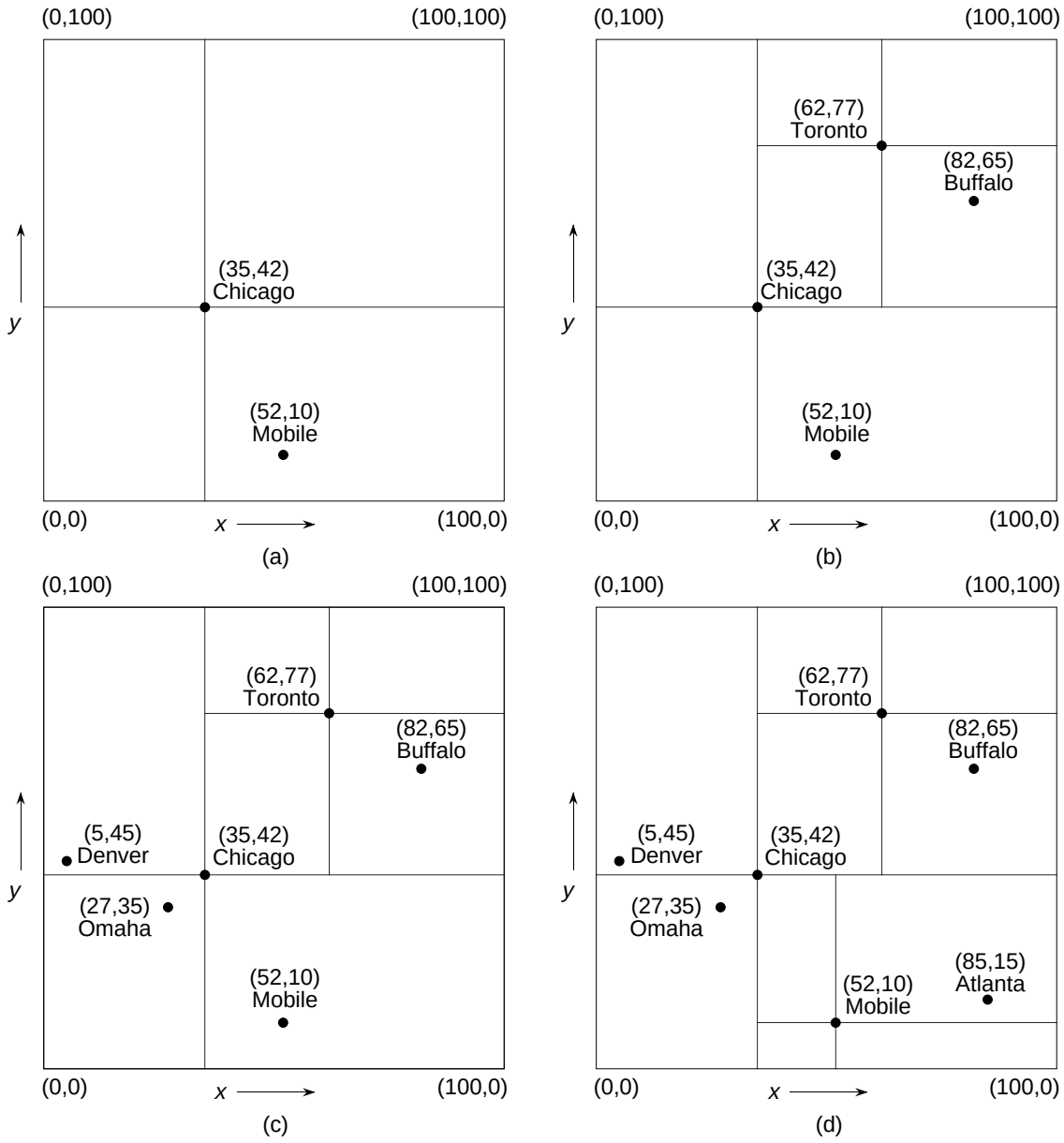


Figure 12: Sequence of partial block decompositions showing how a point quadtree is built when adding (a) Chicago and Mobile, (b) Toronto and Buffalo, (c) Denver and Omaha, and (d) Atlanta corresponding to the data of Figure 1.

median value of the sorted file and the remaining records are regrouped into four subcollections that will form the four subtrees of A. The process is applied recursively to the four subtrees.

The reason that this technique works is that all records preceding A in the sorted list will lie in the `NW` and `SW` quadrants (assuming that the x coordinate value serves as the primary key) and all records following A will lie in the `NE` and `SE` quadrants. Thus, the requirement that no subtree can contain more than one half of the total number of nodes is fulfilled. Of course, this construction method still does not guarantee that the resulting tree will be complete[13] (see Exercise 3).

[13] A t-ary tree containing N nodes is *complete* if we can map it onto a one-dimensional array so that the first element consists of the root, the next t elements are the roots of its t subtrees ordered from left to right, the next t^2 elements are the roots of all of the subtrees of the previous t elements again ordered from left to right, etc. This process stops once N is exhausted. Thus, each level of the tree, with

The optimized point quadtree requires that all of the data points are known a priori. Overmars and van Leeuwen [111] discuss an alternative approach which is a dynamic formulation of the above method — that is, the optimized point quadtree is built as the data points are inserted into it. The algorithm is similar to the one used to construct the conventional point quadtree, except that every time the tree fails to meet a predefined balance criterion, the tree is partially rebalanced using techniques similar to those developed by Finkel and Bentley to build the optimized point quadtree. For more details, see Exercise 6.

Exercises

Assume that each point in a point quadtree is implemented as a record of type *node* containing six fields. The first four fields contain pointers to the node's four sons corresponding to the directions (i.e., quadrants) `NW`, `NE`, `SW`, and `SE`. If P is a pointer to a node and I is a quadrant, then these fields are referenced as `SON`(P,I). We can determine the specific quadrant in which a node, say P, lies relative to its father by use of the function `SONTYPE`(P), which has a value of I if `SON`(`FATHER`(P),I)= P. `XCOORD` and `YCOORD` contain the values of the x and y coordinates, respectively, of the point. The empty point quadtree is represented by `NIL`.

1. Give an algorithm `PT_COMPARE` to determine the quadrant of a quadtree rooted at r in which point p lies.

2. Give an algorithm `PT_INSERT` to insert a point p in a point quadtree rooted at node r. Make use of procedure `PT_COMPARE` from Exercise 1. There is no need to make use of the `SONTYPE` function.

3. Suppose that you could construct the optimized point quadtree (i.e., minimal depth) for N nodes. What is the worst-case depth of an optimized point quadtree?

4. What is the maximum TPL in an optimized point quadtree?

5. Analyze the running time of the algorithm for constructing an optimized point quadtree.

6. In the text, we intimated that the optimized point quadtree can also be constructed dynamically. Given δ ($0 < \delta < 1$), we stipulate that every nonleaf node with a total of m nodes in its subtrees has at most $\lceil m/(2-\delta) \rceil$ nodes in each subtree. Give an algorithm to construct such a point quadtree.

7. Prove that the depth of the optimized point quadtree constructed in Exercise 6 is always at most $\log_{2-\delta} n + O(1)$ and that the average insertion time in an initially empty data structure is $O(\frac{1}{\delta \log_2^2 N})$. n denotes the number of nodes currently in the tree and N is the total number of nodes that have been inserted.

8. The TPL can also be reduced by applying balancing operators analogous to those used to balance binary search trees. Give the point quadtree analogs of the single and double rotation operators [73, p. 454].

9. What is the worst-case cost of building a point quadtree of N nodes?

4.1.2 Deletion

There are several ways of deleting nodes in two-dimensional point quadtrees. The first, suggested by Finkel and Bentley [37], is quite simple in that we reinsert all nodes of the tree rooted at the deleted node. This is usually a very expensive process, unless the deleted node is a leaf node or its sons are leaf nodes. In the rest of this section we describe a more complex, and more efficient, process developed by Samet [121]. It may be skipped on an initial reading.

---

the exception of the deepest level, contains a maximum number of nodes. The deepest level is partially full but has no empty positions when using this array mapping. For more details, see Knuth [72, pp. 400-401] as well as Section **??** of Chapter **??**.

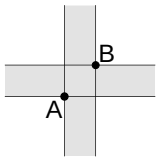


Figure 13: Idealized point quadtree deletion situation.

Ideally, we want to replace the deleted node (say A at (x_A, y_A)) with a node (say B at (x_B, y_B)), such that the region between the lines $x = x_A$ and $x = x_B$ and the region between the lines $y = y_A$ and $y = y_B$ are empty. The shaded area in Figure 13 illustrates this concept for nodes `A` and `B` in that `A` is deleted and replaced by `B`. We use the term *hatched* to describe the region that we would like to be empty. Unfortunately, finding a node that will lead to an empty hatched region involves a considerable amount of search. In fact, it is not uncommon that such a node fails to exist (e.g., when deleting node `A` in Figure 14).

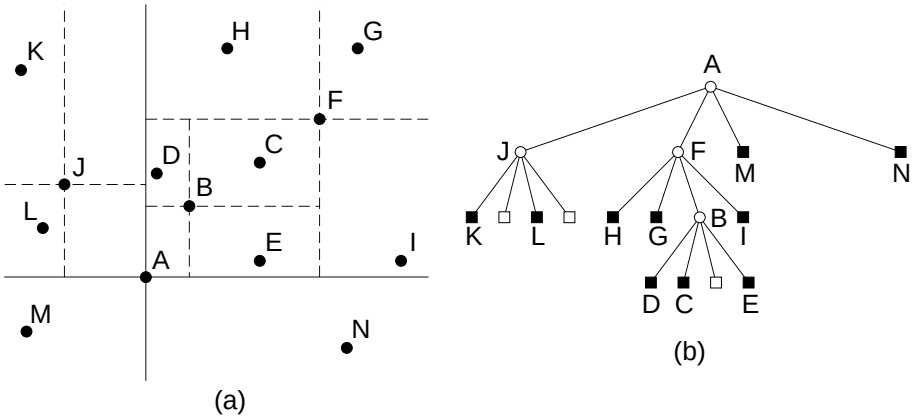


Figure 14: A point quadtree and the records it represents: (a) the resulting partition of space, and (b) the tree representation.

The algorithm we describe proceeds in a manner analogous to the method for binary search trees. For example, for the binary search tree of Figure 15, when node `A` is to be deleted, it can be replaced by one of nodes `D` or `G` — the two 'closest' nodes in value. In the case of a point quadtree (e.g., Figure 14), it is not clear which of the remaining nodes should replace `A`. This is because no node is simultaneously the closest in both the x and y directions. We notice that no matter which of the nodes is chosen, some of the nodes will assume different positions in the new tree. For example, in Figure 14, if `L` replaced `A`, then `J` would no longer occupy the `NW` quadrant with respect to the root (which is now `L` instead of `A`) and, thus, `J` would need to be reinserted, along with some of its subtrees, in the quadrant rooted at `F`. Similarly, `E` and `I` would have to be reinserted, along with some of their subtrees, in the quadrant rooted at `N`.

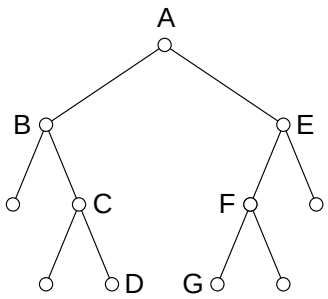


Figure 15: Example binary search tree.

Our algorithm determines four candidate nodes for the replacement node, one for each quadrant, as follows. Let the primitive OPQUAD(i) denote the quadrant that is 180 degrees apart from quadrant i (e.g., OPQUAD('NW')='SE'). The candidate in quadrant i of the root r is obtained by starting at the node SON(r,i) and repeatedly following the branch corresponding to OPQUAD(i) until encountering a node having no subtree along this branch. Figure 16 shows how node D is selected as the candidate node from the NE quadrant of node A. As a more complex example, in Figure 14, the candidates are B, J, M, and N.

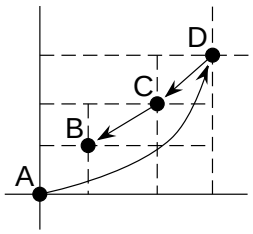


Figure 16: Example of the application of the FIND_CANDIDATEprocedure to the NE quadrant of node A.

Once the set of candidate nodes is found, an attempt is made to find the 'best' candidate, which becomes the replacement node. There are two criteria for choosing the best candidate. *Criterion 1* stipulates the choice of the candidate that is closer to each of its bordering axes than any other candidate which is on the same side of these axes, if such a candidate exists. For example, in Figure 14, node B is the best candidate according to this criterion. Note that the situation may arise that no candidate satisfies criterion 1 (e.g., Figure 17 or that several candidates satisfy it (e.g., B and D in Figure 18). In such a case, the candidate with the minimum L_1 metric value is chosen. This is called *criterion 2*. It is also known as the *city block metric* (or the *Manhattan metric*).

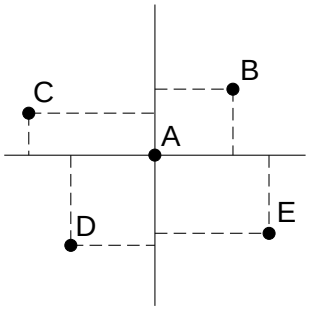


Figure 17: Example of a point quadtree with no 'closest' terminal node.

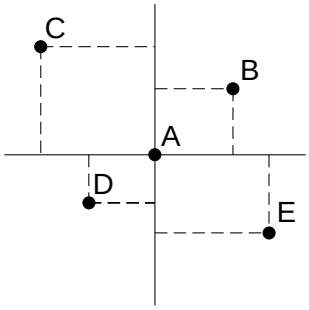


Figure 18: Example of a point quadtree with two nodes being 'closest' to their bordering axes.

The L_1 metric is the sum of the displacements from the bordering x and y axes. To justify its use, we

assume that the nodes are uniformly distributed in the two-dimensional space. Our goal is to minimize the area which is obtained by removing from the hatched region the rectangle whose opposite vertices are the root and candidate nodes (e.g., nodes A and B, respectively, in Figure 13).

Assume that the two-dimensional space is finite, having sides of length L_x and L_y parallel to the x and y axes, respectively. Let this space be centered at the node to be deleted. Assume further that the candidate node is at a distance of d_x and d_y from the x and y axes, respectively. Under these assumptions, the remaining area is $L_x \cdot d_y + L_y \cdot d_x - 2 \cdot d_x \cdot L_y$ (see Figure 19). We can ignore the area of the rectangle with sides d_x and d_y because the candidate selection process guarantees that it is empty. As L_x and L_y increase, as occurs in the general problem domain, the contribution of the $2 \cdot d_x \cdot d_y$ term becomes negligible and the area is proportional to the sum of d_x and d_y[14].

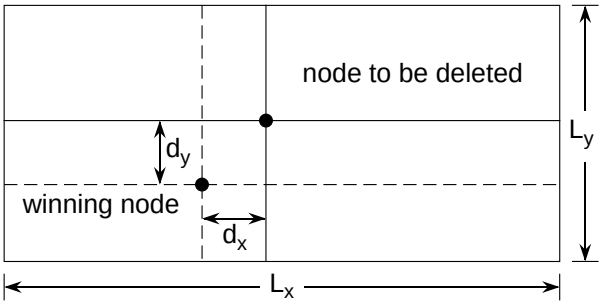


Figure 19: Example of a two-dimensional space.

Criterion 2 is not sufficient by itself to insure that the selected candidate partitions the space so the hatched region contains no other candidate. For example, see Figure 20, in which O has been deleted and A satisfies criterion 2 but only C satisfies criterion 1. A pair of axes through C leaves all other candidates outside the hatched region, while a pair of axes through A results in B being in the hatched region.

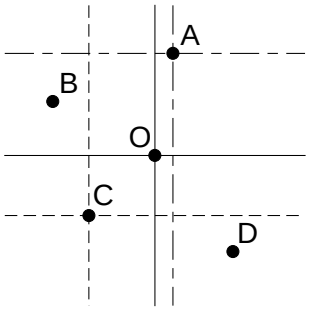


Figure 20: Example of the insufficiency of criterion 2 for an empty hatched region.

If no candidate is found to satisfy criterion 1, then criterion 2 guarantees that at least one of the candidates, say X, has the property that no more than one of the remaining candidates occupies the hatched region between the original axes and the axes passing through X. To see this, we examine Figure 17 and note that whichever candidate is selected to be the new root (say C in the NW quadrant), then the candidate in the opposite quadrant (i.e., SE) lies outside of the hatched region. In addition, the candidate in a quadrant on the same side of an axis as is C, and to which axis C is closer (i.e., B) lies outside of the hatched region.

We are now ready to present the deletion algorithm. It makes use of the properties of the space obtained by the new partition to reduce the number of nodes requiring reinsertion. The algorithm consists of two procedures ADJQUAD and NEWROOT. Let A be the node to be deleted and let I be the quadrant of the tree containing B, the replacement node for A. Note that no nodes in quadrant OPQUAD(I) need to be reinserted. Now, separately

[14] Actually, this statement only holds for $L_x = L_y$. However, this approximation is adequate for the purpose of this discussion.

process the two quadrants adjacent laterally to quadrant I using procedure ADJQUAD, followed by application of procedure NEWROOT to quadrant I.

Procedure ADJQUAD proceeds as follows. Examine the root of the quadrant, say R. If R lies outside of the hatched region, then two subquadrants can automatically remain in the quadrant and need no further processing while the remaining subquadrants are separately processed by a recursive invocation of ADJQUAD. Otherwise, the entire quadrant must be reinserted in the quadtree which was formerly rooted at A.

As an example of the effect of applying ADJQUAD to the two quadrants laterally adjacent to the quadrant containing the replacement node, consider Figure 14 where node A is deleted and replaced by node B in the NE quadrant. J and the subquadrant rooted at K remain in the NW quadrant, while the subquadrant rooted at L is recursively processed. Eventually, L must be reinserted in the tree rooted at M. The SE quadrant of A (rooted at N) does not require reinsertion. The result of these manipulations is given in Figure 21 which shows the outcome of the entire deletion. Below, we explain how the contents of the quadrant containing the replacement node have been processed (i.e., I or NE in the example of Figure 14).

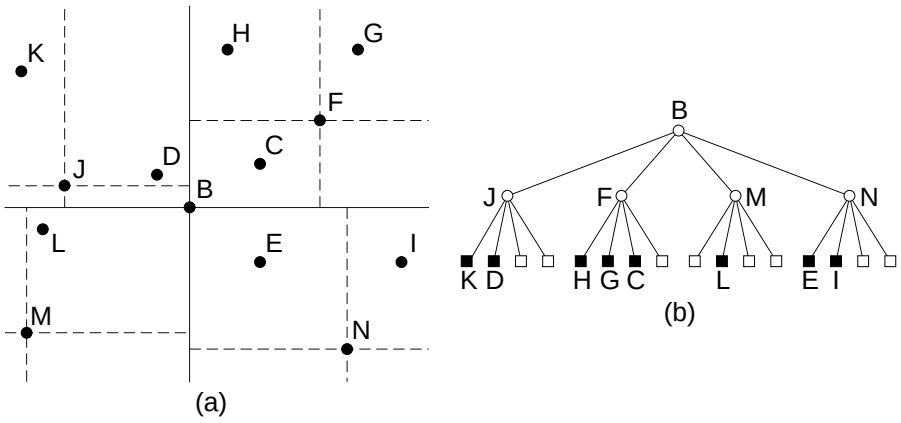


Figure 21: Result of deleting node A from the point quadtree of Figure 14 and replacing it with node B: (a) the resulting partition of space, and (b) the tree representation.

Once the nodes in the quadrants adjacent to quadrant I have been processed by ADJQUAD we apply procedure NEWROOT to the nodes in I. Clearly, all of the nodes in subquadrant I of I will retain their position. Thus NEWROOT must only be applied to the remaining subquadrants of I. NEWROOT starts by invoking ADJQUAD to process the subquadrants adjacent laterally to subquadrant I. This is followed by an iterative reinvokation of NEWROOT to subquadrant OPQUAD(I). This iterative reinvokation process continues until an empty link in direction OPQUAD(I) is encountered (i.e., at this point we are at B, the node replacing the deleted node). Now, insert the nodes in the subquadrants adjacent to subquadrant I of the tree rooted at B in the quadrants adjacent to quadrant I of the tree rooted at A. Recall that by virtue of the candidate selection process, subquadrant OPQUAD(I) of the tree rooted at B is empty. Also, subquadrant I of the tree rooted at B replaces subquadrant OPQUAD(I) of the previous father node of B.

Figure 22 illustrates the subquadrants that NEWROOT calls ADJQUAD (labeled ADJ) to process when node 0 is deleted and the resulting tree is rooted at node 4. For the example of Figure 14 where node A is deleted and whose result is shown in Figure 21, application of NEWROOT results in the tree rooted at G being left alone. Trees rooted at H and I are processed by ADJQUAD. Trees rooted at D and E are reinserted in quadrants NW and SE, respectively. The tree rooted at C replaces B as the son of F in subquadrant SW.

Theoretical and empirical results for the above deletion method are described by Samet [121]. It is shown theoretically that for data that is uniformly distributed, the average number of nodes requiring reinsertion is reduced by a factor of 5/6 (i.e., 83%) when the replacement node satisfies criteria 1 and 2. The relaxation of the requirement that the replacement node must satisfy criteria 1 and 2, and its substitution by a selection at random of one of the candidates as the replacement node, caused the factor by which the average number of

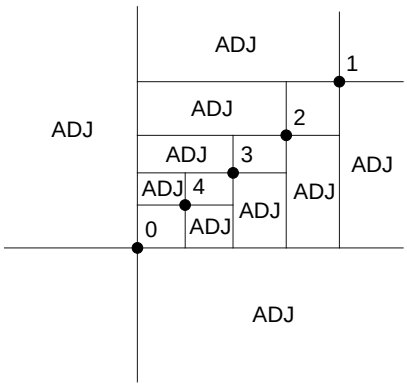


Figure 22: Subquadrants processed by procedure ADJQUAD (labeled ADJ) when node 0 is deleted and replaced by node 4.

nodes required reinsertion to decrease to 2/3 (i.e., 67%). Of course, the candidate selection process becomes considerably simpler in this case.

The empirical tests led to the following interesting observations. First, the number of comparison operations is proportional to $\log_4 N$ versus a considerably larger factor when using the deletion method of Finkel and Bentley [37]. Second, the total path length of the tree after deletion using Samet's method [121] decreases slightly, whereas when Finkel and Bentley's method [37] is used, the total path length increases significantly. This data is important because it correlates with the effective search time (see also [16, 84]). In other words, the smaller the total path length, the faster a node can be accessed.

It is interesting to observe that one of the main reasons for the complexity of deletion in point quadtrees is the fact that the data points also serve to partition the space from which they are drawn. The *pseudo quadtree* of Overmars and van Leeuwen [111] simplifies deletion by using arbitrary points, not in the set of data points being represented, for the partitioning process. The pseudo quadtree is constructed by repeatedly partitioning the space into quadrants, subquadrants, etc., until each subquadrant contains at most one data point of the original set. This means that the data points occur as leaf nodes of the pseudo quadtree. The partition points are chosen in a manner that splits the remaining set in the most balanced way.

Overmars and van Leeuwen show that for any N data points in a d-dimensional space, there exists a partitioning point such that every quadrant contains at most $\lceil N/(d+1) \rceil$ data points. They also demonstrate that the resulting pseudo quadtree has a depth of at most $\lceil \log_{d+1} N \rceil$ and can be built in $O(N \cdot \log_{d+1} N)$ time. For example, Figure 23 is the pseudo quadtree corresponding to the data of Figure 1. Efficient deletion and insertion in a pseudo quadtree requires that the bound on the depth be weakened slightly (see Exercise 15).

Exercises

1. Assuming the same quadtree node implementation as in the Exercises in Section 4.1.1, and that p points to the son in quadrant q of the node to be deleted, give an algorithm `FIND_CANDIDATE` to find the candidate in quadrant q.

2. Assuming the same quadtree node implementation as in the Exercises in Section 4.1.1, give an algorithm `PT_DELETE` for deleting a point stored in a node p from a point quadtree rooted at node r. The node does not have a `FATHER` field although you may make use of the `SONTYPE` function. Make use of procedure `PT_COMPARE` from Exercise 1. Procedure `PT_DELETE` implements procedures `ADJQUAD` and `NEWROOT` as described in the text. You can simplify the task by making use of primitives `CQUAD`(q) and `CCQUAD`(q) to yield the adjacent quadrants in the clockwise and counterclockwise directions, respectively, to quadrant q. Also, assume the existence of procedure `INSERT_QUADRANT`(t,p) which inserts the nodes of the subtree rooted at node t in the subtree rooted at node p.

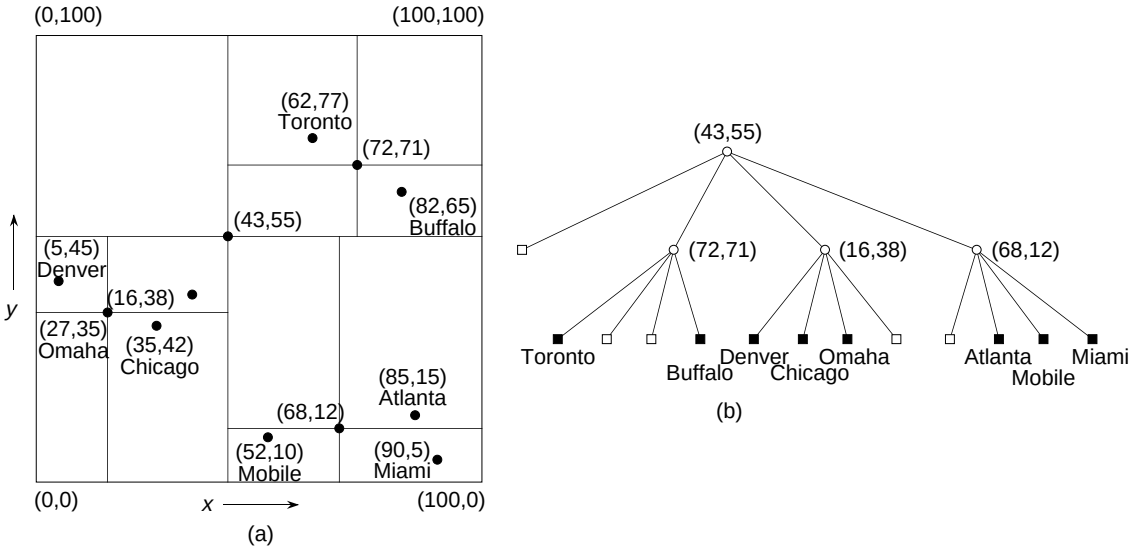


Figure 23: A pseudo quadtree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

3. Suppose that a point quadtree node is implemented as a record with a `FATHER` field containing a pointer to its father. Modify procedure `PT_DELETE` in the solution to Exercise 2 to take advantage of this additional field.

4. Let T be a tree containing N nodes with TPL(T) denoting its total path length. Prove that the sum of the sizes of its subtrees is TPL(T)+N. Show that this result holds for binary search trees and point quadtrees as well.

5. Define a *nontrivial* subtree to be a subtree with two or more nonempty subtrees. Let $Q(N)$ be the expected nontrivial subtree size in a complete point quadtree of N nodes. Prove that as N gets large, $Q(N)$ is $4 \cdot \log_4 \left(\frac{3}{4} \cdot N\right) - 4/3$. Do not take the root of the subtree into account. Note that $Q(N)$ is the cost of deletion when the method of Finkel and Bentley [37] is used. A complete point quadtree is used because such a configuration minimizes the average cost of deletion in this case since, recalling Exercise 4, the sum of the subtree sizes for a given tree T containing N nodes is TPL(T)+N. This quantity is at a minimum when T is a complete tree.

6. Assuming a complete point quadtree of N nodes, let $r(N)$ denote the proportion of nodes that do not require reinsertion when using the deletion method of Samet [121]. Let A and B be the deleted and replacement node respectively where B is a node returned by the candidate selection procedure (see procedure `FIND_CANDIDATE` in Exercise 1). Assume further that the nodes are partitioned uniformly throughout the partition of the two-dimensional space rooted at A. Use values of 1/2 for the probability that a node needs to be reinserted. Show that when B satisfies criteria 1 and 2, then $r(N)$ is 5/6 or 3/4 depending on whether none of the adjacent quadrants have their candidate replacement node in the hatched region or not. Similarly, show that $r(N)$ is 2/3 when the replacement node is chosen at random from the set of nodes returned by procedure `FIND_CANDIDATE`.

7. Can you prove that the point quadtree deletion algorithm encoded given by procedures `ADJQUAD` and `NEWROOT` is $O(\log_4 N)$?

8. Why do the resulting trees get bushier when the point quadtree node deletion algorithm given by procedures `ADJQUAD` and `NEWROOT` is used?

9. Extend the deletion method of Samet [121] to handle d-dimensional point quadtrees and compute $r(N)$.

10. Write an algorithm, `BUILD_PSEUDO_QUADTREE`, to construct a pseudo quadtree for two-dimensional data.

11. Given a set of N points in d-dimensional space, prove that there exists a partitioning point such that every quadrant contains $\lceil N/(d+1) \rceil$ points.

12. Given a set of N points in d-dimensional space, prove that there exists a pseudo quadtree for it with a depth of at most $\lceil \log_{d+1} N \rceil$.

13. Show that the upper bound obtained in Exercise 12 is a strict upper bound in that there exists a configuration of N points such that there is no corresponding pseudo quadtree with depth less than $\lceil \log_{d+1} N \rceil$.

14. Prove that the pseudo quadtree of Exercise 12 can be built in $O(N \cdot \log_{d+1} N)$ time.

15. Prove that for any fixed δ ($0 < \delta < 1$) there is an algorithm to perform N insertions and deletions in an initially empty d-dimensional pseudo quadtree such that its depth is always at most $\log_{d+1-\delta} n + O(1)$ and that the average transaction time is bounded by $O(\frac{1}{\delta \log_2^2 N})$. n denotes the number of nodes currently in the tree.

4.1.3 Search

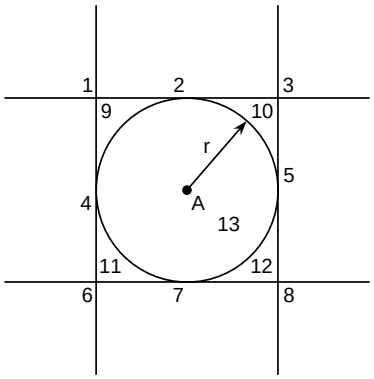
The point quadtree, as well as the trie-based quadtree, is suited for applications that involve proximity search. A typical query is one that requests the determination of all nodes within a specified distance of a given data point — such as, all cities within 50 miles of Washington, D.C. Since the mechanics of the search are the same for both point and trie-based quadtrees, in the rest of this discussion we use the term *quadtree* unless a distinction between the different quadtrees is necessary, in which case we restore the appropriate qualifier. The efficiency of the quadtree data structure lies in its role as a pruning device on the amount of search that is required. Thus, many records will not need to be examined.

For example, suppose that in the hypothetical database of Figure 1 we wish to find all cities within eight units of a data point with coordinate values `(83,10)`. In such a case, there is no need to search the `NW`, `NE`, and `SW` quadrants of the root (i.e., `Chicago` located at `(35,42)`). Thus, we can restrict our search to the `SE` quadrant of the tree rooted at `Chicago`. Similarly, there is no need to search the `NW` and `SW` quadrants of the tree rooted at `Mobile` (i.e., located at `(52,10)`).

As a further illustration of the amount of pruning of the search space that is achievable by use of the quadtree, we make use of Figure 24. In particular, given the problem of finding all nodes within radius `r` of point `A`, use of the figure indicates which quadrants need not be examined when the root of the search space, say R, is in one of the numbered regions. For example, if R is in region `9`, then all but its `NW` quadrants must be searched. If R is in region `7`, then the search can be restricted to the `NW` and `NE` quadrants of R.

Similar techniques can be used to search for data points in any connected figure. For example, Finkel and Bentley [37] give algorithms for searching within a rectangular window of arbitrary size. These algorithms are more general than the query that we examined here as they are applicable to both locational and nonlocational data. For example, such a a query would seek to find all cities within 50 miles of the latitude position of Washington, DC and within 50 miles of the longitude position of Washington, DC. The difference between these two formulations of the query is that the former admits a circular search region while the latter admits a rectangular search region. In particular, the latter query is applicable to both locational and nonlocational data while the former is only applicable to locational data. As a concrete example, suppose that we wish to find all people of height between 5 and 6 feet (152.4 and 182.88 cm) and of weight between 150 and 180 pounds (68 and 82 kg).

To handle more complex search regions such as those formed by arbitrary hyperplanes (rather than ones that are parallel to one of the attribute axes as in our examples) as well as convex polygons, Willard [146] defines a *polygon tree* where the x-y plane is subdivided by J lines that need not be orthogonal, although



Problem: Find all nodes within radius r of point A
Solution: If the root is in region I (I=1…13), then continue to search in the quadrant specified by I

1.	SE	6.	NE	11.	All but SW
2.	SE, SW	7.	NE, NW	12.	All but SE
3.	SW	8.	NW	13.	All
4.	SE, NE	9.	All but NW		
5.	SW, NW	10.	All but NE		

Figure 24: Relationship between a circular search space and the regions in which a root of a quadtree may reside.

there are other restrictions on these lines (see Exercise 6). When $J = 2$, the result is a point quadtree with nonorthogonal axes.

Thus, we see that quadtrees can be used to handle all three types of queries specified by Knuth [73]. Range and Boolean queries are described immediately above while simple queries (e.g., what city is located at a given pair of coordinate values) are a byproduct of the point quadtree insertion process described earlier. Nearest neighbor queries as well as k-nearest neighbor queries are also feasible [59].

The cost of search in a point quadtree has been studied by Bentley, Stanat, and Williams [16] (see Exercise 2) and Lee and Wong [84]. In particular, Lee and Wong show that in the worst case, range searching in a complete two-dimensional point quadtree takes $O(2 \cdot N^{1/2})$ time. This result can be extended to d dimensions to take $O(d \cdot N^{1-1/d})$ time, and is derived in a manner analogous to that for a k-d tree as discussed in Section 5. Note that complete point quadtrees are not always achievable as seen in Exercise 3 in Section 4.1.1. Partial range queries can be handled in the same way as range searching. Lee and Wong [84] show that when ranges for s out of d keys are specified, then the algorithm has a worst case running time of $O(s \cdot N^{1-1/d})$.

Exercises

1. Write a procedure `PT_REGION_SEARCH` which performs a region search for a circular region in a point quadtree. Repeat for a rectangular region.

2. Bentley and Stanat [15] define a *perfect point quadtree* of height m as a point quadtree that (1) has 4^i nodes at level $m - i$ where the root is at level m, $0 \leq i \leq m - 1$, and (2) every node is at the centroid of the finite space spanned by its subtrees. Assume a search space of $[0, 1]^2$ and a rectangular search region of sides of length x and y where x and y are in [0,1]. Find the expected number of nodes that are visited in a perfect point quadtree of height m when a region search procedure is applied to the above search region.

3. Bentley and Stanat [15] also define the concept of *overwork* of a search algorithm as the difference between the number of records visited and the number of records in the search region. Assume a search space of $[0, 1]^2$ with N records and a square search region of side length x where x is in [0,1]. Compare

the amount of overwork for an inverted list (i.e., inverted file) and a perfect point quadtree.

4. Prove that the worst case running time for a partial range query such that ranges for s out of d keys are specified is $O(s \cdot N^{1-1/d})$.

5. Perform an average case analysis for a region query in a point quadtree.

6. What are the restrictions on the choice of subdivision lines in Willard's polygon tree [146]?

4.2 Trie-based Quadtrees

From Section 4.1 we saw that for the point quadtree, the points of decomposition are the data points themselves (e.g., in Figure 11, `Chicago` at location `(35,42)` subdivides the two-dimensional space into four rectangular regions). Requiring that the regions resulting from the subdivision process be of equal size (congruent, to be precise) leads to a trie-based quadtree.

Trie-based quadtrees are closely related to the region quadtree representation of region data (see Section **??** in Chapter **??**). Region quadtrees are built on the basis of recursively decomposing a region into four congruent blocks and stopping the decomposition whenever the block is homogeneous. In the case of region data, the homogeneity condition usually means that the area spanned by the block belongs to just one region (assuming that the underlying regions do not overlap). For example, Figure 25 shows a region and its corresponding region quadtree. There are many possible adaptations of the homogeneity condition for point data. For example, if we treat the points as elements in a square matrix, then the homogeneity condition could be that the block contains no more than b nonzero elements (when $b = 1$, we have a PR quadtree). Alternatively, we could stipulate that all elements in the block must be zero, or that the block contains just one element and this single element must be nonzero (an MX quadtree).

Although conceivably there are many other ways of adapting the region quadtree to represent point data, our presentation focuses on the MX and PR quadtrees. In particular, we discuss the MX quadtree in Section 4.2.1 and the PR quadtree in Section 4.2.2. We outline how they are created and updated as well as give a brief review of some of their applications. We omit a discussion of search operations because they are performed in the same manner as in point quadtree (see Section 4.1.3).

4.2.1 MX Quadtrees

There are a number of ways of adapting the region quadtree to represent point data. If the domain of the data points is discrete and all attribute values are discrete and have the same type and range, then we can treat the data points as if they were `black` pixels in a region quadtree corresponding to a square image. An alternative characterization is to treat the data points as nonzero elements in a square matrix. We shall use this characterization in the subsequent discussion. In fact, we use the term *MX quadtree* to describe the structure on account of the analogy to the matrix (where *MX* denotes matrix), although the term *MX quadtrie* would probably be more appropriate.

The MX quadtree is organized in a similar way to the region quadtree. The difference is that leaf nodes are `black` ccorresponding to regions of size 1×1, or empty (i.e., `white` and represented by `NIL`) in which case there is no restriction on their size. The leaf nodes correspond to the presence or absence, respectively, of a data point in the appropriate position in the matrix. For example, Figure 26 is the $2^3 \times 2^3$ MX quadtree corresponding to the data of Figure 1. It is obtained by applying the mapping f such that $f(z) = z \div 12.5$ to both x and y coordinate values. The result of the mapping is reflected in the coordinate values in the figure.

Each data point in an MX quadtree corresponds to a 1×1 square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left corner of the square. This adheres to the general convention followed throughout the text that the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. We also assume that the lower left corner of the matrix is located at (0,0). Note that, unlike the region quadtree, when a nonleaf node has

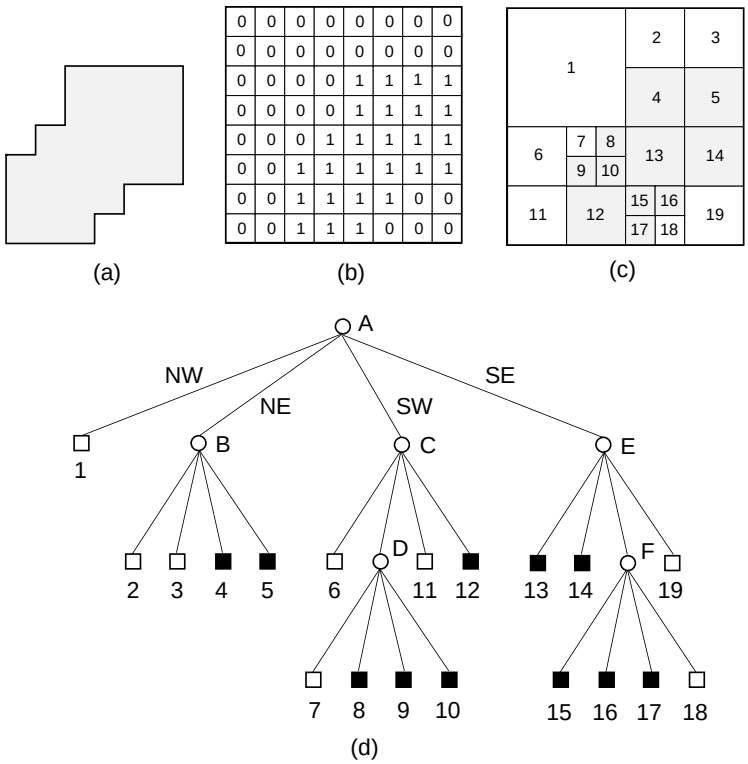


Figure 25: (a) Sample region, (b) its binary array representation, (c) its maximal blocks with the blocks in the region being shaded, and (d) the corresponding quadtree.

four `black` sons, they are not merged. This is natural since a merger of such nodes would lead to a loss of the identifying information about the data points, as each data point is different. On the other hand, the empty leaf nodes have the absence of information as their common property, so, four `white` sons of a nonleaf node can be safely merged.

Data points are inserted into an MX quadtree by searching for them. This search is based on the location of the data point in the matrix (e.g., the discretized values of its x and y coordinates in our city database example). It is achieved by recursively descending the tree while comparing the location of the point with the implied coordinate values[15] of the root of the subtree being descended. The result of this comparison indicates which of the four subtrees is to be descended next. The search terminates when we have encountered a leaf node. If this leaf node is occupied by another record s, then s is replaced by a record corresponding to the data point being inserted (although the location is the same, some of the nonlocational attribute information may have changed). If the leaf node is `NIL`, then we may have to repeatedly subdivide the space spanned by it until it is a 1×1 square. This will result in the creation of nonleaf nodes. This operation is termed *splitting*. For a $2^n \times 2^n$ MX quadtree, splitting will have to be performed at most n times.

The shape of the resulting MX quadtree is independent of the order in which data points are inserted into it. However, the shapes of the intermediate trees do depend on the order. For example, Figure 27a-d shows how the tree was constructed in incremental fashion for `Chicago`, `Mobile`, `Toronto`, and `Buffalo`by giving the appropriate block decompositions.

Deletion of nodes from MX quadtrees is considerably simpler than for point quadtrees since all records are stored in the leaf nodes. This means that we don't have to be concerned with rearranging the tree as is necessary when a record stored in a nonleaf node is being deleted from a point quadtree. For example, to delete `Omaha`

[15] The coordinate values are implied because they are not stored explicitly as they can be derived from knowledge of the width of the space spanned by the MX quadtree and the path from the root that has been descended. This is one of the properties of a trie.

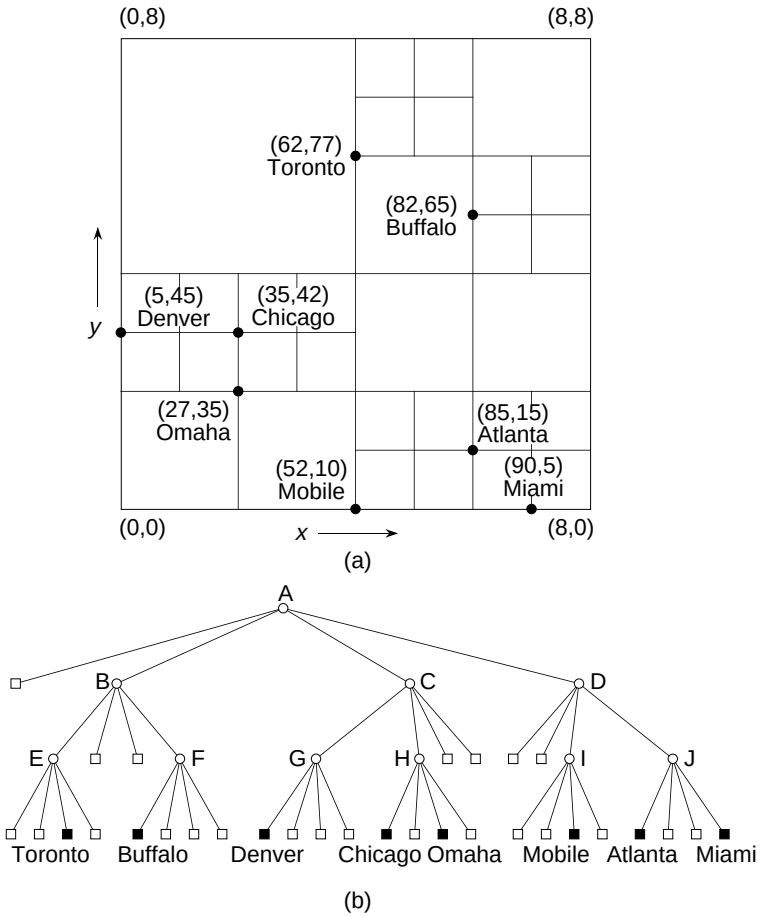


Figure 26: An MX quadtree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

from the MX quadtree of Figure 26 we simply set the `SW` son of its father node, `H`, to `NIL`. Deleting `Toronto` is slightly more complicated. Setting the `SW` son of its father node, `E`, to `NIL` is not enough. We must also perform merging because all four sons of `E` are `NIL` after deletion of `Toronto`. This leads to replacing the `NW` son of `E`'s father, `B`, by `NIL` and returning nonleaf node `E` to the free storage list. This process is termed *collapsing* and is the counterpart of the splitting operation that was necessary when inserting a record in an MX quadtree.

Collapsing may take place over several levels. In essence, we apply the collapsing process repeatedly until encountering a nearest common ancestor with an additional non-`NIL` son. As collapsing takes place, the affected nonleaf nodes are returned to the free storage list. For example, suppose that after deleting `Toronto` from Figure 26 we also delete `Buffalo`. The subsequent deletion of `Buffalo` means that nodes `F` and `B` are subjected to the collapsing process with the result that the `NE` son of node `A` is set to `NIL` and nonleaf nodes `B` and `F` are returned to the free storage list. The execution time of the deletion process is bounded by two times the depth of the quadtree. This upper bound is attained when the nearest common ancestor for the collapsing process is the root node.

Performing a range search in an MX quadtree is done in the same way as in the point quadtree. The worst-case cost of searching for all points that lie in a rectangle whose sides are parallel to the quadrant lines is $O(F + 2^n)$ where F is the number of points found and n is the maximum depth (see Exercise 5).

The MX quadtree is used in a number of applications. It can serve as the basis of a quadtree matrix manipulation system (see Exercises 6–10). The goal is to take advantage of the sparseness of matrices to achieve space and execution time efficiencies (e.g., [1, 148, 149]. Algorithms for matrix transposition and matrix mul-

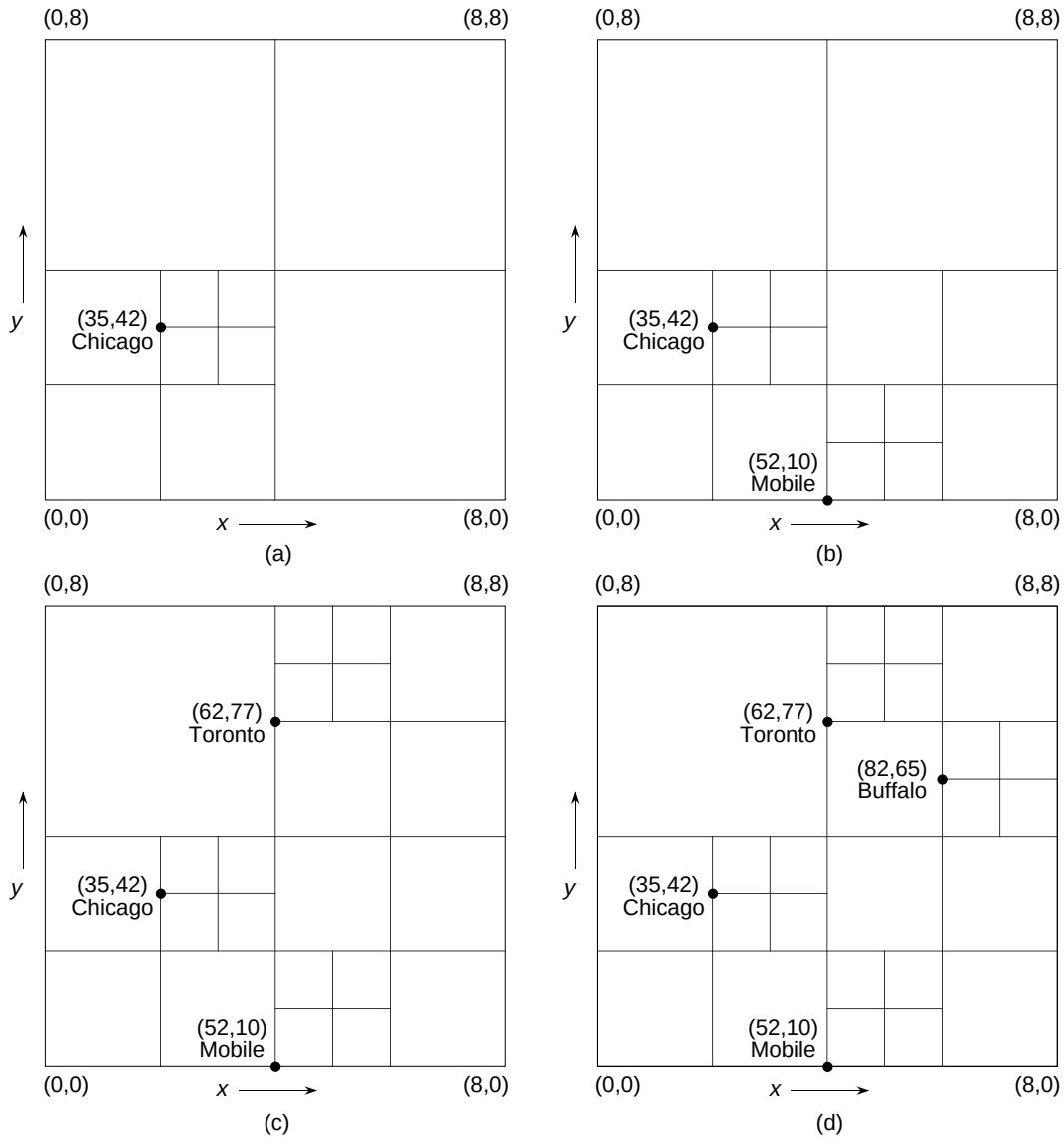


Figure 27: Sequence of partial block decompositions showing how an MX quadtree is built when adding (a) Chicago, (b) Mobile, (c) Toronto, and (d) Buffalo corresponding to the data of Figure 1.

tiplication that take advantage of the recursive decomposition are easy to develop. Letelier [85] makes use of the MX quadtree to represent silhouettes of hand motions to aid in the telephonic transmission of sign language for the hearing impaired. The region quadtree formulation of Hunter and Steiglitz [61, 63] (see Sections **??** of Chapter **??** and **??** of Chapter **??**) utilizes a three color variant of the MX quadtree to represent digitized simple polygons. De Coulon and Johnsen [25] describe the use of the MX quadtree in the coding of black and white facsimiles for efficient transmission of images.

Exercises

In the following exercises assume that each point in an MX quadtree is implemented as a record of type *node* containing five fields. The first four fields contain pointers to the node's four sons corresponding to the directions (i.e., quadrants) `NW`, `NE`, `SW`, and `SE`. If P is a pointer to a node and I is a quadrant, then these

fields are referenced as SON(P,I). The fifth field, NODETYPE, indicates whether the node contains a data point (BLACK), is empty (WHITE), or is a nonleaf node (GRAY). There is no need to store information about the coordinate values of the data point since this is derivable from the path to the node from the root of the MX quadtree. The empty MX quadtree is represented by NIL.

1. Give an algorithm MX_COMPARE to determine the quadrant of a block of width $2 \times w$ centered at (w, w) in which a point (x, y) lies.

2. Give an algorithm MX_INSERT for inserting a point p in an MX quadtree rooted at node r that spans a space of width w. Make use of procedure MX_COMPARE from Exercise 1.

3. Give an algorithm MX_DELETE for deleting data point (x, y) from an MX quadtree rooted at node r that spans a space of width w. Make use of procedure MX_COMPARE from Exercise 1.

4. Write an algorithm to perform a range search for a rectangular region in an MX quadtree.

5. Show that the worst-case cost of performing a range search in an MX quadtree is $O(F + 2^n)$, where F is the number of points found and n is the maximum depth. Assume a rectangular query region.

6. Letting a_{ij} and b_{ij} denote the elements of row i and column j of matrices A and B, respectively, then the transpose of matrix A is the matrix B with $b_{ij} = a_{ji}$. Give an algorithm to transpose a matrix represented by an MX quadtree.

7. How many interchange operations are needed to transpose an MX quadtree representation of a $2^n \times 2^n$ matrix so that it is not sparse (i.e., all blocks are of size 1)?'

8. Compare the savings in space and time requirements when a matrix is represented as an MX quadtree and as an array. Use the time required to perform a transpose operation as the basis of the comparison. You should assume the worst case which occurs when there is no sparseness (i.e., all blocks are of size 1).

9. Give an algorithm for multiplying two matrices stored as MX quadtrees.

10. Unless there is a large number of square blocks of zeros, the MX quadtree is not a particularly attractive representation for matrix multiplication. The problem is that this is inherently a row and column operation. One possible alternative is to make use of a runlength representation (see Section **??** of Chapter **??**). This is a one-dimensional aggregation of consecutive identically-valued elements into blocks. The problem is that we would ideally like to have both the rows and columns encoded using this technique. Can you see a problem with using such a dual representation?

4.2.2 PR Quadtrees

The MX quadtree is feasible as long as the domain of the data points is discrete and finite. If this is not the case, then the data points can not be represented using the MX quadtree because the minimum separation between the data points is unknown. This observation leads to an alternative adaptation of the region quadtree to point data, which associates data points (which need not be discrete) with quadrants. We call it a *PR quadtree* (P for point and R for region) although the term *PR quadtrie* would probably be more appropriate.

The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., white) or contain a data point (i.e., black) and its coordinate values. A leaf node's corresponding block contains at most one data point. For example, Figure 28 is the PR quadtree corresponding to the data of Figure 1. To cope with data points that lie directly on one of the quadrant lines emanating from a subdivision point, we adopt the convention that the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. For example, in Figure 28, a point located at (50,30) would lie in quadrant SE of the tree rooted at (50,50).

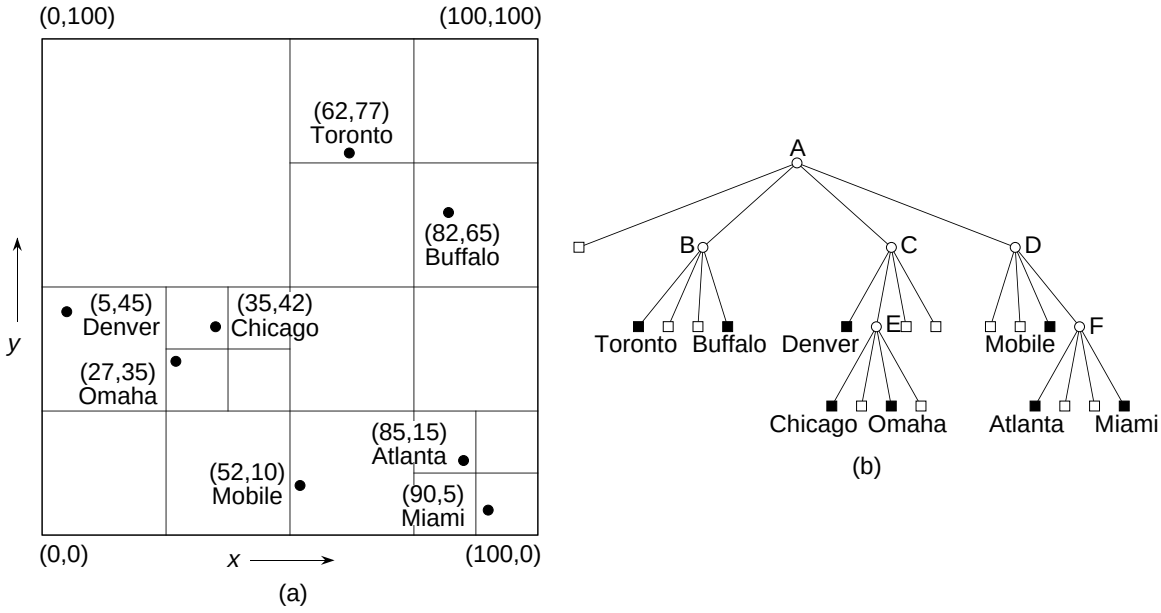


Figure 28: A PR quadtree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

A record r corresponding to data point (a, b) is inserted into a PR quadtree by searching for it. This is achieved by recursively descending the tree by comparing the location of the point with the location of the root of the subtree being descended. Note that the locations of the roots of the subtrees are not stored explicitly in the tree as they can be derived from knowledge of the space spanned by the PR quadtree and the path from the root that has been descended so far (this is one of the properties of a trie). If such a node t exists (i.e., containing (a, b)), then r replaces the record associated with t.

Actually, we don't really search for r. Instead, we search for the block h in which r belongs (i.e., a leaf node). If h is already occupied by another record s with different x and y coordinate values, say (c, d), then we must subdivide the block repeatedly (termed *splitting*) until records r and s no longer occupy the same block. This may cause a large number of subdivision operations, especially if the two points (a, b) and (c, d) are both contained in a very small quadtree block. A necessary, but not sufficient, condition for this situation to arise is that the Euclidean distance between (a, b) and (c, d) is very small. As a result, we observe that every nonleaf node in a PR quadtree of a data set consisting of more than one data point has at least two descendant leaf nodes that contain data points.

It should be clear that the shape of the resulting PR quadtree is independent of the order in which data points are inserted into it. However, the shapes of the intermediate trees do depend on the order. For example, the tree in Figure 28 was built for the sequence `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`. Figure 29a-d shows how the tree was constructed in an incremental fashion for `Chicago`, `Mobile`, `Toronto`, and `Buffalo` by giving the appropriate block decompositions.

Deletion of nodes in PR quadtrees is considerably simpler than for point quadtrees since all records are stored in the leaf nodes. This means that there is no need to be concerned with rearranging the tree as is necessary when records stored in nonleaf nodes are being deleted from point quadtrees. For example, to delete `Mobile` from the PR quadtree in Figure 28 we simply set the `SW` son of its father node, `D`, to `NIL`.

Continuing this example, deleting `Toronto` is slightly more complicated. Setting the `NW` son of its father node, `B`, to `NIL` is not enough since now we violate the property of the PR quadtree that each nonleaf node in a PR quadtree of more than one record has at least two descendant leaf nodes that contain data points. Thus, we must also reset the `NE` son of `A` to point to `Buffalo` and return nonleaf node `B` to the free storage list. This process is termed *collapsing* and is the counterpart of the splitting operation that was necessary when inserting

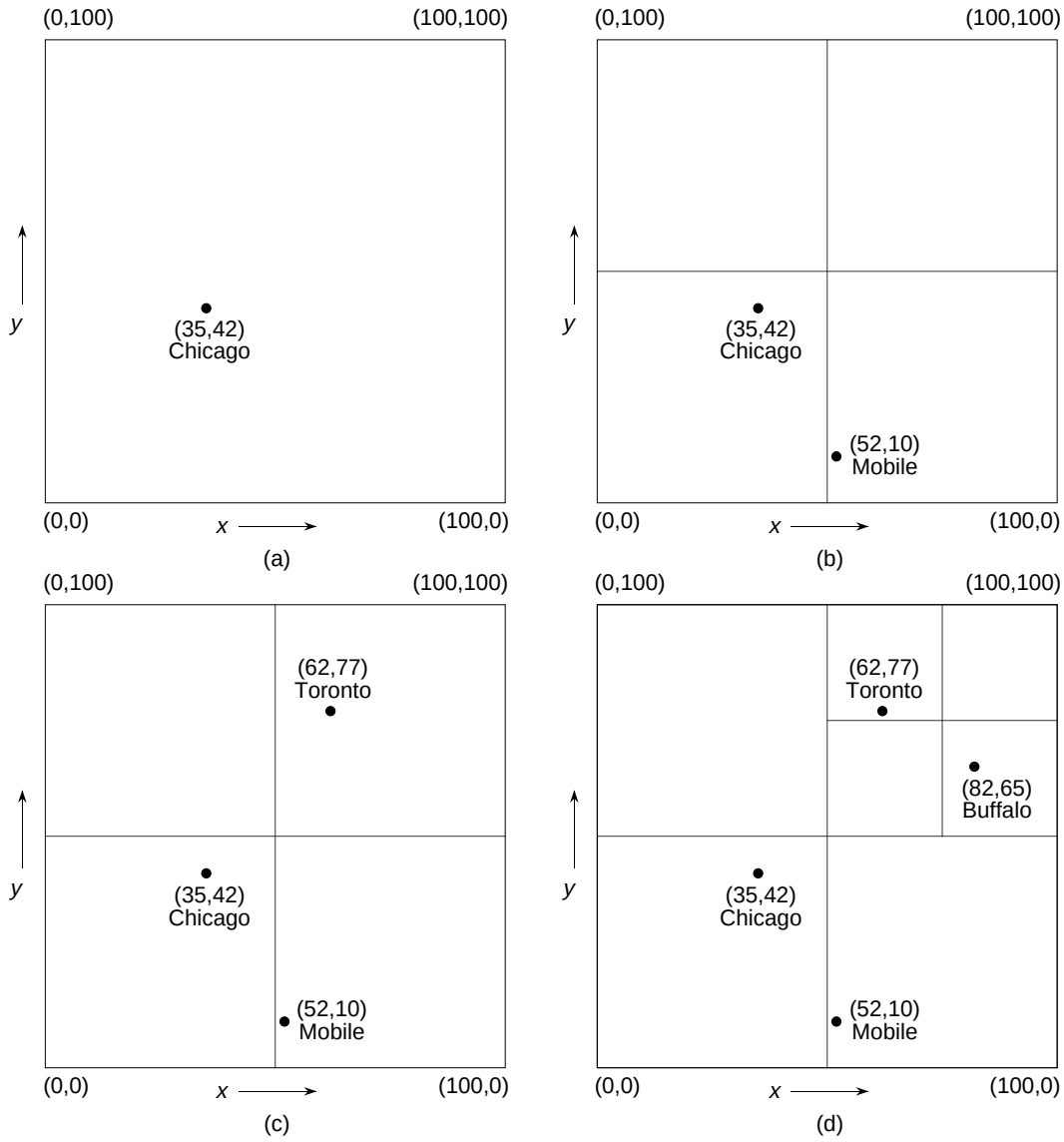


Figure 29: Sequence of partial block decompositions showing how a PR quadtree is built when adding (a) Chicago, (b) Mobile, (c) Toronto, and (d) Buffalo corresponding to the data of Figure 1.

a record in a PR quadtree.

At this point, it is appropriate to elaborate further on the collapsing process. Collapsing can only take place when the deleted node has exactly one brother that is a non-`NIL` leaf node, and has no brothers that are nonleaf nodes. When this condition is satisfied, we can perform the collapsing process repeatedly until encountering the nearest common ancestor that has more than one son. As collapsing occurs, the affected nonleaf nodes are returned to the free storage list.

As an example of collapsing, suppose that we delete `Mobile` and `Atlanta` in sequence from Figure 28. The subsequent deletion of `Atlanta` results in `Miami` meeting the conditions for collapsing to take place. `Miami`'s nearest ancestor with more than one son is `A`. The result of the collapsing of `Miami` is that `Miami` becomes the `SE` son of `A`. Moreover, nonleaf nodes `D` and `F` become superfluous and are returned to the free storage list. The execution time of the deletion process is bounded by two times the depth of the quadtree.

This upper bound is achieved when the nearest ancestor for the collapsing process is the root node.

Analyzing the cost of insertion and deletion of nodes in a PR quadtree depends on the data points already in the tree. In particular, this cost is proportional to the maximum depth of the tree. For example, given a square region of side length s, such that the minimum Euclidean distance separating two points is d, the maximum depth of the quadtree can be as high as $\lceil \log_2((s/d) \cdot \sqrt{2}) \rceil$. Assuming that the data points are drawn from a grid of size $2^n \times 2^n$, for N data points the storage requirements can be as high as $O(N \cdot n)$. This analysis is somewhat misleading as N and n do not necessarily grow in an independent manner. Often, the case is that the storage requirements are proportional to N (see Exercises 5 and 6).

Performing a range search in a PR quadtree is done in basically the same way as in the MX and point quadtrees. The worst-case cost of searching for all points that lie in a rectangle whose sides are parallel to the quadrant lines is $O(F + 2^n)$, where F is the number of points found and n is the maximum depth (see Exercise 8).

The PR quadtree is used in a number of applications. Anderson [6] makes use of a PR quadtree (termed a *uniform* quadtree) to store endpoints of line segments to be drawn by a plotter. The goal is to reduce pen plotting time by choosing the line segment to be output next whose endpoint is closest to the current pen position. Rosenfeld, Samet, Shaffer, and Webber make use of the PR quadtree in a geographic information system [117] as a means of representing point data in a consistent manner with region data that is represented using region quadtrees. In this way, queries can be made on different data sets which are partitioned at the same locations, assuming that the space spanned by the data sets is of the same size and relative to the same origin. This enables answering queries such as finding all cities in wheat-growing regions by simply overlaying the crop map (represented by a region quadtree) and the city map (represented by a PR quadtree, assuming that the cities are treated as points). The overlay operation is implemented by traversing the two trees in tandem and making descents only if encountering two nonleaf nodes in the same position in the tree (for more details, see [123, Chapter 6]).

When the data is not uniformly distributed (e.g., when it is clustered), then the PR quadtree may contain many empty nodes, and, may become unbalanced. The imbalance may be overcome by aggregating the blocks corresponding to the nodes into larger blocks termed buckets. This is achieved by varying the decomposition rule so that a block is split only if it contains more than b points where b is termed the bucket capacity [91]). We use the term *bucket PR quadtree* to describe this representation. It should be clear that increasing the number of points permitted in a block reduces the dependence of the maximum depth of the PR quadtree on the minimum Euclidean distance separation of two distinct points to that of two sets of b points. For example, Figure 30 is the bucket PR quadtree corresponding to the data of Figure 1 when the bucket capacity is 2.

Nelson and Samet [97, 98] analyze the distribution of nodes in a bucket PR quadtree in terms of their occupancies for various values of the bucket capacity[16]. They model a quadtree as a collection of populations in which each population represents the set of nodes in the quadtree that have a particular occupancy. Thus, the set of all empty nodes constitutes one population, the set of all nodes containing a single point constitutes another population, etc. Note that the individual populations contain nodes of different sizes.

As points are added to the bucket PR quadtree, each population grows in a manner that depends on the other populations. For example, when the bucket capacity is c, the probability of producing a new node with occupancy i depends both on the fraction of nodes with occupancy $i - 1$ and on the population of full nodes (occupancy c), since nodes of any occupancy can be produced when a full node splits.

The result of the analysis is the determination of a steady state where the proportions of the various populations are constant under addition of new points according to some data model (e.g., uniform distribution, Gaussian distribution, etc.). This steady state is taken as a representative distribution of populations from which expected values for structure parameters such as average node occupancies are calculated.

The population model is an alternative to the use of a statistical analysis to define a 'typical' structure. The statistical analyses are based on the computation, relative to some model of data distribution, of statistical sums over the space of possible data structure configurations. Statistical analyses for uniformly distributed data have

[16] The rest of the discussion in this section is somewhat technical and may be skipped.

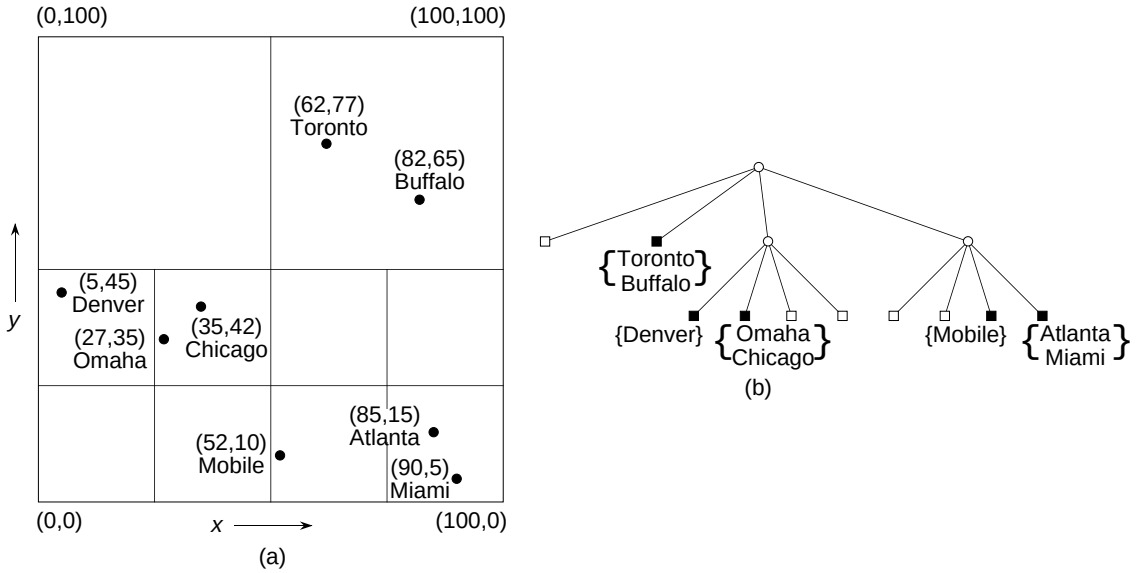


Figure 30: A bucket PR quadtree and the records it represents corresponding to the data of Figure 1 for bucket size 2: (a) the resulting partition of space, and (b) the tree representation.

been applied by Fagin et al. to extendible hashing [33], Flajolet and Puech [38] to tries, Regnier to the grid file (see Section 7.2.1) [113], and Tamminen to EXCELL (see Section 7.2.2) [133].

The advantage of the population model is that dependencies between various populations and the steady state can be determined with relative ease. Moreover, the method is sufficiently flexible that it can be applied to other data structures in which the decomposition is determined adaptively by the local concentration of the data.

Nelson and Samet [97, 98] computed the average bucket occupancies for a number of different bucket capacities when points were generated under a uniform distribution. They found that the theoretical occupancy predictions were consistently slightly higher than the experimental values, and that the size of the discrepancy had a cyclical structure. They use the terms *aging* and *phasing*, respectively, to describe these phenomena. They are explained briefly below.

The population model of bucket splitting assumes that the probability of a point falling into a node of type n_i (i.e., occupancy i) is proportional to the fraction of the total type n_i nodes. This means that the average size of a node is independent of its occupancy. In actuality, we have the situation that nodes with a larger area are formed before the nodes with a smaller area and, hence, the origin of the term *aging*. These larger nodes will tend to have a slightly higher average occupancy than the smaller nodes. Therefore, to maintain a steady state situation, the fraction of high occupancy nodes must be less than predicted by the model, with the result that we have more lower occupancy nodes, thereby yielding a smaller average bucket occupancy.

When nodes are drawn from a uniform distribution, the nodes in the quadtree will tend to split and fill in phase with a logarithmic period repeating every time the number of points increases by a factor of four. In particular, all nodes of the same size tend to fill up and split at the same time. This situation is termed *phasing*. It explains why the average bucket occupancy for a fixed bucket size oscillates with the number of data points. In contrast, when the distribution of points is nonuniform (e.g., Gaussian), then initially there is oscillatory behavior; however, it damps out as node populations in regions of different densities get out of phase.

Exercises

In the following exercises assume that each point in a PR quadtree is implemented as a record of type *node* containing seven fields. The first four fields contain pointers to the node's four sons corresponding to the four quadrants. If P is a pointer to a node and I is a quadrant, then these fields are referenced as `SON`(P,I). The fifth field, `NODETYPE`, indicates whether the node contains a data point (`BLACK`), is empty (`WHITE`), or is a nonleaf node (`GRAY`). `XCOORD` and `YCOORD` contain the x and y coordinate values, respectively, of the data point. The empty PR quadtree is represented by `NIL`.

1. Give an algorithm `PR_COMPARE` to determine the quadrant of a block centered at (x, y) in which a point p lies.

2. Give an algorithm `PR_INSERT` for inserting a point p in a PR quadtree rooted at node r where r corresponds to a $lx \times ly$ rectangle centered at (x, y). Make use of procedure `PR_COMPARE` from Exercise 1.

3. Give an algorithm `PR_DELETE` for deleting data point p from a PR quadtree rooted at node r where r corresponds to a $lx \times ly$ rectangle centered at (x, y). Make use of procedure `PR_COMPARE` from Exercise 1.

4. Assume that the minimum Euclidean distance separating two points is d. Given a square region of side length s, show that $\lceil \log_2((s/d) \cdot \sqrt{2}) \rceil$ is the maximum depth of the PR quadtree.

5. Suppose that you are given N data points drawn from a grid of size $2^n \times 2^n$. Construct an example PR quadtree that uses $O(N \cdot n)$ storage.

6. Under what situation does the PR quadtree use storage proportional to N?

7. Write an algorithm to perform a range search for a rectangular region in a PR quadtree.

8. Show that the worst-case cost of performing a range search in a PR quadtree is $O(F + 2^n)$, where F is the number of points found and n is the maximum depth. Assume a rectangular query region.

Exercises 9–16 are based on an analysis of PR quadtrees performed by Hunter [62].

9. Assume that you are given point data that is uniformly distributed in the unit square. What is the probability of a node at depth k containing a particular point?

10. Continuing the previous exercise, for a collection of v points, what is the probability that none of them lies in a given cell at depth k?

11. What is the probability that exactly one of the v points lies in a given cell at depth k?

12. What is the probability that a given cell at depth k contains two or more points (i.e., it corresponds to a parent node)?

13. Each of the nodes that contains two or more points in it must be a nonleaf node and hence will have four sons. From the probability of the existence of a nonleaf node at depth k, derive the expected total number of nodes (i.e., leaf plus nonleaf nodes) at depth $k + 1$.

14. From the expected number of nodes at depth $k + 1$, derive the expected number of nodes in the entire PR quadtree, say E. Can you find a closed form solution to it?

15. Calculate the ratio of E/v, the expected number of nodes per point. Does it converge?

16. Extend these results to data that is uniformly distributed in the m-dimensional unit hypercube.

4.3 Comparison of Point and Trie-based Quadtrees

A comparison of point and trie-based quadtrees reduces, in part, to a comparison of their decomposition methods. In this discussion we refer to the MX and PR quadtrees as trie-based quadtrees unless they differ in the context of the discussion. The trie-based quadtrees rely on regular decomposition (i.e., subdivision into four congruent rectangular regions). Data points are only associated with leaf nodes in trie-based quadtrees, whereas for point quadtrees, data points can also be stored at nonleaf nodes. This leads to a considerably simpler node deletion procedure for trie-based quadtrees. In particular, there is no need to be concerned about rearranging the tree as is the case when nonleaf nodes are being deleted from a point quadtree.

A major difference between the two quadtree types is in the size of the rectangular regions associated with each data point. For the trie-based quadtrees, the space spanned by the quadtree is constrained to a maximum width and height. For the point quadtree, the space is rectangular and may, at times, be of infinite width and height. For the MX quadtree, this region must be a square with a particular size associated with it. This size is fixed at the time the MX quadtree is defined and is the minimum permissible separation between two data points in the domain of the MX quadtree (equivalently, it is the maximum number of elements permitted in each row and column of the corresponding matrix). For the PR quadtree, this region is also rectangular, but its size depends on what other data points are currently represented by nodes in the quadtree.

In the case of the MX quadtree, there is a fixed discrete coordinate system associated with the space spanned by the quadtree, whereas no such limitation exists for the PR quadtree. The advantage of such a fixed coordinate system is that there is no need to store coordinate information with a data point's leaf node. The disadvantage is that the discreteness of the domain of the data points limits the granularity of the possible differentiation between data points.

The size and shape of a quadtree are important from the standpoints of efficiency of both storage and search operations. The size and shape of the point quadtree is extremely sensitive to the order in which data points are inserted into it during the process of building it. Assuming that the root of the tree has a depth of 0, this means that for a point quadtree of N records, its maximum depth is $N - 1$ (i.e., one record is stored at each level in the tree) while its minimum depth is $\lceil \log_4 ((3 \cdot N + 1)/4) \rceil$ (i.e., each level in the tree is completely full).

In contrast, the shape and size of the trie-based quadtrees is independent of the insertion order. For the MX quadtree, all nodes corresponding to data points appear at the same depth in the quadtree. The depth of the MX quadtree depends on the size of the space spanned by the quadtree and the maximum number of elements permitted in each row and column of the corresponding matrix. For example, for a $2^n \times 2^n$ matrix, all data points will appear as leaf nodes at a depth of n.

The size and shape of the PR quadtree depend on the data points currently in the quadtree. The minimum depth of a PR quadtree for $N > 1$ data points is $\lceil \log_4 N \rceil$ (i.e., all the data points are at the same depth) while there is no upper bound on the depth in terms of the number of data points. Recall from Section 4.2.2 that for a square region of side length s, such that the minimum Euclidean distance separating two points is d, the maximum depth of the PR quadtree can be as high as $\lceil \log_2((s/d) \cdot \sqrt{2}) \rceil$.

The volume of data also plays an important part in the comparison between the point and trie-based quadtrees. When the volume is very high, the MX quadtree loses some of its advantage since an array representation may be more economical in terms of space, as there is no need for links. While the size of the PR quadtree was seen to be affected by clustering of data points especially when the number of data points is relatively small, this is not a factor in the size of a point quadtree. However, when the volume of data is large and is uniformly distributed, the effect of clustering is lessened and there should not be much difference in storage efficiency between the point and PR quadtrees.

Exercises

1. Use the uniform distribution over [0,1] to construct two-dimensional point quadtrees, MX quadtrees, and PR quadtrees and compare them using the criteria set forth in this section. For example, compare

their total path length, maximum depth, etc.

2. Perform an average case analysis of the cost of a partial match query (a subset of the range query) for a point quadtree and a PR quadtree when the attributes are independent and uniformly distributed.

5 K-d Trees

As the dimensionality (i.e., number of attributes) d of the underlying space increases, each level of decomposition of the quadtree results in many new cells since the fanout value of the tree is high (i.e., 2^d). This is alleviated by making use of variants of a *k-d tree* [9]. Originally, in the term *k-d tree*, *k* denotes the dimensionality of the space that is being represented. However, in many applications, the convention is to let the value of d denote the dimension, and this is the practice we have chosen to follow here. In principle, the k-d tree is a binary tree where the underlying space is partitioned on the basis of the value of just one attribute at each level of the tree instead of on the basis of the values of all d attributes, thereby making d tests at each level, as is the case for the quadtree. In other words, the distinction from the quadtree is that in the k-d tree only one attribute (or key) value is tested when determining the direction in which a branch is to be made.

Restricting the number of attributes that are tested at each level of the tree has a number of advantages over the quadtree. First, we make just one test at each level instead of d tests. Second, each leaf node in the quadtree is rather costly in terms of the amount of space that is required due to a multitude of `NIL` links. In particular, the node size gets rather large for a d-dimensional tree, since at least $d + 2^d$ words (assuming that each pointer is stored in one word) are required for each node. Third, algorithms can be simplified as at each recursive step we only have two options since each node only partitions the underlying space into two parts. Finally, the same data structure (pointer-wise) can be used to represent a node for all values of d, notwithstanding the partitioning axis values which may need to be stored for some variants of the k-d tree.

The disadvantage of replacing a test of d values by a sequence of d tests is, in the case of a k-d tree, that now decomposition in d-dimensional space has become a sequential process as the order in which the various axes (i.e., attributes) are partitioned is important. In contrast, this ordering is of no consequence in the case of a quadtree as all d axes are partitioned simultaneously (i.e., in parallel). This lack of an ordering has an advantage in a parallel environment as the key comparison operation can be performed in parallel for the d key values. Therefore, we can characterize the k-d tree as a superior serial data structure and the quadtree as a superior parallel data structure. For a discussion of the use of quadtrees in a multiprocessor environment, see Linn [86], while Hoel [60] and Bestul [17] discuss their use in a parallel environment.

The idea of replacing a test of d attribute values by a sequence of d tests is applicable to both point quadtrees and trie-based quadtrees, although it is applied more often to point quadtrees. The use of the term *k-d tree* to refer to the data structure is somewhat confusing as this term is used uniformly to refer to the point quadtree adaptation whereas a consistent use of our terminology would result in terming it a *point k-d tree*. In fact, we will use this term whenever we need to differentiate it from the trie-based adaptation. There is no consistent terminology for the trie-based variant although *k-d trie* would probably be the most appropriate. The region quadtree counterpart of the k-d tree is termed a *bintree* [134].

In the rest of this section we discuss a number of different variations of the k-d tree with a particular emphasis on the point k-d tree (i.e., the one that organizes the data to be stored rather than organizing the embedding space from which the data is drawn). Section 5.1 discusses the point k-d tree while Section 5.2 presents a number of variants of the trie-based k-d tree.

5.1 Point K-d Trees

There are many variations of the point k-d tree. Their exact structure depends on the manner in which they deal with the following issues.

1. Is the underlying space partitioned at a position that overlaps a data point or may the position of the partition be chosen at random? Recall that trie-based methods restrict the positions of the partitions points thereby rendering moot the question of whether or not the partition actually takes place at a data point.

2. Is there a choice as to the identity of the partition axis (i.e., the attribute or key being tested)? If we adopt a strict analogy to the quadtree, then we have little flexibility in the choice of the partition axis in the sense that we must cycle through the d different dimensions every d levels in the tree although the relative order in which the different axes are partitioned may differ from level to level and among subtrees.

The most common variant of the k-d tree (and the one we focus on in this section) partitions the underlying space at the data points and cycles through the different axes in a pre-defined and constant order. When the partitions can be applied to the underlying space in an arbitrary order rather than having to cycle through the axes, then we preface the name of the data structure with the qualifier *generalized*. For example, a generalized k-d tree also partitions the underlying space at the data points; however, it need not cycle through the axes. In fact, it need not even partition all of the axes. In our discussion, we assume two-dimensional data, and we test x coordinate values at the root and at even depths (given that the root is at depth 0), and y coordinate values at odd depths.

We adopt the convention that when node P is an x-discriminator, then all nodes having an x coordinate value less than that of P are in the left son of P and all those with x coordinate values greater than or equal to that of P are in the right son of P. A similar convention holds for a node that is a y-discriminator. Figure 31 illustrates the k-d tree corresponding to the same eight nodes as in Figure 1.

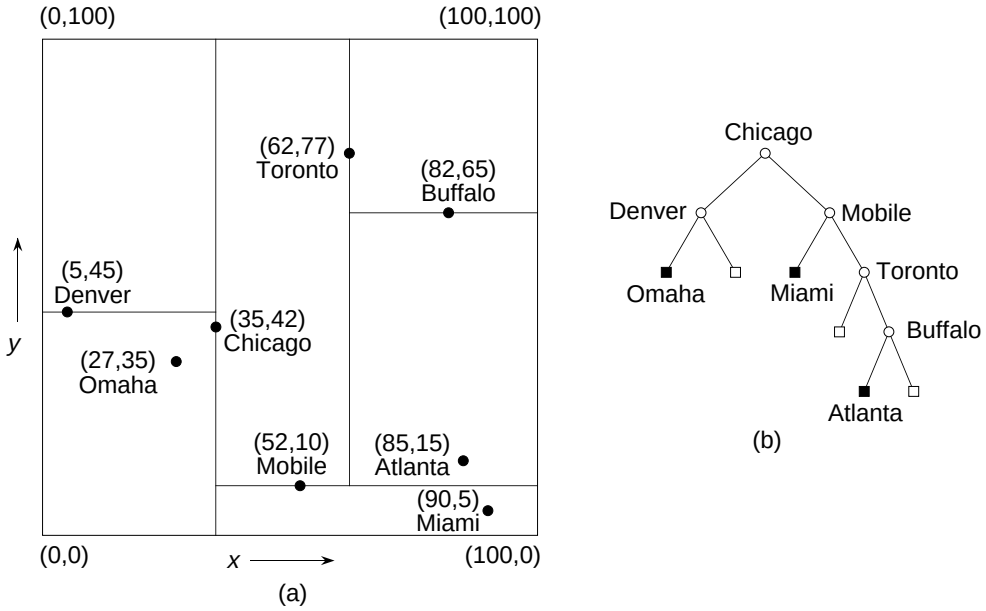


Figure 31: A k-d tree (d=2) and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

In the definition of a discriminator, the problem of equality of particular key values is resolved by stipulating that records which have the same value for a particular key are in the right subtree. As an alternative, Bentley [9] defines a node in terms of a *superkey*. Given a node P, let $K_0(P)$, $K_1(P)$, etc., refer to its d keys. Assuming that P is a j-discriminator, then for any node Q in the left son of P, $K_j(Q) < K_j(P)$, and likewise for any node R in the right son of P, $K_j(R) > K_j(P)$. In the case of equality, a superkey, $S_j(P)$, is defined by forming a cyclical concatenation of all keys starting with $K_j(P)$. In other words, $S_j(P) = K_j(P)$ $K_{j+1}(P) \ldots K_{d-1}(P)$ $K_0(P) \ldots K_{j-1}(P)$. Now, when comparing two nodes P and Q we turn to the left

when $S_j(Q) < S_j(P)$ and to the right when $S_j(Q) > S_j(P)$. If $S_j(Q) = S_j(P)$, then all d keys are equal and a special value is returned to so indicate. The algorithms that we present below do not make use of a superkey.

The rest of this section is organized as follows. Section 5.1.1 shows how to insert a point into a k-d tree. Section 5.1.2 discusses how to delete a point from a k-d tree. Section 5.1.3 explains how to do region searching in a k-d tree. Section 5.1.4 discusses some variants of the k-d tree which provide more flexibility as to the positioning and choice of the partitioning axes.

5.1.1 Insertion

Inserting record r with key values (a, b) into a k-d tree is very simple. The process is the essentially the same as that for a binary search tree. First, if the tree is empty, then allocate a new node containing r and return a tree with r as its only node. Otherwise, search the tree for a node h with a record having key values (a, b). If h exists, then r replaces the record associated with h. Else, we are at a `NIL` node which is a son of type s (i.e., '`LEFT`' or '`RIGHT`', as appropriate) of node c. This is where we make the insertion by allocating a new node t containing r and make t an s son of node c. The only difference from the binary search tree is the fact that we compare x coordinate values at the root and at even depths of the tree, and y coordinate values at odd depths of the tree. Bentley [9] shows that given N points, the average cost of inserting, as well as searching for (i.e., a point query), a node is $O(\log_2 N)$.

As in the case of point quadtrees, the shape of the resulting k-d tree depends on the order in which the nodes are inserted into it. Figure 31 is the k-d tree for the sequence `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`. Figure 32a-d shows how the tree was constructed in an incremental fashion for these cities by giving the appropriate space partitions. In particular, Figure 32a corresponds to the space partition after insertion of `Chicago` and `Mobile`, Figure 32b to the space partition after insertion of `Toronto`, Figure 32c to the space partition after insertion of `Buffalo`, and Figure 32d to the space partition after insertion of `Denver` and `Omaha`.

It should be clear that in the insertion process described here, each node partitions the portion of the plane in which it resides into two parts. Thus, in Figure 31, `Chicago` divides the plane into all nodes whose x coordinate value is less than `35`, and all nodes whose x coordinate value is greater than or equal to `35`. In the same figure, `Denver` divides the set of all nodes whose x coordinate value is less than `35` into those whose y coordinate value is less than `45`, and those whose y coordinate value is greater than or equal to `45`.

As in the case of the point quadtree, the amount of work expended in building a k-d tree is equal to the total path length (TPL) of the tree as it reflects the cost of searching for all of the elements. Bentley [9] shows that the total path length of a k-d tree built by inserting N points in random order into an initially empty tree is $O(N \cdot \log_2 N)$ and, thus, the average cost of inserting a node is $O(\log_2 N)$. The extreme cases are worse since the shape of the k-d tree depends on the order in which the nodes are inserted into it thereby affecting the TPL.

Exercises

Assume that each point in the k-d tree is implemented as a record of type *node* containing three plus d fields. The first two fields contain pointers to the node's two sons corresponding to the directions '`LEFT`' and '`RIGHT`'. If P is a pointer to a node and I is a direction, then these fields are referenced as `SON`(P,I). At times, these two fields are also referred to as `LOSON`(P) and `HISON`(P), corresponding to the left and right sons, respectively. We can determine the side of the tree in which a node, say P, lies relative to its father by use of the function `SONTYPE`(P), which has a value of I if `SON(FATHER`(P),I)$= P$. `COORD` is a one-dimensional array containing the values of the d coordinate of the data point. If P is a pointer to a node and I is a coordinate name, then these fields are referenced as `COORD`(P,I). The `DISC` field indicates the name of the coordinate on whose value the node discriminates (i.e., tests). The `DISC` field is not always necessary as, for example, when the relative order in which the different axes (i.e., coordinates) are partitioned is

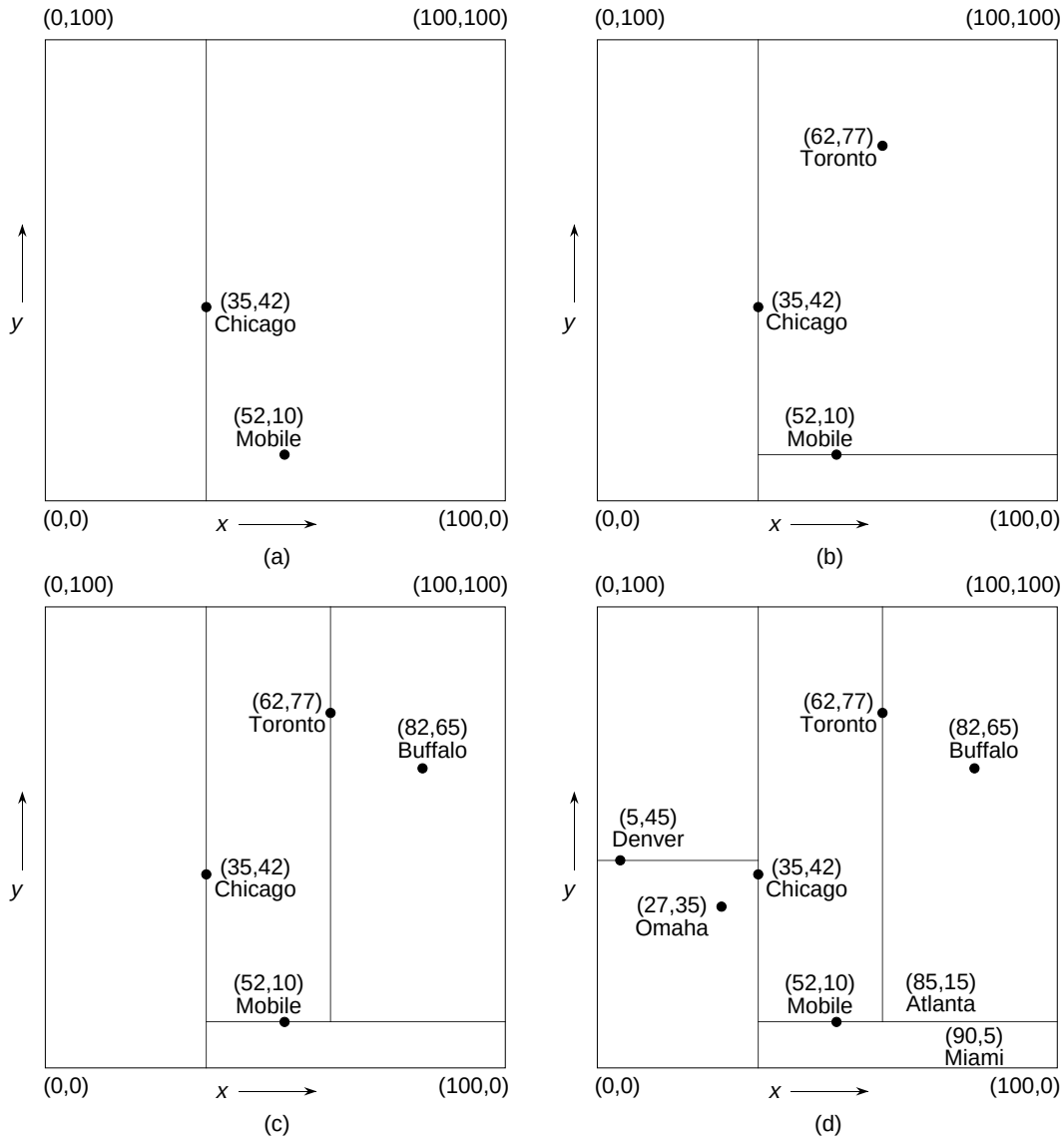


Figure 32: Sequence of partial k-d tree space partitions demonstrating the addition of (a) Chicago and Mobile, (b) Toronto, (c) Buffalo, and (d) Denver and Omaha corresponding to the data of Figure 1.

constant. In this case, it is easy to keep track of the discriminator type of the node being visited as the tree is descended. The empty k-d tree is represented by `NIL`.

1. Give an algorithm `KD_COMPARE` to determine the son of a k-d tree rooted at r in which point p lies.

2. Give an algorithm `KD_INSERT` to insert a point p in a k-d tree rooted at node r. Make use of procedure `KD_COMPARE` from Exercise 1. There is no need to make use of the `SONTYPE` function.

3. Modify procedures `KD_COMPARE` and `KD_INSERT` to handle a k-d tree node implementation that makes use of a superkey in its discriminator field.

4. Prove that the TPL of a k-d tree of N nodes built by inserting the N points in a random order is $O(N \cdot \log_2 N)$.

5. Give the k-d tree analogs of the single and double rotation operators for use in balancing a binary search tree [73, p. 454]. Make sure that you handle equal key values correctly.

5.1.2 Deletion

Deletion of nodes from k-d trees is considerably more complex than for binary search trees. Observe that, unlike the binary search tree, not every subtree of a k-d tree is itself a k-d tree. For example, although, Figure 33a is a k-d tree, its right subtree (see Figure 33b) is not a k-d tree. This is because the root node in a two-dimensional k-d tree discriminates on the value of the x coordinate, while both children of the root node in Figure 33b have x coordinate values that are larger than that of the root node. Thus, we see that special care must be taken when deleting a node from a k-d tree.

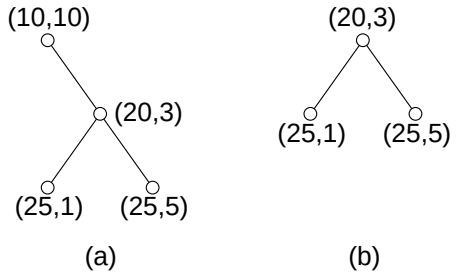


Figure 33: (a) Example of a two-dimensional k-d tree whose (b) right son is not a k-d tree.

In contrast, when deleting a node from a binary search tree, we simply move a node and a subtree. For example, deleting node `3` from Figure 34a results in replacing node `3` with node `4` and replacing the left subtree of `8` by the right subtree of `4` (see Figure 34b). However, this can not be done, in general, in a k-d tree as the nodes with values `5` and `6` might not have the same relative relationship at their new depths.

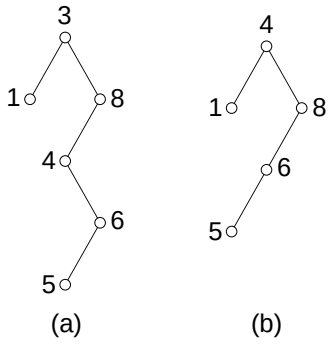


Figure 34: (a) Example of a binary search tree and (b) the result of deleting its root.

Deletion in a k-d tree can be achieved by the following recursive process. We use the k-d tree in Figure 35 to illustrate our discussion, and thus all references to nodes can be visualized by referring to the figure. Let us assume that we wish to delete the node `(a,b)` from the k-d tree. If both subtrees of `(a,b)` are empty, then replace `(a,b)` with the empty tree. Otherwise, find a suitable replacement node in one of the subtrees of `(a,b)`, say `(c,d)`, and recursively delete `(c,d)` from the k-d tree. Once `(c,d)` has been deleted, replace `(a,b)` with `(c,d)`.

At this point, it is appropriate to comment on what constitutes a 'suitable' replacement node. Recall that an x-discriminator is a node that appears at an even depth, and hence partitions its space based on the value of its x coordinate. A y-discriminator is defined analogously for nodes at odd depths. Assume that `(a,b)` is an x-discriminator. We know that every node in `(a,b)`'s right subtree has an x coordinate with value greater

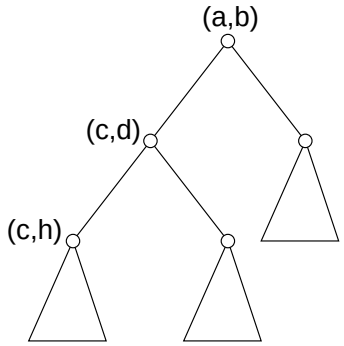


Figure 35: Example of a k-d tree illustrating why the replacement node should be chosen from the right subtree of the tree containing the deleted node.

than or equal to a. The node that will replace (a,b) must bear the same relationship to the subtrees of (a,b). Using the analogy with binary search trees, it would seem that we would have a choice with respect to the replacement node. It must either be the node in the left subtree of (a,b) with the largest x coordinate value, or the node in the right subtree of (a,b) with the smallest x coordinate value.

Actually, we don't really have a choice as the following comments will make clear. If we use the node in the left subtree of (a,b) with the maximum x coordinate value, say (c,d), and if there exists another node in the same subtree with the same x coordinate value, say (c,h), then when (c,d) replaces node (a,b), there will be a node in the left subtree of (c,d) that does not belong there by virtue of having an x coordinate value of c. Thus, we see that given our definition of a k-d tree, the replacement node must be chosen from the right subtree. Otherwise, a duplicate x coordinate value will disrupt the proper interaction between each node and its subtrees. Note that the replacement node need not be a leaf node.

The only remaining question is how to handle the case when the right subtree of (a,b) is empty. We use the k-d tree in Figure 36 to illustrate our discussion, and thus all references to nodes can be visualized by referring to the figure. This is resolved by the following recursive process. Find the node in the left subtree of (a,b) that has the smallest value for its x coordinate, say (c,d). Exchange the left and right subtrees of (a,b), replace the coordinate values of (a,b) with (c,d), and recursively apply the deletion procedure to node (c,d) from its prior position in the tree (i.e., in the previous left subtree of (a,b)).

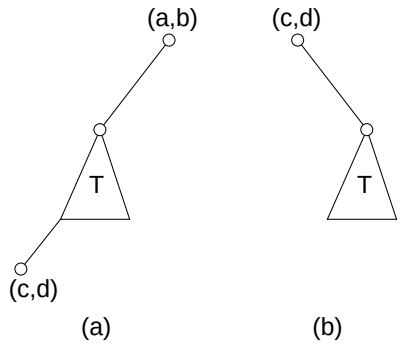


Figure 36: (a) Example k-d tree and (b) the result of deleting (a,b) from it.

With the aid of Figures 35 and 36 we have shown that the problem of deleting a node (a,b) from a k-d tree is reduced to that of finding the node with the least x coordinate value in a subtree of (a,b). Unfortunately, locating the node with the minimum x coordinate value is considerably more complex than the analogous problem for a binary search tree. In particular, although the node with the minimum x coordinate value must be in the left subtree of an x-discriminator, it could be in either subtree of a y-discriminator. Thus, search is involved and care must be taken in coordinating this search so that when the deleted node is an x-discriminator

at depth 0, only one of the two subtrees rooted at each odd depth is searched. This is done using procedure FIND_D_MINIMUM (see Exercise 1).

As can be seen from the discussion, deleting a node from a k-d tree can be costly. We can obtain an upper bound on the cost in the following manner. Clearly, the cost of deleting the root of a subtree is bounded from above by the number of nodes in the subtree. Letting TPL(T) denote the total path length of tree T, it can be shown that the sum of the subtree sizes of a tree is TPL(T)+N (see Exercise 4 in Section 4.1.2).

Bentley [9] proves that the TPL of a k-d tree built by inserting N points in a random order is $O(N \cdot \log_2 N)$, which means that the average cost of deleting a randomly selected node from a randomly built k-d tree has an upper bound of $O(\log_2 N)$. This relatively low value for the upper bound reflects the fact that most of the nodes in the k-d tree are leaf nodes. The cost of deleting root nodes is considerably higher. Clearly, it is bounded by N. Its cost is dominated by the cost of the process of finding a minimum element in a subtree which is $O(N^{1-1/d})$ (see Exercise 6) since on every d^{th} level (starting at the root) only one of the two subtrees needs to be searched.

As an example of the deletion process, consider the k-d tree in parts a and b of Figure 37. We wish to delete the root node A at (20,20). Assume that A is an x-discriminator. The node in the right subtree of A with a minimum value for its x coordinate is C at (25,50). Thus, C replaces A (see Figure 37c), and we recursively apply the deletion procedure to C's position in the tree. Since C's position was a y-discriminator, we seek the node with a minimum y coordinate value in the right subtree of C. However, C's right subtree was empty, which means that we must replace C with the node in its left subtree that has the minimum y coordinate value.

D at (35,25) is the node in the left subtree that satisfies this minimum value condition. It is moved up in the tree (see Figure 37d). Since D was an x-discriminator, we replace it by the node in its right subtree having a minimum x coordinate value. H at (45,35) satisfies this condition, and it is moved up in the tree (see Figure 37e). Again, H is a x-discriminator and we replace it by I — the node in its right subtree with a minimum x coordinate value. Since I is a leaf node, our procedure terminates. Figures 37f and 37g show the result of the deletion process.

In the above discussion, we had to make a special provision to account for our definition of a k-d tree node as a partition of space into two parts: one less than the key value tested by the node and one greater than or equal to the key value tested by the node. Defining a node in terms of a superkey alleviates this problem since we no longer have to always choose the replacing node from the right subtree. Instead, we now have a choice. The best algorithm is one that 'flip flops' between the left and right sons, perhaps through the use of a random number generator (see Exercise 5). Of course, node insertion and search are slightly more complex since it is possible that more than one key will have to be compared at each level in the tree (see Exercise 3 in Section 5.1.1 and Exercise 3 in Section 5.1.3).

Exercises

1. Given a node P in a k-d tree that discriminates on key D, write an algorithm, FIND_D_MINIMUM, to compute the D-minimum node in its left subtree. Repeat for the right subtree of P. Generalize your algorithm to work for either of the two subtrees.

2. Assuming the same k-d tree node implementation as in the Exercises in Section 5.1.1, give an algorithm KD_DELETE for deleting a point p from a k-d tree rooted at node r. The node does not have a FATHER field although you may make use of the SONTYPE function. Make use of procedures KD_COMPARE and FIND_D_MINIMUM from Exercise 1 in Section 5.1.1 and Exercise 1, respectively.

3. Modify procedure KD_DELETE in the solution to Exercise 2 to make use of a variant of FIND_D_MINIMUM that in addition to returning the D-minimum node t, also returns t's SONTYPE values relative to its father f, as well as f.

4. Suppose that a k-d tree node is implemented as a record with a FATHER field containing a pointer to its father. Modify procedure KD_DELETE in the solution to Exercise 2 to take advantage of this additional field.

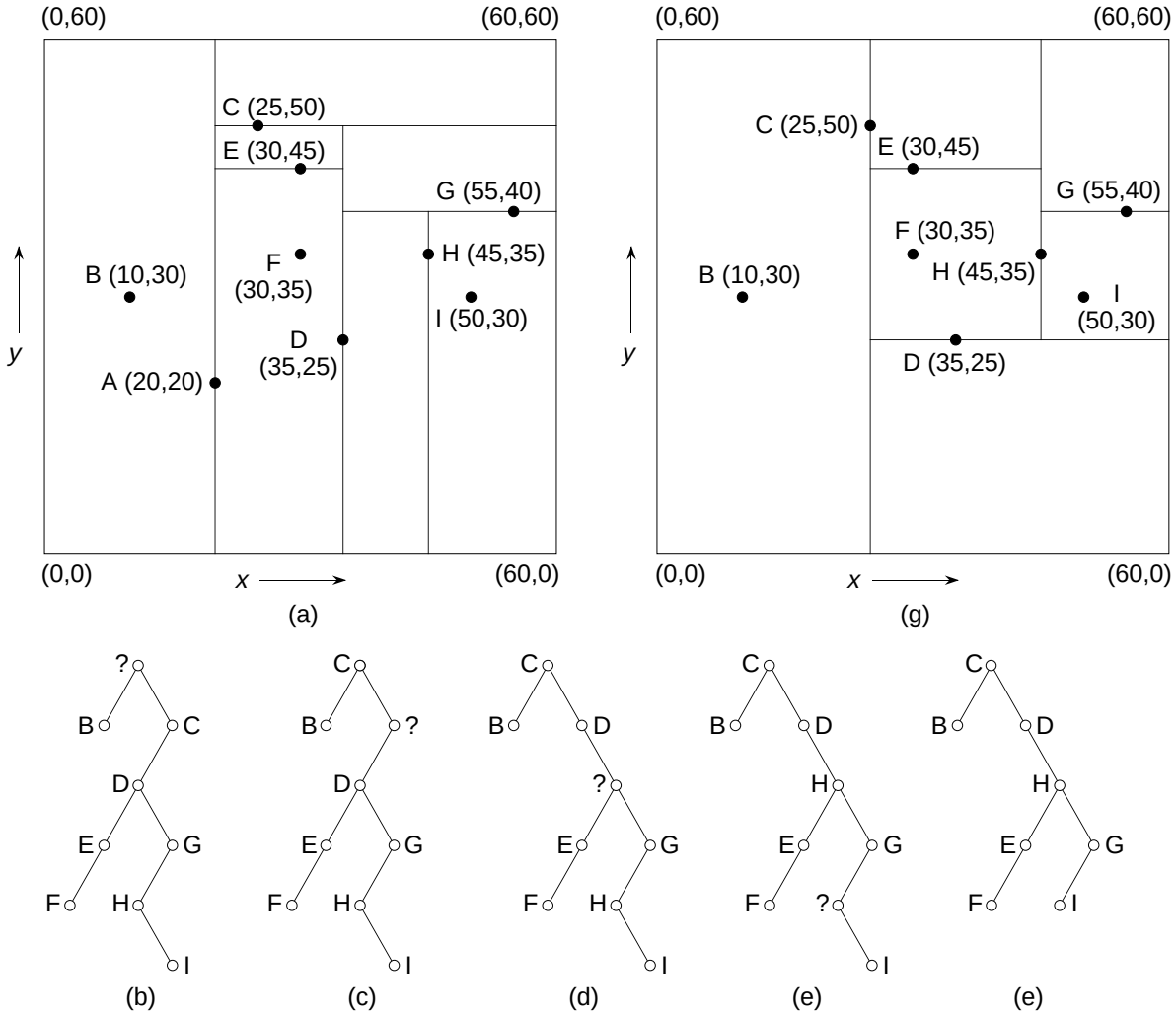


Figure 37: Example illustrating deletion in k-d trees where ? indicates the node being deleted: (a) the original k-d tree, (b-f) successive steps in deleting node A, (g) final k-d tree.

5. Modify procedure `KD_DELETE` in the solution to Exercise 2 to handle a k-d tree node implementation that makes use of a superkey in its discriminator field.

6. Prove that the cost of finding a D-minimum element in a k-d tree is $O(N^{1-1/d})$.

5.1.3 Search

Like quadtrees, k-d trees are useful in applications involving search. Again, we consider a typical query that seeks all nodes within a specified distance of a given point. The k-d tree data structure serves as a pruning device on the amount of search that is required: that is, many nodes need not be examined. To see how the pruning is achieved, suppose we are performing a region search of distance r around a node with coordinate values (a, b). In essence, we want to determine all nodes (x, y) whose Euclidean distance from (a, b) is less than or equal to r — that is, $r^2 \geq (a - x)^2 + (b - y)^2$.

Clearly, this is a circular region. The minimum x and y coordinate values of a node in this circle can not be less than $a - r$ and $b - r$, respectively. Similarly, the maximum x and y coordinate values of a node

in this circle can not be greater than $a + r$ and $b + r$, respectively. Thus, if the search reaches a node with coordinate values (e, f) and KD_COMPARE($(a - r, b - r), (e, f)$)='RIGHT', then there is no need to examine any nodes in the left subtree of (e, f). Similarly, the right subtree of (e, f) need not be searched when KD_COMPARE($(a + r, b + r), (e, f)$)='LEFT'.

For example, suppose that in the hypothetical database of Figure 31 we wish to find all cities within three units of a point with coordinate values (88,6). In such a case, there is no need to search the left subtree of the root (i.e., Chicago with coordinate values (35,42)). Thus, we need only examine the right subtree of the tree rooted at Chicago. Similarly, there is no need to search the right subtree of the tree rooted at Mobile (i.e., coordinate values (52,10)). Continuing our search, we find that only Miami, at coordinate values (90,5), satisfies our request. Thus, we only had to examine three nodes during our search.

Similar techniques are applied when the search region is rectangular making the query meaningful for both locational and nonlocational data. In general, the search cost depends on the type of query. Given N points, Lee and Wong [83] have shown that in the worst case, the cost of a range search of a complete k-d tree is $O(d \cdot N^{1-1/d})$. To see how this bound is obtained we assume that $d = 2$ and, without loss of generality, consider a rectangular search region (see Exercise 1), marked hatched, in Figure 38. Although not explicitly shown in the figure, nodes G, H, R, and V are roots of subtrees which may also be in the shaded region. Similarly, nodes E, I, S, and W are also roots of subtrees which may also contain just one node in which case they are leaf nodes. We use the number of nodes visited while searching the tree as a measure of the amount of work that needs to be expended.

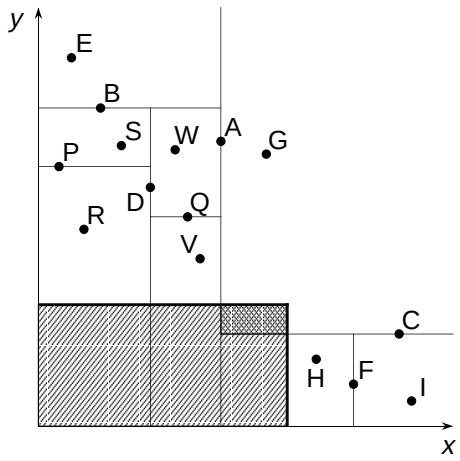


Figure 38: Example space partition for a k-d tree that illustrates the worst case behavior for the k-d tree search procedure.

Since we are interested in the worst case, we assume that B and C (see Figure 38) partition the tree rooted at A in such a way that the search region is overlapped by the regions rooted at B and C. If B is within the search region, then the subtree rooted at D need not be searched further. If B is outside of the search region (as is the case here), then E and its subtrees (if any) need not be searched further. Similarly, if F is within the search region, then H and its subtrees (if any) need not be searched further. Finally, when F is outside of the search region (as is the case here), then I and its subtrees (if any) need not be searched further. G is chosen in such a way that both of its subtrees will have to be searched. Further partitioning of G is analogous to a recursive invocation of this analysis two levels deeper in the tree than the starting level (see the cross-hatched region in Figure 38).

The analysis of the remaining subtrees of B and F (i.e., rooted at D and H) are equivalent to each other and enable the elimination of two subtrees from further consideration at every other level. For example, the sons of the subtrees rooted at D are P and Q, with sons rooted at R and S, and V and W, respectively. The subtrees (if any) rooted at S and W need no further processing.

We are now ready to analyze the worst case of the number of nodes that will be visited in performing a

range search in a two-dimensional k-d tree. Let t_i denote the number of nodes visited when dealing with a k-d tree rooted at level i (where the deepest node in the tree is at level 0). Let u_j denote that number of nodes that are visited when dealing with a subtree of the form illustrated by the subtrees rooted at nodes `D` and `H` in Figure 38. This leads to the following recurrence relations:

$$\begin{aligned} t_i &= 1+1+1+t_{i-2}+u_{i-2}+1+u_{i-3} \\ u_j &= 1+1+1+2\cdot u_{j-2} \end{aligned}$$

with initial conditions $t_0 = u_0 = 0$ and $t_1 = u_1 = 1$.

t_i and u_j can best be understood by referring to Figure 39 which is a tree-like representation of the k-d tree of Figure 38. The terms in t_i correspond to nodes or subtrees rooted at `A`, `B`, `C`, `G`, `D`, `F`, and `H` in order, while the terms in u_j correspond to nodes or subtrees rooted at `D`, `P`, `Q`, `R`, and `V` in order. The square nodes in Figure 39 (e.g., `E`, `I`, `S`, and `W`) correspond to leaf nodes or roots of subtrees which have been pruned: they do not have to be searched.

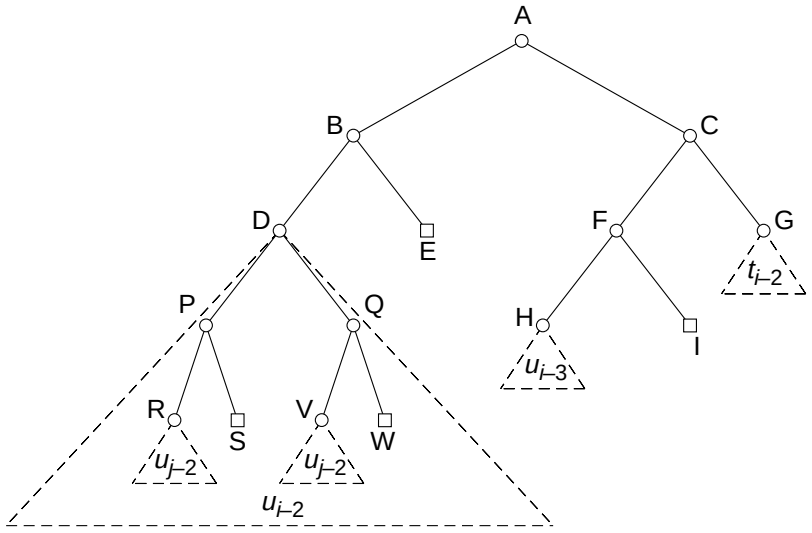


Figure 39: Tree representation of the space partition in Figure 38.

When these relations are solved, we find that, under the assumption of a complete binary tree (i.e., $N = 2^n - 1$), t_n is $O(2 \cdot N^{1/2})$. Note that, unlike the point quadtree, a complete binary k-d tree can always be constructed (see the optimized k-d tree of Bentley [9] discussed in Section 5.1.1). In general, for arbitrary d, t_n is $O(d \cdot N^{1-1/d})$.

Partial range queries can be handled in the same way as range searching. Lee and Wong [84] show that when ranges for s out of d keys are specified, the algorithm has a worst case running time of $O(s \cdot N^{1-1/d})$. For an alternative derivation, see Exercise 6.

As is the case for the point quadtree, the k-d tree can be used to handle all three types of queries specified by Knuth [73]. The range query is described above while simple queries are a byproduct of the k-d tree insertion process. Boolean queries are straightforward. Range queries can be facilitated by use of a bounds array `B`[i] of $2 \cdot d$ elements stored at each node. It contains the range of values for all of the coordinates of the points stored in the k-d tree rooted at the node.

Exercises

1. Why was the search region in Figure 38 chosen in the lower left corner of the space instead of somewhere in the middle?

2. Write a procedure KD_REGION_SEARCH to determine all the nodes in a given k-d tree that intersect a given rectangular region in d-dimensional space. Use a bounds array to facilitate your task. KD_REGION_SEARCH should make use of a pair of auxiliary functions; one to determine if a subtree can be pruned; and the other, to determine if a node is indeed in the search region.

3. Modify procedure KD_REGION_SEARCH of Exercise 2 to handle a k-d tree node implementation that makes use of a superkey in its discriminator field.

4. Solve the recurrence relations for t_i and u_j used to analyze range searching in a k-d tree.

5. A partial match query is closely related to the range query. In this case, values are specified for t ($t < d$) of the keys and a search is made for all records having such values for the specified keys. Show how you would use procedure KD_REGION_SEARCH of Exercise 2 to respond to a partial match query.

6. Given a complete k-d tree of N nodes where $N = 2^{d \cdot h} - 1$ and all leaf nodes appear at depth $d \cdot h - 1$, prove Bentley's [9] result that, in the worst case, $O(N^{(d-t)/d})$ nodes are visited in performing a partial match query with t out of d keys specified.

7. In Exercise 2 in Section 4.1.3, a *perfect point quadtree* was defined and used to analyze the expected number of nodes visited in performing a region query for a 2-dimensional point quadtree. Define a *perfect k-d tree* in an analogous manner and repeat this analysis.

8. Perform an average case analysis for region queries and partially specified queries in a k-d tree.

5.1.4 Point K-d Tree Variants

Our formulation of the k-d tree as well as its ancestral predecessors (i.e., the fixed-grid and the point quadtree) have the property that all partition lines at a given level of subdivision are parallel to the coordinate axes. When more than one axis is partitioned at a given level in the tree, then all partition lines at a given level of subdivision are orthogonal. If there is just one partition at a given level of subdivision (e.g., a k-d tree), then all partition lines at all nodes at this subdivision level are along the same axis and are orthogonal to all partition lines at the immediately preceding and subsequent level of subdivision. Moreover, we have assumed that the partitions cycle through the various axes in a pre-defined and constant order, and that the partition lines must pass through the data points (with the exception of the trie-based methods).

Building k-d trees to satisfy these properties can lead to trees that are unbalanced and hence not very good for search operations. In particular, the total path length (TPL) may be quite high. There are a number of ways of relaxing these rules with the effect that the resulting trees will be more balanced. This will reduce the TPL and thereby make subsequent searching operations faster. There are two approaches to reduce the TPL: static and dynamic. They are discussed below.

The static approach assumes that all the data points are known a priori. Bentley [9] proposes an optimized k-d tree, which is constructed in the same manner as the optimized point quadtree of Section 4.1.1. In this case, the partition lines must pass through the data points and still cycle through the various axes in a fixed and constant order.

An alternative static data structure incorporating 'adaptive partitioning' is the *adaptive k-d tree* of Friedman, Bentley, and Finkel [49]. Unlike the standard k-d tree, and in the spirit of the pseudo quadtree of Overmars and van Leeuwen [111] (see Section 4.1.2), data is only stored at the leaf nodes. In other words, the partition lines need not pass through the data points. Each interior node contains the median of the set (along one key) as the discriminator.

Moreover, the partition lines need not cycle through the various axes. In particular, at each level of subdivision, the discriminator is chosen to be the key for which the spread of the values of the key is a maximum. This spread can be measured by any convenient statistic such as the variance (e.g., the VAMSplit k-d tree [143] which is described in greater detail in Section 7.1), the distance from the minimum to the maximum value (usually normalized with respect to the median value), etc. All records with key values less than the discriminator

are added to the left subtree and all records with key values greater than or equal to the discriminator are added to the right subtree. This process is continued recursively until there are only a few nodes left in a set, at which point they are stored as a linked list. Note that since we no longer require a cyclical discriminator sequence, the same key may serve as the discriminator for a node and its father, as well as its son. Thus the resulting structure could also be characterized as an instance of a *generalized pseudo k-d tree*. We shall use this term in the rest of our discussion whenever we are not focussing directly on a tree construction process that leaves open the method for choosing the discriminator at each level of subdivision. We shall make use of this term in Section 7.1.

Figure 40 shows the adaptive k-d tree corresponding to the data of Figure 1. Note that although Figure 40b leads to the impression that the adaptive k-d tree is balanced, this is not necessarily the case. For example, when several nodes have the same value for one of their keys, then a middle value is impossible to obtain. The concept of a superkey is of no use in such a case.

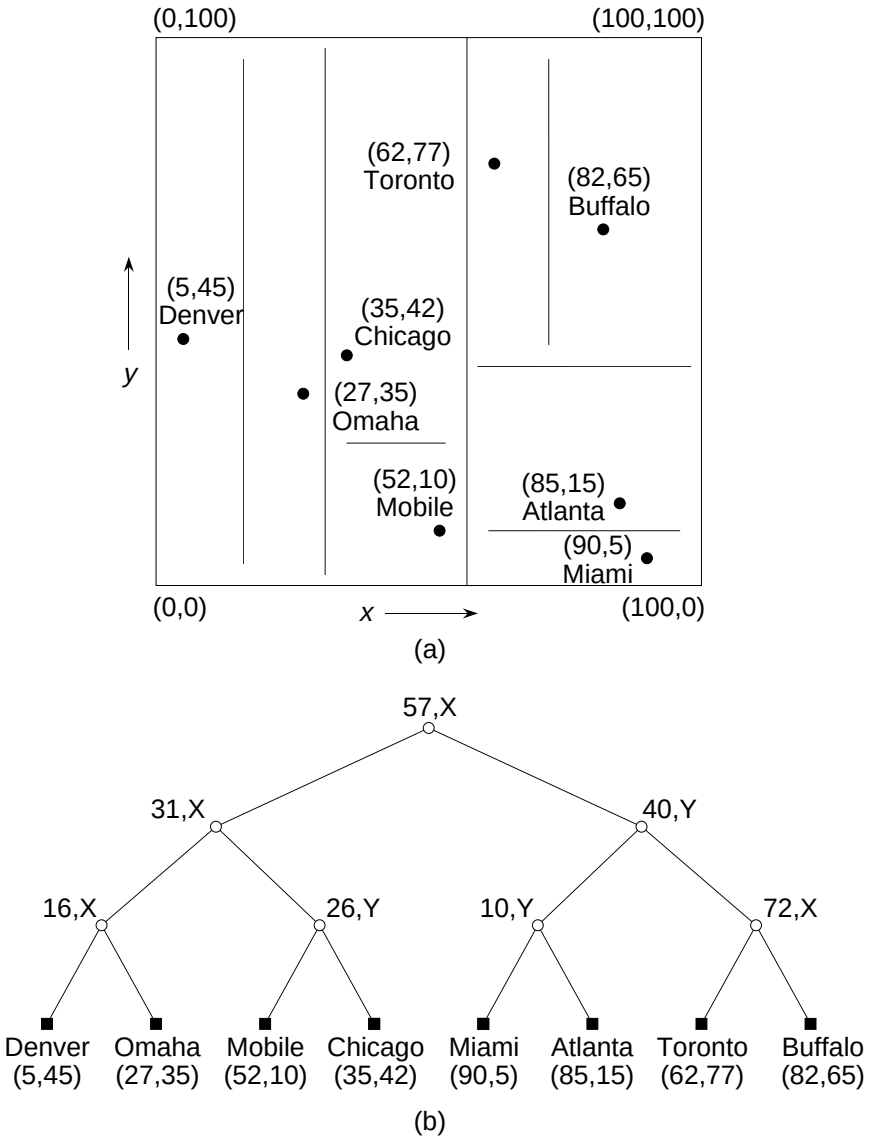


Figure 40: An adaptive k-d tree (d=2) corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation: (a) the resulting partition of space, and (b) the tree representation.

When the data volume becomes large, we may not want as fine a partition of the underlying space as is provided by the adaptive k-d tree and the other variants of the generalized pseudo k-d tree. In particular, it may be desirable for points to be grouped by proximity so that they can be processed together. The generalized pseudo k-d tree can be adapted to facilitate this as well as to identify clusters of points by grouping points by spatial proximity into buckets, and using the generalized pseudo k-d tree decomposition rule (e.g., the one used for the adaptive k-d tree) to partition a bucket whenever it overflows its capacity. We term the result a *bucket generalized pseudo k-d tree*. We have already seen the use of bucketing in our definition of the bucket PR quadtree (see Section 4.2.2). However, in contrast to the bucket PR quadtree, in the bucket generalized pseudo k-d tree, the choice of both the key across which to split as well as as the positions of the partition lines depends on the data. For example, consider Figure 41, the bucket generalized pseudo k-d tree corresponding to the data of Figure 1 when the bucket capacity is 2. The splitting rule is one that uses as a discriminator the key whose values have the maximum range (i.e., the one used for the adaptive k-d tree and hence the result can be described as an instance of the *bucket adaptive k-d tree*).

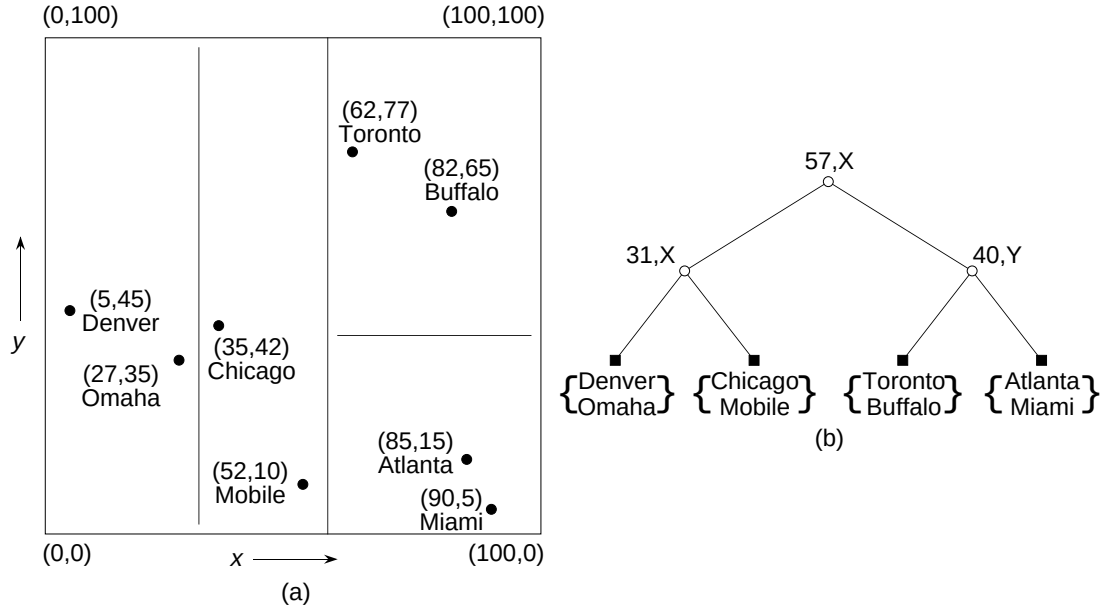


Figure 41: A bucket generalized pseudo k-d tree and the records it represents corresponding to the data of Figure 1 for bucket size 2: (a) the resulting partition of space, and (b) the tree representation.

Matsuyama, Hao, and Nagao [91] use the bucket generalized pseudo k-d tree in a geographic information system (see also the spatial k-d tree of Ooi, McDonell, and Sacks-Davis [102], which is discussed in Section **??** of Chapter **??**). O'Rourke [106, 107] also uses the bucket adaptive k-d tree, calling it a *Dynamically Quantized Space (DQS)*, for cluster detection as well as for multidimensional histogramming to aid in focusing the Hough Transform (e.g., [27]). The *DQP* (denoting *Dynamically Quantized Pyramid*) [107, 128] is closely related to the DQS in the sense that it is also used in the same application. Both the DQP and the DQS are instances of the bucket pseudo quadtree and the bucket generalized pseudo k-d tree, respectively, with the difference that in the DQP, the buckets are obtained by partitioning across all of the keys at each step while still varying the positions of the partition lines (see Section **??** of Chapter **??** for more details).

Below, we briefly describe the Hough Transform in order to show an application of these bucketing methods. This discussion is very specialize and thus may be skipped. Techniques such as the Hough Transform facilitate the detection of arbitrary sparse patterns, such as curves, by mapping them into a space where they give rise to clusters. A DQS (i.e., a bucket generalized pseudo k-d tree) has k attributes or dimensions corresponding to a set of parameters that can be mapped by an appropriate function to yield image patterns. The buckets correspond to the parameter regions. Associated with each bucket is a count indicating the number of times that an element in its region appears in the data. The goal is to find evidence for the presence of clusters

in parameter space.

The DQS is particularly useful in parameter spaces of high dimension (i.e., $\gg 2$). However, for the sake of this discussion, we just look at two-dimensional data. In particular, consider a large collection of short edges and try to determine if many of them are colinear. We do this by examining the values of their slopes and y-intercepts (recall that the equation of a line is $y = m \cdot x + b$). It turns out that a high density of points (i.e., counts per unit volume) in the (m, b) plane is evidence that many colinear edges exist. We want to avoid wasting the space while finding the cluster (most of the space will be empty). To do this, we vary the sizes of the buckets (i.e., their parameter regions) in an attempt to keep the counts equal.

In two dimensions, the Hough Transform for detecting lines is quite messy unless each detected point has an associated slope. The same would be true in three dimensions where we would want not only a position reading (x, y, z) but also the direction cosines of a surface normal (α, β, γ). We can then map each detection into the parameters of a plane, e.g., (ρ, α, β), where ρ is the perpendicular distance from the plane to the origin. If there are many coplanar detectors, this should yield a cluster in the vicinity of some particular (ρ, α, β).

As the buckets overflow, new buckets (and corresponding parameter regions) are created by a splitting process. This splitting process is guided by the two independent goals of an equal count in each bucket and the maintenance of a uniform distribution in each bucket. This is aided by keeping a count and an imbalance vector with each bucket. When the need arises, buckets are split across the dimension of greatest imbalance. There is also a merging rule, which is applied when counts of adjacent neighbors (possibly more than two) are not too large and the merge will not produce any highly unbalanced region. This is especially useful if the data is dynamic.

The DQP addresses the same problem as the DQS with the aid of a complete bucket pseudo quadtree which is known as a pyramid [137] (see Section **??** of Chapter **??** for more details). Thus for k attributes or dimensions it is a full balanced tree where each nonleaf node has 2^k sons. In this case, the number of buckets (i.e., parameter regions), and the relationship between fathers and sons are fixed. The DQP differs from the conventional pyramid in that the partition points (termed *cross-hairs*) at the various levels are variable rather than being fixed.

The partition points of a DQP are initialized to the midpoints of the different attributes. They are adjusted as data is entered. This adjustment process occurs at all levels and is termed a *warping process*. One possible technique when inserting a new data point, say P, in a space rooted at Q, is to take a weighted average of the position of P, say α, and of Q, say $(1-\alpha)$. This changes the boundaries of all nodes in the subtree rooted at Q. P is recursively added to the appropriate bucket (associated with a leaf node), which causes other boundaries to change.

Figure 42 is an example of a two-dimensional DQP for three levels. It should be clear that regions grow smaller near inserted points and that the shape of the DQP depends on the insertion history, thereby providing an automatic focusing mechanism. The warping process is analogous to the splitting operation used in conjunction with a DQS. The advantage of a DQP over the DQS is that it is easier to implement, and merging is considerably simpler. On the other hand, the DQP allocates equal resources for each dimension of the parameter space, whereas the DQS can ignore irrelevant dimensions, thereby yielding more precision in its focusing. Also, the DQS takes up less storage than the DQP.

Now, let us return to our more general discussion. Both the optimized and generalized pseudo k-d trees are static data structures which means that we must know all of the data points a priori before we can build the tree. Thus, deletion of nodes is considerably more complex than for conventional k-d trees (see Section 5.1.2) since we must obtain new partitions for the remaining data. Searching in optimized and generalized pseudo k-d trees proceeds in an analogous manner to that in conventional k-d trees.

The dynamic approach to reducing the TPL in a k-d tree constructs the tree as the data points are inserted into it. The algorithm is similar to that used to build the conventional k-d tree, except that every time that the tree fails to meet a predefined balance criterion, the tree is partially rebalanced using the techniques developed for constructing the optimized k-d tree. This approach is discussed by Overmars and van Leeuwen [111] (see also Willard [145]) who present two variations: the first is analogous to the optimized point quadtree and the

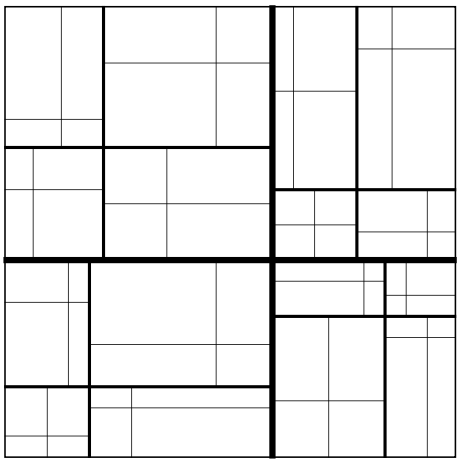


Figure 42: Example two-dimensional DQP for three levels.

second to the pseudo quadtree. One of the differences between these two approaches is that in the dynamic optimized k-d tree, the partition lines must pass through the data points while this need not be the case for the dynamic pseudo k-d tree. This does not have an effect on searching. However, deletion will be considerably simpler in the dynamic pseudo k-d tree than in the dynamic optimized k-d tree.

Methods such as the dynamic optimized k-d tree and the dynamic pseudo k-d tree are characterized by Vaishnavi [141] as yielding 'severely' balanced trees in that they are generalizations of complete binary trees for one-dimensional data. As such, they can not be updated efficiently and this has led Vaishnavi to propose a relaxation of the balancing criterion by generalizing the height balancing constraint for a height-balanced tree (also known as an AVL tree [3, 5]). This is done by storing data in a nested sequence of binary trees so that each nonleaf node P tests a particular key J. Each nonleaf node P has three sons. The left and right sons point to nodes with smaller and larger, respectively, values for the J^{th} key. The third son is the root of a $(k-1)$-dimensional tree containing all data nodes that have the same value for the J^{th} key as the data point corresponding to P. The 'rotation' and 'double rotation' restructuring operations of height-balanced trees are adapted to the new structure. For N points, their use is shown to yield $O(\log_2 N + k)$ bounds on their search and update times (i.e., node insertion and deletion). Unfortunately, this data structure does not appear well-suited for range queries.

The most drastic relaxation of the rule for the formation of the k-d tree is achieved by removing the requirements that the partition lines be orthogonal and parallel to the coordinate axes, and that they pass through the data points. This means that the partition lines are arbitrary, and thus it is no longer meaningful to speak about the order in which the different coordinate axes are partitioned. Also, the ancestral relationship (i.e., the common bond) between the k-d tree and the fixed-grid is now meaningless as an array access structure to the results of the space partition is no longer possible.

The BSP tree of Fuchs, Kedem, and Naylor [50] is an example of a k-d tree where the subdivision lines are not necessarily parallel to the coordinate axes, orthogonal, or pass through the data points. Each subdivision line is really a hyperplane which is a line in two dimensions and a plane in three dimensions. Thus in two dimensions each node's block is a convex polygon, while in three dimensions it is a convex polyhedron.

The BSP tree is a binary tree where each son corresponds to a region. In order to be able to assign regions to the left and right subtrees, we need to associate a direction with each subdivision line. In particular, the subdivision lines are treated as separators between two halfspaces[17]. Let the subdivision line have the equation $a \cdot x + b \cdot y + c = 0$. We say that the right subtree is the 'positive' side and contains all subdivision lines formed

[17] A (linear) *halfspace* in d-dimensional space is defined by the inequality $\sum_{i=0}^{d} a_i \cdot x_i \geq 0$ on the $d+1$ homogeneous coordinates ($x_0 = 1$). The halfspace is represented by a column vector a. In vector notation, the inequality is written as $a \cdot x \geq 0$. In the case of equality, it defines a hyperplane with a as its normal. It is important to note that halfspaces are volume elements; they are not boundary elements.

by separators that satisfy $a \cdot x + b \cdot y + c \geq 0$. Similarly, we say that the left subtree is 'negative' and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c < 0$. As an example, consider Figure 43a which is one of many possible adaptations of the BSP tree to store the point data in Figure 1. Notice the use of arrows to indicate the direction of the positive halfspaces. Although in this example, the tree is balanced, this need not be the case. Similarly, the subdivision lines may pass through the data points themselves.

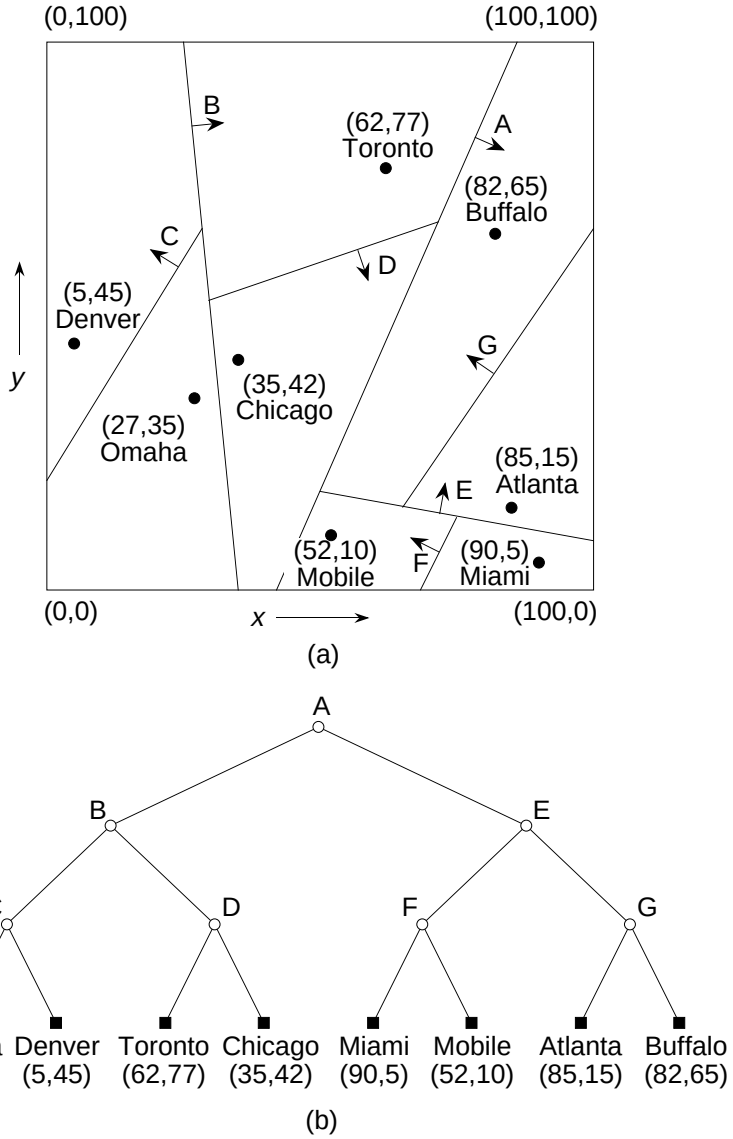


Figure 43: An adaptation of the BSP tree to points and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

Exercises

1. Write a procedure, `BUILD_OPTIMIZED_KDTREE`, to construct an optimized k-d tree.
2. In the text we indicated that the optimized k-d tree can also be constructed dynamically. Given δ ($0 < \delta < 1$), show that there exists an algorithm to build such a tree so that its depth is always at most $\log_{2-\delta} n + O(1)$ and that the average insertion time in an initially empty data structure is $O(\frac{1}{\delta \log_2^2 N})$.

In this case, n denotes the number of nodes currently in the tree and N is the total number of nodes in the final tree.

3. Write a procedure, `BUILD_ADAPTIVE_KDTREE`, to construct an adaptive k-d tree.

4. Analyze the running time of procedure `BUILD_ADAPTIVE_KDTREE`.

5. Define a pseudo k-d tree in an analogous manner to that used to define the pseudo quadtree in Section 4.1.2. Repeat Exercise 15 in Section 4.1.2 for the pseudo k-d tree. In other words, prove that the same result holds for N insertions and deletions in a d-dimensional pseudo k-d tree.

6. How would you use the Hough Transform to detect if some range data (i.e., x, y, z readings) are coplanar.

7. Assume that you are given a BSP tree for a set of points S that yields a partition of space into m regions r_i with t_i ($1 \leq i \leq m$) points in each region r_i. Can you prove that a space partition could also be obtained with orthogonal subdivision lines that are parallel to the axis so that we again have m regions u_j where each u_j ($1 \leq j \leq m$) contains the same points as one of the regions r_i? If this is true, then you have shown that the BSP tree does not yield a more general partition of space than the most general form of the adaptive k-d tree in the sense that the space is partitioned into orthogonal blocks with faces that are parallel to the coordinate axes.

8. Does the result of Exercise 7 also hold for BSP trees for non point objects (i.e., objects with extents such as line segments as discussed in Chapter **??**)?

5.2 Trie-based K-d Trees

The trie-based k-d tree is a binary tree adaptation of the PR quadtree and is obtained in the same way as the point k-d tree is obtained from the point quadtree. However, unlike the point k-d tree, in the trie-based k-d tree, the positions at which decomposition takes place do not have to coincide with data points. Trie-based k-d trees are closely related to the bintree representation of region data (see Section **??** of Chapter **??**). In this case, we recursively decompose the underlying space from which the points are drawn into halves. The exact identity of the key or dimension which we partition depends on the partitioning scheme that is used.

There are many partitioning options, although the most common, and the one we focus on in this section, applies the halving process by cycling through the different axes in a pre-defined and constant order. Again, as in the point k-d tree, in our discussion, we assume two-dimensional data and we test x coordinate values at the root and at even depths (given that the root is at depth 0), and y coordinate values at odd depths. Such a tree is described by Orenstein [103] who calls it a *k-d trie*. Using the terminology of Section 4.2 such a data structure would be called a *PR k-d tree* or a *PR bintree*. For the purpose of consistency with Section 4.2, we shall use the term PR k-d tree. For example, Figure 44 shows the PR k-d tree corresponding to the data of Figure 1.

The advantage of the cyclic partitioning is that there is no need to store the identity nor value of the discriminating key. Of course, other partitioning options are also possible although not as many as in the case of the point k-d tree since the positions through which the partition lines must pass are restricted. For example, we can vary the order in which we partition the various axes and employ rules similar to those used to construct the generalized pseudo k-d tree (e.g., the adaptive k-d tree) except that they can only be used to choose the partition axis rather than both the partition axis and the partition position. When there are absolutely no rules as to the order in which the axes corresponding to the keys are partitioned, then we term the result a *generalized k-d trie*. The generalized k-d trie represents point data in a manner analogous to that used for region data by the *AHC* (*Adaptive Hierarchical Coding*) [21] or *generalized bintree* described in Section **??** of Chapter **??**.

One of the shortcomings of some trie-based representations such as the PR k-d tree (as well as the PR quadtree) is that when the data is not uniformly distributed (e.g., when the data is clustered)[18], the tree contains

[18] See [136] for an analysis of this situation.

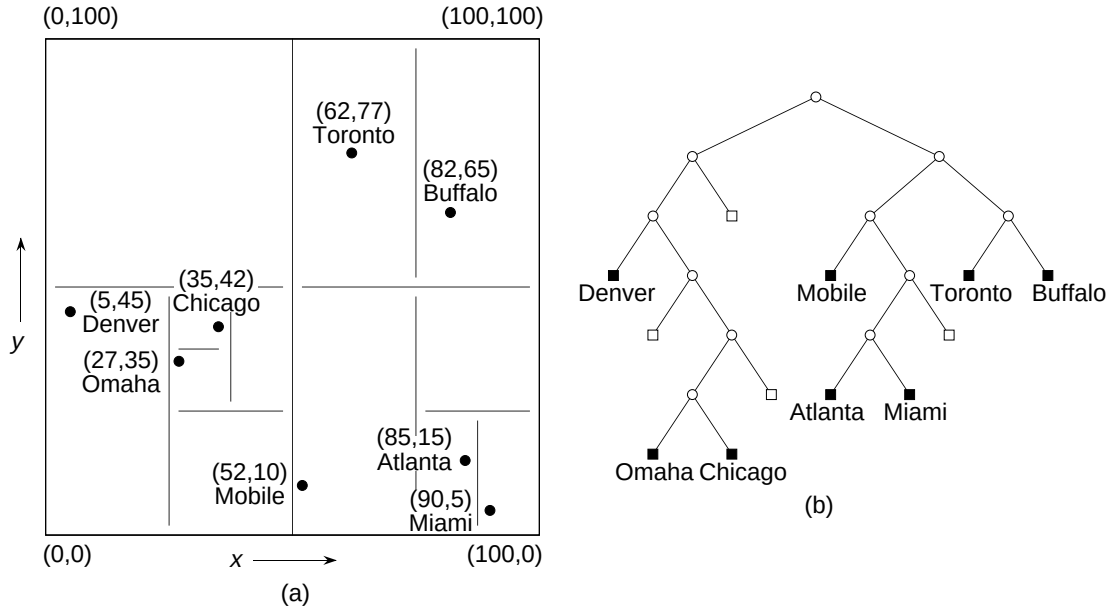


Figure 44: A PR k-d tree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

many empty nodes and, thereby, becomes unbalanced. For example, inserting `Amherst` at location `(83,64)` in the PR k-d tree (PR quadtree) given in Figure 44 (28) will result in much decomposition to separate it from `Buffalo` which is at `(82,65)`. This effect is shown explicitly by the amount of decomposition necessary to separate `Omaha` and `Chicago` in the PR k-d tree given in Figure 44.

There are a number of ways of overcoming this shortcoming. Both of the approaches that we describe are characterized as making use of bucketing, albeit in a different manner than in Section 7. The first approach treats each block in the PR k-d tree as a bucket of capacity b and stipulates that the block is split whenever it contains more than b points. Having buckets of capacity b ($b > 1$) reduces the dependence of the maximum depth of the PR k-d tree on the minimum Euclidean distance separation of two distinct points to that of two sets of at most b points apiece. This is a good solution as long as the cluster contains b or less points, as otherwise we still need to make many partitions. The result is termed a *bucket PR k-d tree* (also known as a *hybrid k-d trie* [103]) and is shown in Figure 45 for the data in Figure 1 when the bucket capacity is 2. The bucket PR k-d tree assumes a cyclic partitioning. If we permit the order in which we partition the various axes rather than cycling through them in a particular order, then the result is termed a *bucket generalized k-d trie*. It is referred to extensively in Section 7.1.

The PMR k-d tree [96, 97, 98] for points[19] addresses the clustering problem by making use of a concept related to bucket capacity which we term a *splitting threshold*. Given a splitting threshold b, we say that if a block c contains more than b points, then it is split *once*, and only once. This is so even if one of the resulting blocks f still has more than b points. If during the insertion of another point p we find that p belongs in f, then f is split, once and only once. The idea is to avoid many splits when more than b points are clustered and there are no other points in the neighborhood. The PMR k-d tree makes use of a cyclic partitioning.

On the other hand, a point is deleted from a PMR k-d tree by removing it from the node corresponding to the block that contains it. During this process, the occupancy of the node and its siblings is checked to see if the deletion causes the total number of points in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the node and its siblings, then they are merged and the merging process is reapplied to the resulting node and its siblings. Notice the asymmetry between the splitting and

[19] This structure was originally defined as a PMR quadtree but its adaptation for k-d trees is straightforward: we just vary the decomposition from 2^d blocks at each level to just 2 blocks at each level.

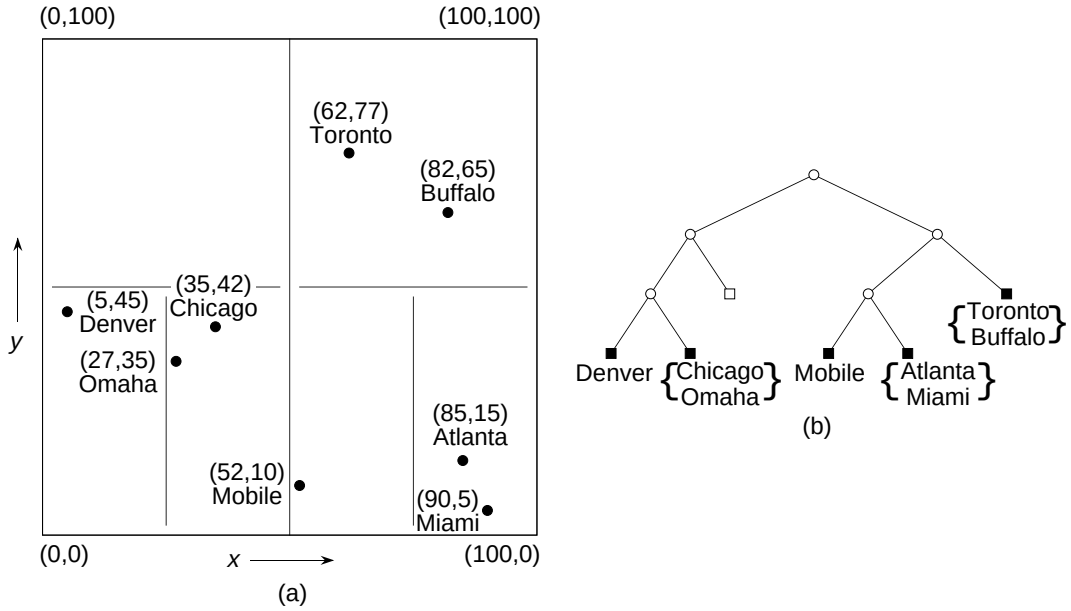


Figure 45: A bucket PR k-d tree and the records it represents corresponding to the data of Figure 1 for bucket size 2: (a) the resulting partition of space, and (b) the tree representation.

merging rules. Also, observe that, unlike the bucket PR k-d tree, the shape of the PMR k-d tree does depend on the order in which the points were inserted.

The second approach applies test compression techniques similar to those devised by Ohsawa and Sakauchi [100, 101] in their development of the BD tree which is closely related to the BANG file [42] and to a lesser extent the hB-tree [88, 89, 118] (see Section sec-tree-directory for more details about these two structures). In essence, to form a BD tree, paths in the PR k-d tree (also applicable to the PR quadtree) that consist of nonleaf nodes are compressed so that all nonleaf nodes that have just one non-`NIL` son are eliminated. In particular, the nodes are replaced by one nonleaf node that indicates the results of the comparisons performed in accessing the first node in the path that has more than one non-`NIL` son. This is similar to the notion of Patrician tries in digital searching [73, p. 490].

As an example of the use of test compression techniques, let us examine their application to the PR k-d tree given in Figure 44. The BD tree as described by Ohsawa and Sakauchi is a bucket method in the sense that the nodes in the tree correspond to buckets and are split once they overflow. In our example, we assume a bucket capacity of 1. Each nonleaf node in the BD tree contains a variable length bit string (termed a *discriminator zone expression (DZE)* by Ohsawa and Sakauchi), say s, consisting of `0`s and `1`s that indicates the results of the tests that lead to the left son of the node. The right son is accessed when the test result does not match the DZE by complementing the last binary digit of s. We say that 0 and 1 correspond to $<$ and '$\geq$', respectively.

Figure 46a-h shows how a BD tree is constructed in an incremental fashion for `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`, in order. All nonleaf nodes are labeled with their DZEs, assuming a sequence of alternating $x, y, x, y, \ldots$ tests starting at the left end of the DZE. For example, in Figure 46b, the root has a DZE of 0, which means that the left son corresponds to $x < 50$ (i.e., the left half of the space spanned by the quadtree). As a more complicated example, the left son of the root in Figure 46e has a DZE of 000, which means that the left son of the left son of the root corresponds to $x < 25$ and $y < 50$.

At this point, let us make a few observations about the BD tree. First, the BD tree is not unique. Rotation operations can be applied to reformulate the tests in the nonleaf nodes so that a greater degree of balancing is achieved. Such transformations are easy to express in terms of the DZEs [101]. Second, when the bucket capacity is greater than 1, say B, then we can decide to continue splitting until no region contains more than

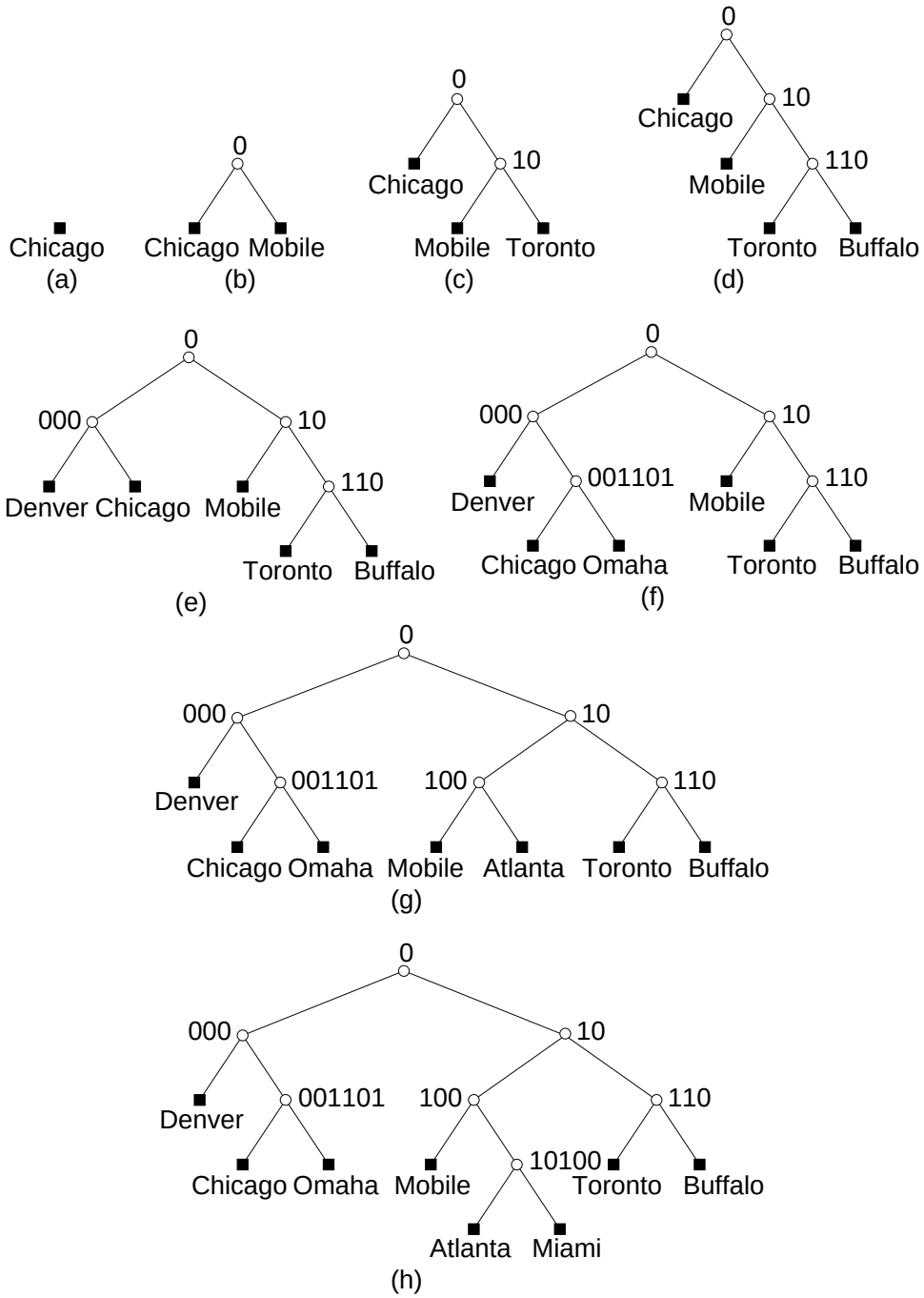


Figure 46: Sequence of partial BD trees demonstrating the addition of (a) Chicago, (b) Mobile, (c) Toronto, (d) Buffalo, (e) Denver, (f) Omaha, (g) Atlanta, and (h) Miami corresponding to the data of Figure 1.

$x \cdot B$ $(0 < x < 1)$ points. Ohsawa and Sakauchi suggest choosing $x = 2/3$ [100, 101]. Third, unlike the PR quadtree and PR k-d tree, a BD tree does not result in the partitioning of space into a collection of hyper-rectangles.

It should be clear that the BD tree can be used for data of arbitrary dimensionality. For more details on how to implement basic operations such as insertion and deletion, as well as answering exact match, partial match,

and range queries for BD trees, see Dandamudi and Sorenson [24]. For an empirical performance comparison of the BD tree with some variations of the k-d tree, see Dandamudi and Sorenson [23].

Exercises

1. Formulate k-d tree variant of a matrix representation analogous to the MX quadtree (discussed in Section 4.2.1) and term it an *MX k-d tree* or an *MX bintree*. Give procedures `MX_KD_INSERT` and `MX_KD_DELETE` to insert and delete data points in an MX bintree.

2. Why would an MX k-d tree be a better matrix representation than the MX quadtree?

3. Write a pair of procedures `PR_KD_INSERT` and `PR_KD_DELETE` to insert and delete, respectively, data points in a PR k-d tree.

4. Assuming that key values are uniformly distributed real numbers in [0,1) represented in binary, prove that the average depth of a PR k-d tree is $O(\log_2 N)$. Can you extend this result to PR quadtrees?

5. Let the size of a data item be bounded by h (i.e., the maximum total number of bits in the k keys) and let the bucket capacity be c. Suppose that a bucket in a bucket PR k-d tree overflows. Assuming that each bit has an equal probability of being 0 or 1, find the expected number of bits that must be tested to resolve the overflow.

6. Write a pair of procedures `PMR_KD_INSERT` and `PMR_KD_DELETE` to insert and delete, respectively, data points in a PMR k-d tree.

7. In general, the BD tree does not result in the partitioning of space into a collection of hyper-rectangles. Under what conditions do some of the subtrees correspond to hyper-rectangles?

8. Is it necessary to store the entire DZE with each node in the BD tree?

9. Write a procedure to insert a point into a two-dimensional BD tree.

10. Write a procedure to delete a point from a two-dimensional BD tree.

11. Write a procedure to perform a point search in a two-dimensional BD tree.

12. Write a procedure to perform a range search for a rectangular region in a two-dimensional BD tree.

13. Implement a BD tree with a bucket capacity of c. In particular, write procedures to insert and delete nodes from it. The key issue is how to handle bucket overflow and underflow. Try to use techniques analogous to those used for B-trees — such as, rotation of elements between adjacent buckets that are not completely full.

6 One-dimensional Orderings

When point data is not uniformly distributed over the space from which it is drawn, then using the fixed-grid representation means that some of the grid cells have too much data associated with them. The result is that the grid must be refined by further partitions. Unfortunately, this leads to the creation of many empty grid cells thereby wasting storage. In Sections 4 and 5 we described how to eliminate some of the empty cells by aggregating spatially adjacent empty grid cells to form larger empty grid cells. However, not all spatially adjacent empty grid cells can be aggregated in this manner as the identity of the ones that can be aggregated is a function of the specific tree-based space decomposition process that was applied to generate them. In particular, recall that both the quadtree and the k-d tree are the result of a recursive decomposition process, and thus only empty grid cells that have the same parent can be aggregated.

In this section, we describe how to eliminate all empty grid cells by ordering all the grid cells (regardless of whether or not they are empty), and then imposing a tree access structure such as a balanced binary search tree, B-tree, etc. on the elements of the ordering that correspond to nonempty grid cells. The ordering is really a mapping from the d-dimensional space from which the data is drawn to one dimension. The mapping that is chosen should be invertible. This means that given the position of a grid cell in the ordering, it should be easy to determine its location. When the grid cells are not equal-sized but still resulting from a fixed grid, then we must also record the size of the nonempty grid cells.

There are many possible orderings with different properties (see Figure **??** and the accompanying discussion in Section **??** of Chapter **??**). The result of drawing a curve through the various grid cells in the order in which they appear in the ordering is called a *space-filling curve*. In this section, we elaborate further on two orderings: bit interleaving and bit concatenation. When all the attributes are either locational or numeric with identical ranges, then the orderings are applicable to the key values as well as the locations of the grid cells. On the other hand, when some of the attributes are nonlocational, then the orderings are only applicable to the locations of the grid cells. In the latter case, we still need to make use of an additional mapping from the actual ranges of the nonlocational attribute values to the locations of the grid cells. We do not discuss this issue further here.

Bit interleaving consists of taking the bit representations of the values of the keys comprising a record and forming a *code* consisting of alternating bits from each key value. For example, for $d = 2$, the code corresponding to data point $A = (X, Y) = (x_m x_{m-1} \ldots x_0, y_m y_{m-1} \cdots y_0)$ is $y_m x_m y_{m-1} x_m - 1 \ldots y_0 x_0$, where we arbitrarily deem key y to be the most significant. Figure 47 is an example of the bit interleaving mapping when $d = 2$. We have also labeled the codes corresponding to the cities of the example database of Figure 1, applying the mapping f defined by $f(z) = z \div 12.5$ to the values of both x and y coordinates. Recall that the same mapping was used to obtain the MX quadtree of Figure 26. The drawback of using bit interleaving is the fact that it is not performed efficiently on general computers. Its complexity depends on the total number of bits in the keys. Thus, in d dimensions, when the maximum depth is n, the work required is proportional to $d \cdot n$.

Bit concatenation consists of concatenating the key values (e.g., for data point A the concatenated code would be $y_m y_{m-1} \ldots y_0 x_m x_{m-1} \ldots x_0$). Bit concatenation is the same as row order or column order (depending on whether the y coordinate value or x coordinate value, respectively, is deemed the most significant). It is a classical ordering for storing images and is also known as *raster scan* or *scan* order.

Above, we have shown how to apply the orderings to the key values. Applying the orderings to the locations of the grid cells is easy. We simply identify one location in each grid cell c which serves as the representative of c. This location is in the same relative position in all of the grid cells (e.g., in two dimensions, this could be the location in the lower-left corner of each grid cell). Once the location has been identified, we simply apply the ordering to its corresponding key values. Again, if the grid cells can vary in size, then we must also record their size.

It should be clear that in light of our interest in range searching, bit interleaving is superior to the bit concatenation method since the latter results in long narrow search ranges, whereas the former results in more square-like search ranges. This argument leads to the conclusion that bit concatenation is analogous to the inverted list method while bit interleaving is analogous to the fixed-grid method. In fact, bit concatenation results in the records being sorted according to a primary key, secondary key, etc.

Range searching using a tree-based representation of the result of applying a one-dimensional ordering is fairly straightforward. Below we show how it is performed for a binary search tree by an algorithm proposed by Tropf and Herzog [139]. We use Figure 48 which is the binary search tree corresponding to the key values of Figure 1 encoded using bit interleaving as in Figure 47. The range is specified by the minimum and maximum codes in the search area. For example, to find all the cities within the rectangular area defined by `(25,25)`, `(50,25)`, `(50,63)`, and `(25,63)`, which is represented by codes `12`, `24`, `50`, and `38` (see Figure 49), respectively, we must examine the codes between `12` and `50`. The simplest algorithm is recursive and traverses the binary search tree starting at its root. If the root lies between the minimum and maximum codes in the range, then both subtrees must be examined (e.g., code `21` in Figure 48). Otherwise, one subtree needs

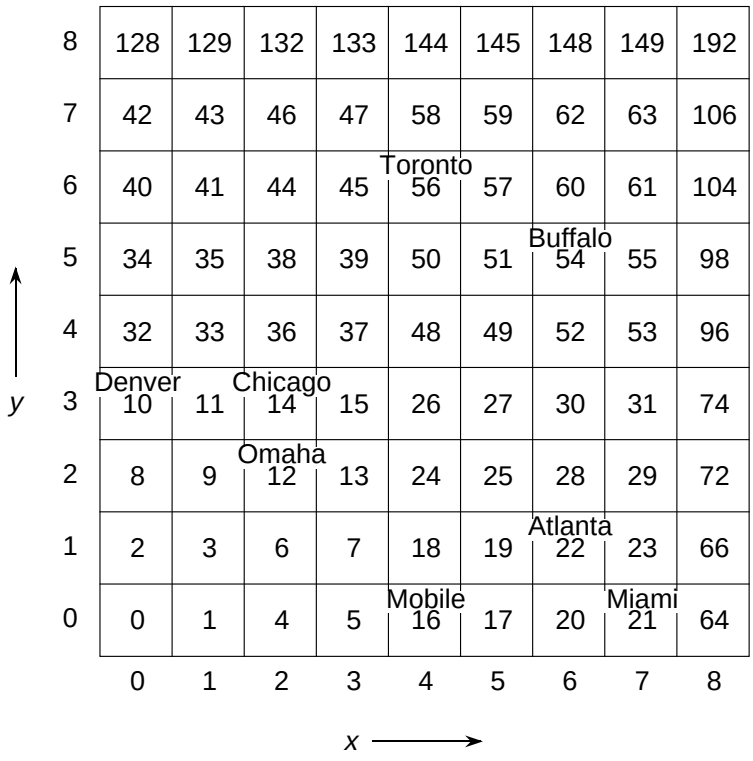


Figure 47: Example of the result of applying the bit interleaving mapping to two keys ranging in values from 0 to 8. The city names correspond to the data of Figure 1 scaled by a factor of 12.5 to fall in this range.

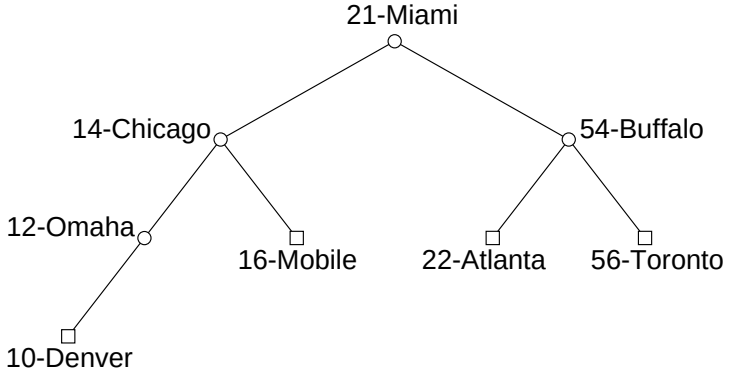


Figure 48: A binary search tree corresponding to the data of Figure 1 encoded using bit interleaving as in Figure 47.

to be searched (e.g., the left subtree of code 54 in Figure 48).

This algorithm is inefficient because many codes lie in the range between the minimum and maximum codes without being within the query rectangle. This is illustrated by the two staircases in Figure 49. Codes above the upper staircase are greater than the maximum search range code, while codes below the lower staircase are less than the minimum search range code. All remaining codes are potential candidates yet most are not in the query rectangle.

To prune the search range when the root lies within the search range (e.g., code 21 in Figure 48), Tropf and

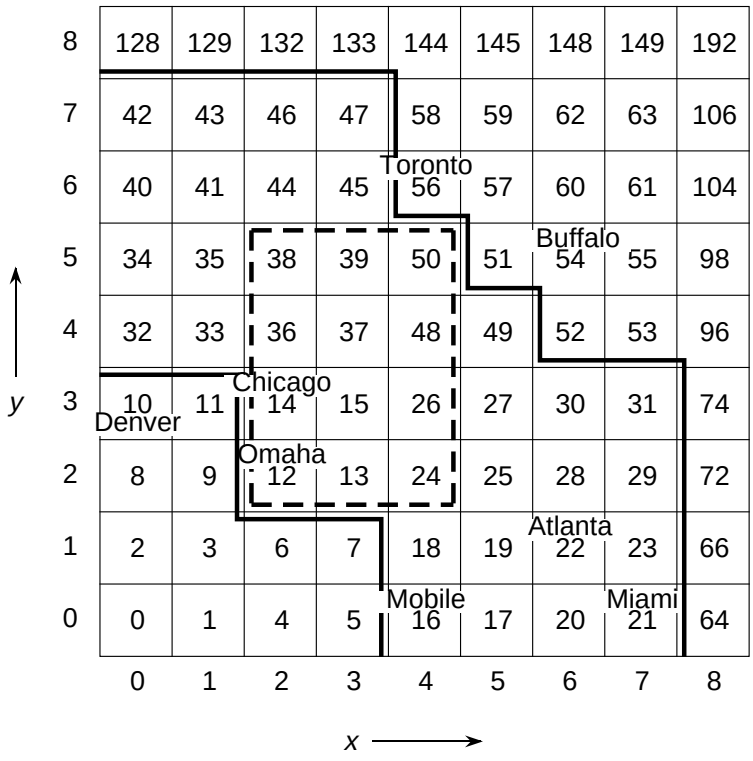


Figure 49: Illustration of the range search process when using bit interleaving.

Herzog define two codes termed `LITMAX` and `BIGMIN` which correspond to the maximum code in the left son and the minimum code in the right son, respectively, that are in the query rectangle. The subsequent search of the left son uses `LITMAX` as the maximum code in the range (e.g., `15` in the left subtree of Figure 48) while the search of the right son uses `BIGMIN` as the minimum code in the range (e.g., `24` in the right subtree of Figure 48) with the result that many codes are eliminated from consideration (e.g., codes between `16` and `23` in Figure 48) after examining the root for the query rectangle between `12` and `50`).

Experiments reported by Tropf and Herzog [139] show that given N records, for small hypercube ranges, the average number of records inspected is $O(d \cdot \log_2 N + F)$, where F is the number of records found. White [144] shows how to perform the same task when using a B-tree as the underlying representation with the same order of execution time.

As pointed out earlier, the ordering methods that we have described can be applied to any of the representations that make use of a recursive decomposition process to generate a set of grid cells of unequal size. In this case, recall that since the mapping is no longer onto, we also need to record some information to enable us to determine the size of the grid cell. A simple, and different, way to obtain an ordering is to traverse the underlying tree access structure and assign successively higher numbers to the grid cells. The problem with such an approach is that there is no simple correlation between the number that is assigned to the grid cell and its actual location in space (i.e., the mapping is not invertible).

An alternative, and preferable solution, is to assign a unique number to each grid cell c in the resulting space decomposition based on the path from the root of the access structure to c, and to record some information about the size of the partition that takes place at each level of the tree. In fact, this approach is equivalent to the solution that we proposed at the start of this section which represents each nonempty block consistently by some easily identifiable location in the block (e.g., the one in the lower-left corner of the block) and then applies the mapping (e.g., bit interleaving) to it. The result of the mapping is stored in the one-dimensional access structure along with the size information. In fact, this is the basis of a linear index into a two-dimensional spatial database developed by Morton [94] and refined later by a number of other researchers including Gar-

gantini [51] and Abel and Smith [2].

As we can see, bit interleaving is quite a powerful an idea. In the rest of this section, we give a brief historical background of its development and mention an interesting application. It is difficult to determine the origin of the notion of bit interleaving. The first mention of it was by Peano [112]. Bentley [9] attributes bit interleaving to McCreight as a way to use B-trees [22] to represent multidimensional data. The resulting B-tree is called an *N-tree* by White [144], while Orenstein and Merrett [105] term it a *zkd Btree*. Note that the result is different from the k-d-B-tree of Robinson [116] (see Section 7.1.1). It was proposed by Tropf and Herzog [139] to give a linear order on multidimensional data and accessing it via a binary search tree or balanced variant thereof. Orenstein and Merrett [105] use the term *Z order* to denote the resulting ordering while reviewing its use in a number of data structures in the context of range searching. They use the Z order to build a variant of a binary search tree which they term a *zkd tree*, while cautioning that it differs from the conventional k-d tree since an inorder traversal of a k-d tree does not necessarily yield the nodes in Z order.

An interesting twist on bit interleaving that improves the efficiency of partial match queries, and potentially also of range queries, is reported by Faloutsos [34]. He suggests that the keys be encoded by their Gray codes (recall the definition in Section **??** of Chapter **??**) prior to the application of bit interleaving. The result is that the transitions between successive points in the ordering are either in the horizontal or vertical directions — not diagonal as is the case for the Z order, which characterizes the normal bit interleaving process. Faloutsos feels that diagonal transitions between successive records are undesirable because these records do not have any key values in common. This is most visible when performing a partial match query (i.e., a partial range query).

As an example of the use of Gray codes, suppose $k = 2$ and the keys are X and Y with ranges between 0 and 3. The query $X \geq 2$ means that we only need to examine one cluster of points (e.g., Figure 50a) when bit interleaving is applied after coding the keys with their Gray codes (but see Exercise 6). In contrast, for the same query, using bit interleaving without a Gray code means that we must examine two clusters of points (e.g., Figure 50b). Using bit concatenation such that key Y is more significant than X, the same query requires us to examine four clusters of points (e.g., Figure 50c).

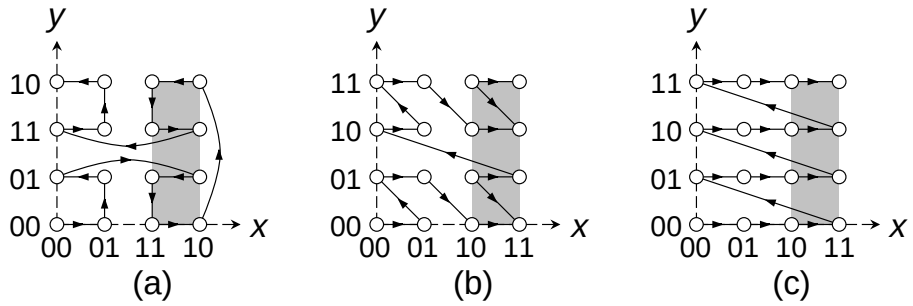


Figure 50: Orderings induced on points from $\{1, 2, 3\}$ and the result of the partial query x≥2 shown shaded. (a) Gray codes prior to bit interleaving. (b) bit interleaving without Gray codes. (c) bit concatenation with y being more significant than x.

Depending on how the data is structured, a multiplicity of clusters may have a significant impact on the cost of retrieving the answer. Faloutsos [35] has shown that for a partial match query, using the Gray codes never increases the number of clusters and can reduce the number of clusters by at most 50%. Faloutsos further conjectures that the use of Gray codes has a similar effect on range queries, although at the present this is an open problem. The overhead cost incurred by using Gray codes is that of conversion from a binary representation, which is linear in the size of the codeword [114].

Exercises

1. Write an algorithm to perform bit interleaving of the x and y coordinate values of a point in an efficient

manner.

2. Write a procedure `BI_SEARCH` to perform a range search for a rectangular region in a two-dimensional database implemented as a binary search tree with bit interleaving as the mapping function.

3. Assume a d-dimensional database implemented as a B-tree with bit interleaving as the mapping function. Write a procedure `BTREE_SEARCH` to perform a range search for a d-dimensional rectangular region.

4. Given a database of N records, prove that for a small rectangular d-dimensional region, the range search in Exercise 3 takes $O(d \cdot \log_2 N + F)$ time where F is the number of records found.

5. Give an algorithm for converting between a binary representation and a Gray code and vice versa.

6. Give an example partial match query involving keys X and Y with ranges between 0 and 3 such that the use of Gray codes prior to interleaving does not result in a decrease in the number of clusters from the number of clusters that result from normal bit interleaving.

7. Prove that for partial match queries, using Gray codes prior to bit interleaving never increases the number of clusters and can reduce them by a maximum of 50%.

8. Derive a method of enumerating all the possible partial match queries for two keys. Also, evaluate the expected number of clusters when using bit concatenation, bit interleaving, and bit interleaving using Gray codes.

9. What is the effect on range queries of using Gray codes prior to bit interleaving?

10. One of the problems with the Gray code encoding is the existence of long horizontal and vertical transitions between successive elements in the order. Find an ordering that avoids this problem (recall Section **??** of Chapter **??**).

7 Bucket Methods

When the data volume becomes very large, much of the data resides in secondary storage (e.g., disk) rather than in main memory. This means that tree access structures such as the quadtree and k-d tree variants discussed in Sections 4 and 5, which are based on making between 1 and d tests at each level of the tree, become impractical due to the limited fanout at each node. The problem is that each time we have to follow a pointer, we may have to make a disk access. This is far costlier than indirect addressing which is the case when the tree resides entirely in main memory. This has led to the development of what are termed *bucket methods*.

There are many different bucket methods and even more ways of distinguishing between them. One way to do so is by noting whether they organize the data to be stored or the embedding space from which the data is drawn. An alternative, and the one we follow in this section, is to subdivide them into two principal classes. The first class consists of aggregating the data objects in the underlying space. The second class consists of decompositions of the underlying space with an appropriate access structure. It is important to note that not all of the methods can be classified so neatly in the sense that some of them can be made to fit into both classes. We do not dwell on this issue further in this chapter.

The first class aggregates the actual data objects into sets (termed *buckets*) usually the size of a disk page in secondary memory so that the buckets are as full as possible. These methods still make use of a tree access structure where each node in the tree is the size of a disk page and all leaf nodes are at the same level. Thus the fanout is much larger than in the quadtree (and, of course, the k-d tree). It should be clear that decomposing the underlying space is not a goal of these representations. Instead, they are more accurately classified as object hierarchies since they try to aggregate as many objects (points in our case) as possible into each node. When the nodes are too full, they are split at which time some of the representations (e.g., the R-tree [55]) attempt to aggregate spatially proximate objects in the nodes resulting from the split. However, there is no guarantee

that the space spanned by the collections of objects in the nodes (e.g., their convex hull, minimum bounding box, minimum bounding circle, etc.) is disjoint.

Nondisjointness is a problem for search algorithms as the path from the root of such structures (e.g., an R-tree) to a node that spans the space containing a given data point or object is not necessarily unique. In other words, the data point or object can be covered by several nodes of the R-tree yet it is only found in one of them. This results in more complex searches. No such problems exist when the decomposition of the space spanned by the collections of objects in the nodes is disjoint. Examples of such methods include the R^+-tree [36, 127, 129] which is really a k-d-B-tree [116] with bounding boxes around the portions of space resulting from the decomposition. When the data objects have extent (e.g., nonpoint objects such as rectangles, lines, etc.), the disjointness requirement of these representations may result in decomposing the individual objects that make up the collection at the lowest level into several pieces. The disjointness requirement is usually expressed in the form of a stipulation that the minimum bounding box of the collection of objects at the lowest level of the tree (i.e., at the leaf nodes) are disjoint instead of that the minimum bounding boxes of the individual objects are disjoint[20]. The price that we pay for disjointness is that there is no longer a guarantee that each node will be as full as possible.

Neither of these object aggregation methods (i.e., nondisjoint and disjoint) is discussed further in this chapter as they are covered in great detail in Sections **??** and **??**, respectively, of Chapter **??** where their usage is discussed for nonpoint objects (e.g., region data). Of course, that discussion is also applicable for point data.

Methods that are based on a disjoint decomposition of space can also be viewed as elements of the second class. The key property that is used to distinguish between the different elements of this class is whether or not the underlying space is decomposed into a grid of cells. When a grid is used, the access structure is usually an array (termed a *grid directory*) When a grid is not used, an access structure in the form of a tree is used (termed a *tree directory*). Such methods are the focus of the rest of this section with tree directory methods being discussed in Section 7.1 and grid directory methods being discussed in Section 7.2.

7.1 Tree Directory Methods

The tree access structures that form the basis of bucket methods that make use of a tree directory differ from the quadtree and k-d tree access structures discussed in Sections 4 and 5, respectively, in terms of the fanout of the nodes. As in the case of an R-tree, the data points are aggregated into sets (termed *point buckets* for the moment) where the sets correspond to subtrees S of the original access structure T (assuming variants of T with bucket size 1 and where all data points are stored in the leaf nodes). They are made to look like the R-tree by also aggregating the internal (i.e., nonleaf) nodes of T into buckets (termed *region buckets* for the moment) thereby also forming a multiway tree (i.e., like a B-tree). The elements of the region buckets are regions. The tree access structures differ from the R-tree in the following respects:

1. the aggregation of the underlying space spanned by the nodes is implicit to the structure, and
2. all nodes at a given level are disjoint and together they span the entire space.

In contrast, for the R-tree, the following must hold:

1. the spatial aggregation must be represented explicitly by storing the minimum bounding boxes that correspond to the space spanned by the underlying nodes that comprise the subtree, and
2. the bounding boxes may overlap (i.e., they are not necessarily disjoint).

A region bucket R, whose contents are internal nodes of T, corresponds to a subtree of T and its fanout value corresponds to the number of leaves in the subtree represented by R. Again, note that the leaves in the subtrees represented by the region bucket are internal nodes of the access structure T.

[20] Requiring the minimum bounding boxes of the individual objects to be disjoint may be impossible to satisfy as is the case, for example, for a collection of line segments all of which meet at a particular point.

The use of a directory in the form of a tree to access the buckets was first proposed by Knott [70]. The nodes in a B-tree can also be used as buckets. A linear ordering on multidimensional point data can be obtained by using bit interleaving, as mentioned in Section 6, and then storing the results in a B-tree (recall the zkd Btree). Unfortunately, in such a case, a B-tree node does not usually correspond to a k-dimensional rectangle. Thus, the representation is not particularly conducive to region searching.

A number of different representations have been developed to overcome these deficiencies. They are the subject of the rest of this section which is organized as follows. Section 7.1.1 describes the k-d-B-tree which is the simplest tree directory method from a conceptual point of view. The drawback of the k-d-B-tree is the possible need to split many nodes when a node overflows. Section 7.1.2 presents the LSD tree which tries to overcome some of the drawbacks of the k-d-B-tree although use of the LSD tree may lead to low storage utilization. Section 7.1.3 contains an overview of the hB-tree which has better storage utilization than the LSD tree. This is achieved by removing the requirement that all blocks are hyper-rectangles. Section 7.1.4 discusses methods where the underlying space from which the data is drawn is partitioned at fixed positions (i.e., based on a trie) rather than in a data-dependent manner as in the k-d-B-tree, LSD tree, and hB-tree. This includes the multilevel grid file, the buddy tree, and the BANG file. Section 7.1.5 explains the BV-tree which is a novel method that has excellent performance guarantees for searching, All of the above techniques are dynamic. Section 7.1.6 describes a couple of static methods based on the adaptive k-d tree.

7.1.1 K-d-B-trees

A simple example of a method that makes use of a tree directory is the k-d-B-tree [116]. It is best understood by pointing out that it is a bucket variant of a k-d tree similar in spirit to the bucket generalized pseudo k-d tree (see Section 5.1.4) in the following sense:

1. the partition lines need not pass through the data points,
2. there is no need to cycle through the keys, and
3. all the data is in the leaf nodes.

The blocks corresponding to the leaf nodes of this variant are buckets of capacity b (termed *point pages*) where a bucket is split in two whenever it contains more than b points. What differentiates the k-d-B-tree from the bucket generalized pseudo k-d tree is that in order to form the k-d-B-tree, the nonleaf nodes comprising subtrees of the bucket generalized pseudo k-d tree are also grouped into buckets of capacity c (termed *region pages*). In this case, c corresponds to the number of pointers in each subtree to point pages or other region pages. In other words, c is the number of leaf nodes in the extended binary tree[21] representation of the k-d tree corresponding to the space partition induced by the region page. Thus c is the maximum number of regions that can make up a region page. Note that c does not have to be the same as b, and, in fact, it is usually smaller as the size of the individual entries in the region pages is usually larger since they represent the boundaries of regions.

As noted above, the nodes of the original bucket generalized pseudo k-d tree that make up each region page form a variant of a k-d tree T corresponding to the space partition induced by the region page. T stores region partitions in its nonleaf nodes and pointers to other region pages or point pages in its leaf nodes. Therefore, T is neither a generalized k-d tree since no actual data is stored in its nonleaf nodes, nor is T a generalized pseudo k-d tree as no actual data is stored in its leaf nodes. Thus we term T a *generalized k$^+$-d tree* in recognition of its similarity in spirit to a B$^+$-tree since their nodes only contain partitions rather than actual data.

It is interesting to observe that obtaining a k-d-B-tree is equivalent to taking a bucket generalized pseudo k-d tree B and replacing it by a multiway tree B' which is a variant of a B$^+$-tree[22] where the nodes of B'

[21] In an extended binary tree, each node is either a leaf node or has two sons. For more details, see Section **??** of Chapter **??**.

[22] We have an analogy with a B$^+$-tree rather than a B-tree because in the k-d-B-tree all the data points are in the leaf nodes, while the nonleaf nodes only contain the boundaries of the partitions.

are aggregates of subtrees of B containing a maximum of c elements. One of the reasons for stating that the resulting structure is a *variant* of the B$^+$-tree is that, unlike B$^+$-trees, we cannot guarantee that each bucket in the k-d-B-tree will be 50% full. However, as in B$^+$-trees, all point pages (i.e., leaf nodes) in a k-d-B-tree are at the same depth.

Of course, the resulting structure of the k-d-B-tree has many of the same properties as the B-tree although updates (i.e., insertion and deletion) cannot always be achieved in analogous ways. In particular, when new data values are inserted into (deleted from) a B-tree, an overflowing (underflowing) bucket is either split (merged with an adjoining bucket) and a middle partition value promoted to (demoted from) the father bucket. Alternatively, we could try to make use of what is termed *deferred splitting* to rotate values from adjacent buckets. Unfortunately, the latter is not possible in the case of multidimensional point data (i.e., $d > 1$) as the buckets are not split on the basis of an ordering — that is, that partitions are made on the basis of the relative positions (i.e., locations) of the points in the underlying space that is spanned by the bucket. Moreover, recall that only the leaf nodes contain actual data.

If insertion of a point into the k-d-B-tree causes a point page p to overflow, then a partition line and axis are chosen and the corresponding bucket is split. This results in the creation of two new point pages s_1 and s_2, and the data in p is inserted into the appropriate new point page based on its position relative to the partition line. Finally, p is deleted from the region page f that contains it, while s_1 and s_2 are added to f.

Adding s_1 and s_2 to f may also cause f to overflow. If this is the case, then we pick a partition line l and axis in the region corresponding to f and split f into region pages f_1 and f_2. For example, consider the region page given in Figure 51a and let its capacity be 9. As we can see, this page has 10 regions and has overflowed. Suppose that we split it at $x = 50$ as shown in Figure 51b. This means that we must check the relative positions of all of the regions corresponding to the elements of f with respect to the partition line l (e.g., $x = 50$ in Figure 51b). In particular, we must determine if they are entirely to the left (e.g., regions `B`, `C`, and `D` in Figure 51b) or entirely to the right (e.g., regions `E`, `F`, `H`, `I`, and `J` in Figure 51b) of l and insert them into f_1 or f_2 as is appropriate. All elements e of f whose regions are intersected by l (e.g., regions `A` and `G` in Figure 51b) must be split and new regions created which are inserted into f_1 or f_2. This additional splitting process is applied recursively to the sons of e and terminates upon reaching the appropriate point pages.

If a region page f has overflowed and f is not a root page of the k-d-B-tree, then f is deleted from its parent page g, the newly-created region pages f_1 and f_2 are inserted into g, and the check for overflow is applied again (this time to g). If f has overflowed and f is a root page of the k-d-B-tree, then a new region page h is created containing region pages f_1 and f_2 as its elements and l as the partition line. Thus we see that the k-d-B-tree grows at the root (i.e., its height increases by one) which is the same way that the B$^+$-tree grows.

Too many deletions may result in very low storage utilization. This can be overcome by either joining two or more adjacent point pages (i.e., catenating them) provided they are joinable (i.e., they have the same father region, their combined region forms a hyper-rectangle, and the total number of points in the new page n does not cause n to overflow), or redistributing the contents of the adjacent point pages which is the case when there is underflow in one of the point pages. If there is a reduction in the number of point pages, then this process is recursive in the sense that the parent region pages should be checked as well. For example, in Figure 51, region pages `H` and `I` are joinable. On the other hand, region page `G` is not joinable with region page `H` nor is `G` joinable with region page `I` as neither of their combined regions forms a hyper-rectangle. Note, however, that `G` is joinable with the combination `H` and `I` as their combined region does form a hyper-rectangle. An alternative solution, which can also be applied in the case of overflow, is to reorganize the region pages by changing the positions (as well as the number) of partition lines. This will mean that descendant pages must be checked as well since this may cause the regions corresponding to the point pages to change thereby necessitating the movement of points between pages which may lead to overflow and creation of new partitions by going up the tree.

The drawback of the k-d-B-tree is that the insertion process is quite complex in the sense that we must be quite careful in our choice of a partition line l and partition axis when a region page f overflows. There are two problems. The first problem is that we must choose the new partition line so that the newly-created region

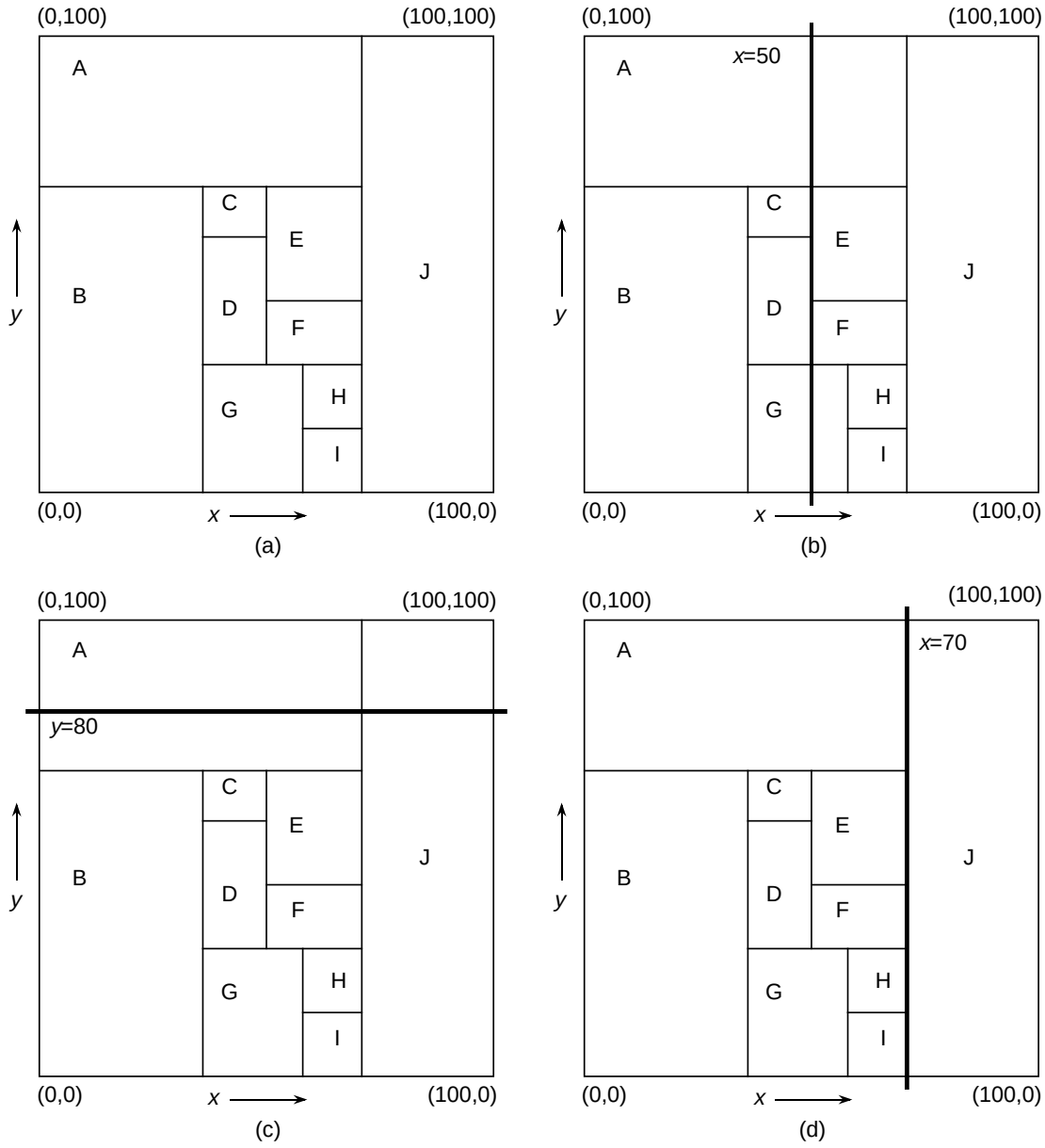


Figure 51: (a) A region page in a k-d-B-tree with capacity 9 that overflows. The partitions induced by splitting it into two region pages at (b) x=50, (c) y=80, and (d) x=70.

pages do not overflow.

For example, suppose that we choose to partition the region page f in Figure 51a at line l at $y = 80$ as shown in Figure 51c. Recall that this page has capacity 9. Let f_1 and f_2 represent the newly-created region page corresponding to the area below and above $y = 80$, respectively. Therefore, all elements of f that are below l (e.g., regions `B`, `C`, `D`, `E`, `F`, `G`, `H`, and `I` in Figure 51c) are inserted into f_1, while all elements of f that are above l (e.g., none in Figure 51c) are inserted into f_2. In addition, the elements of f whose regions are intersected by l (e.g., `A` and `J`) must be split and new regions created which are inserted into f_1 and f_2. The result is that f_1 contains 10 regions and has overflowed which means that it must also be split thereby causing the k-d-B-tree to grow by an additional level. With the appropriate choice of partition lines, this process can be applied repeatedly to create much larger k-d-B-trees. Clearly, this is undesirable and we should choose another

partition line and axis. In fact, it is not difficult to construct another example where both of the newly-created region pages have overflowed (see Exercise 1).

The second problem is that even if neither of the newly-created region pages f_1 and f_2 overflow, then we may still have to recursively apply the splitting process to f_1 and f_2 which may, in turn, cause a descent to the point pages. This recursive application of splitting can be avoided by choosing a partition line which is not intersected by any of the regions that make up the overflowing region page. For example, this is the case if we choose to partition the region page in Figure 51a at $x = 70$ as shown in Figure 51d. By the very nature of the k-d tree, such a partition line can always be found (see Exercise 2). The shortcoming of such a partitioning is that it may lead to very poor storage utilization in the region pages.

Exercises

1. Consider a k-d-B-tree. Give an example partition line and axis whose use to split an overflowing region page f with a capacity c (i.e., f has $c + 1$ regions) yields two newly-created region pages f_1 and f_2 that also overflow.

2. Why can we always find a partition line for a region page in a k-d-B-tree so that when a region page is split upon overflow we do not have to recursively apply the splitting process to the son pages of the overflowing region page?

3. Write a procedure, `KDB_INSERT`, to insert a point in an k-d-B-tree.

4. Write a procedure, `KDB_DELETE`, to delete a point from an k-d-B-tree.

5. Write a procedure, `KDB_POINT_QUERY`, to perform a point query in an k-d-B-tree.

6. Write a procedure, `KDB_RANGE_QUERY`, to perform a range query in an k-d-B-tree.

7.1.2 LSD Trees

The LSD tree [56] chooses a partition line which is not intersected by any of the regions that make up the overflowing region page in the k-d-B-tree thereby avoiding the need for a recursive invocation of the splitting process. This is achieved by partitioning each region page of the k-d-B-tree at the root of the generalized k$^+$-d tree corresponding to the space partition induced by the overflowing region page. As mentioned above, such a partitioning may lead to poor storage utilization by creating region pages which could be almost empty. An extreme example of this situation results from repeated partitions, assuming a bucket capacity c, that create two region pages whose underlying space is spanned by two generalized k$^+$-d trees, having 1 and c leaf nodes, respectively. Recall that these newly-created pages contain pointers to region pages at the next lower level of the k-d-B-tree. Observe that a generalized k$^+$-d tree containing n leaf nodes has a height that can be as large as $n - 1$, where we assume that a tree with one leaf node has a height of 0. Thus the generalized k$^+$-d tree corresponding to the space partition induced by a region page with c leaf nodes may have a height as large as $c - 1$.

Clearly, if the bucket capacity c (the number of pointers to region or point pages here) is large, then the maximum height of the generalized k$^+$-d tree corresponding to the space partition induced by a region page could also be large thereby requiring many tests in performing basic operations on the data such as a point query. The LSD tree alleviates this problem, in part, by choosing the maximum height h of the extended binary tree representation of the generalized k$^+$-d tree corresponding to the space partition induced by the region page to be $\lfloor \log_2 c \rfloor$. In other words, for a given value of c, h is chosen so that $2^h \leq c < 2^{h+1}$. Thus the region page utilization is optimal when the generalized k$^+$-d tree corresponding to the space partition induced by the page is a complete binary tree (i.e., 2^h leaf nodes), while it is at its worst when the space partition is a linear binary tree (i.e., $h + 1$ leaf nodes). Notice also that when c is not a power of 2, then the region page utilization will never be 100%.

As in the k-d-B-tree, the LSD tree is best understood by assuming that we start with a variant of a bucket generalized pseudo k-d tree. Similarly, the key idea behind the LSD tree lies in distinguishing between the treatment of the nonleaf nodes of the tree (i.e., the bucket generalized pseudo k-d tree) from the treatment of leaf nodes that contain the actual data and termed *data buckets* (i.e., point pages using the terminology of the k-d-B-tree). The nonleaf nodes are also grouped into pages of a finite capacity, termed *directory pages* (i.e., region pages using the terminology of the k-d-B-tree). The result is a variant of a B^+-tree where the root, termed an *internal directory page*, always resides in main memory and corresponds to the nonleaf nodes closest to the root of the bucket generalized pseudo k-d tree, while the nodes in the remaining levels, termed *external directory pages*, always reside in external storage. The root (i.e., the internal directory page) is a generalized k^+-d tree of finite size whose leaf nodes point to external directory pages or data buckets. Below, we first describe the external directory pages. This is followed by a description of the internal directory page. We use the LSD tree in Figure 52a to illustrate our explanation.

Each external directory page (as well as the internal directory page) is a generalized k^+-d tree implemented as an extended binary tree of a pre-defined maximum height h. The leaf nodes of each of the binary trees that make up an external directory page are either pointers to data buckets (i.e., point pages, using k-d-B-tree terminology) or other external directory pages (i.e., region pages using k-d-B-tree terminology). The internal nodes correspond to the partition values. The LSD tree definition stipulates that the length of the paths (in terms of the number of external directory pages that are traversed) from a particular leaf node in the internal directory to any data bucket is the same, while the lengths of the paths from different leaf nodes in the internal directory to any data bucket differ by at most one [56] (e.g., Figure 52a).

The pointer structure connecting the elements of the set of external directory pages and the data buckets yields a collection of trees (we are not speaking of the generalized k^+-d trees that make up each external directory page here). The internal nodes of these trees are external directory pages. Each internal node in each tree in this collection has a maximum fanout 2^h. In addition, in each tree, the paths from the root to the leaf nodes (i.e., the data buckets) have the same length, in terms of the number of external directory pages, which is analogous to a stipulation that the trees are 'balanced' (although the individual internal nodes may have a different degree). Moreover, the heights of any two trees can differ by at most one.

When insertion of a data item into the LSD tree causes a data bucket (i.e., a point page using k-d-B-tree terminology) to overflow, the data bucket is split according to a number of possible rules that range between a rule that ensures that the number of data items in the resulting data buckets is approximately the same to a rule that ensures that the size of the space spanned by the two data buckets is the same. These rules are examples of what are termed *data-dependent* and *distribution-dependent* strategies, respectively. The latter is similar to the trie-based decomposition strategies. They differ from the generalized pseudo k-d tree as they are based on a local rather than a global view (i.e., there is no need to know the entire data set). The split will cause the addition of a data bucket (i.e., point page using k-d-B-tree terminology) which will cause an additional node to be inserted into the corresponding ancestor external directory page p.

External directory pages are split if the addition of the new data bucket causes their height to exceed the pre-defined maximum value h. If a split occurs, then the directory page is split at the root of its corresponding generalized k^+-d tree and the split is propagated further up the tree, which may, in turn, eventually cause a node to be added to the internal directory page.

The internal directory page has a finite capacity i which is the maximum number of leaf nodes that it can contain. Note, that the leaf nodes correspond to the regions induced by the generalized k^+-d tree partition of the underlying space. i is usually much larger than the capacity of the external directory pages. The internal directory page is kept in main memory. Again, as in the case of the external directory pages, the internal directory page is an extended binary tree. When the internal directory page is too full, one of its subtrees s is made into an external directory page p and s is removed from the internal directory page (we say that s has been *paged*) In this case, s is replaced by a leaf node that points to p. Subtree s is chosen to be as large as possible subject to the pre-defined maximum height of an external directory page and the 'path length to a bucket' constraints. The latter constraint implies that the path lengths, in terms of the number of external directory pages that are traversed, of all paths from the root of the subtree s to a data bucket must be the same, and that the path length is the minimum of the permitted path lengths (recall that the path lengths may differ

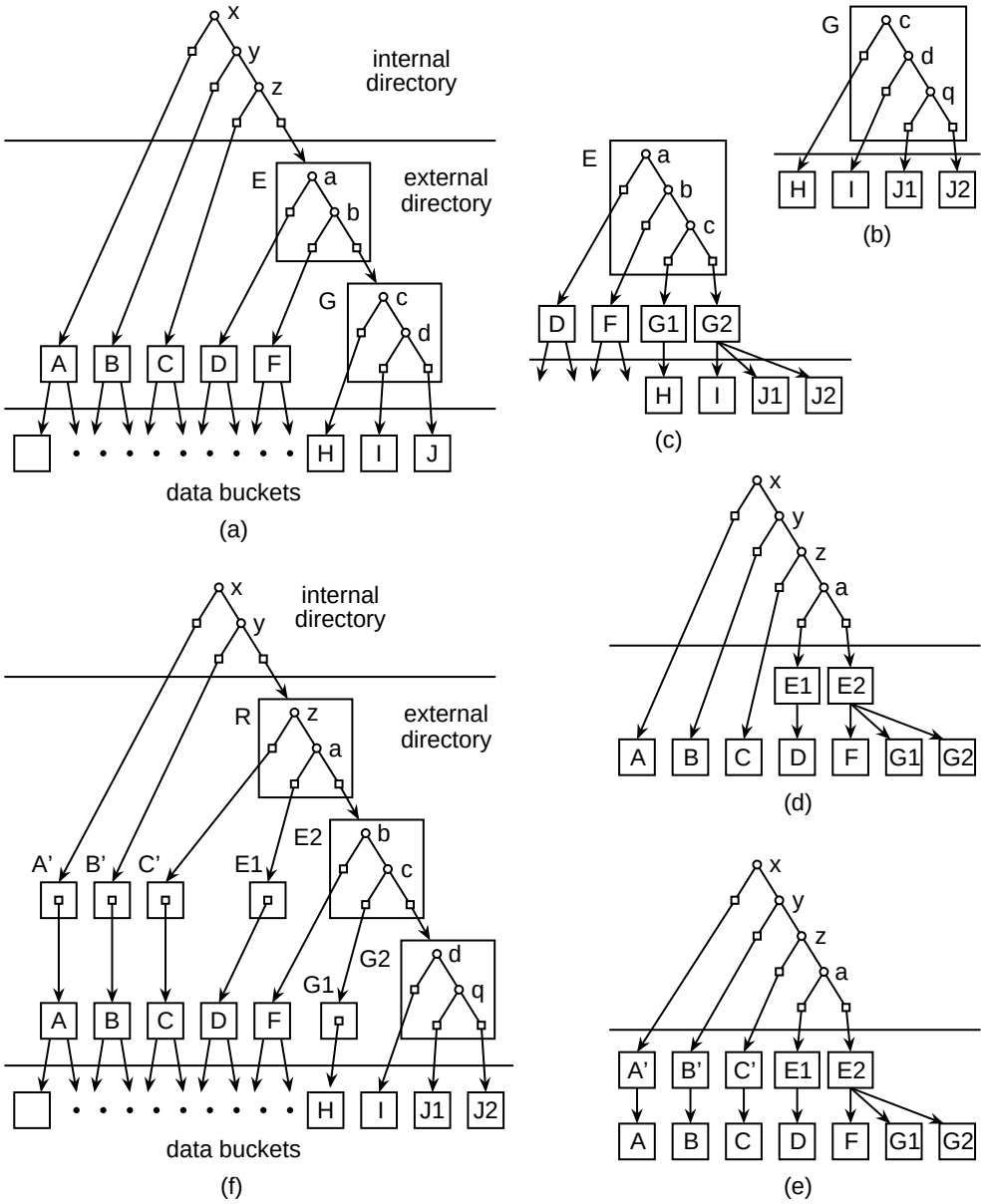


Figure 52: Example LSD tree showing the result of inserting a point in data bucket J which causes it to overflow. (a) Original tree; (b) effect of splitting data bucket J on external directory page G; (c) effect of splitting external directory page G on external directory page E; (d) effect of splitting directory page E on the internal directory page; (e) effect of making new external directory pages upon detecting overflow of the internal directory page so that the lengths of all paths from the root of the internal directory page to any of the data buckets are the same; (f) final tree.

by one). Satisfaction of these constraints ensures that the external directory pages are as large as possible, and also prevents the internal directory page from overflowing too often.

If the subtree s that was chosen has just one node, which must be a leaf node, then the size of the internal directory page has not been reduced. Recall, that leaf nodes are pointers to external directory pages or data buckets. Thus after paging subtree s consisting of just a single leaf node, a pointer to s must be included in the internal directory page in a newly created leaf node t which means that the internal directory page is still

overflowing. However, we have created a new external directory page consisting of one leaf node and thus the path length l from t to all of the data buckets $\{B\}$ accessible from t has increased by one without violating our constraint that l is the same for all elements of $\{B\}$ and that l differs by at most one from those of the remaining trees that make up the collection of external directory pages. In this case, the paging algorithm must be reapplied but now the algorithm has the opportunity to choose another subtree to be made into an external directory page which hopefully consists of more than one leaf node.

For example, letting $i = 4$, $h = 2$, and $c = 4$, consider the situation when data bucket `J` overflows in the LSD tree whose directory map is given in Figure 52a. Data bucket `J` is split into data buckets `J1` and `J2` with `q` serving as the new partitioning hyperplane which is promoted to the parent external directory page `G` (see Figure 52b). This causes `G` to have height 3, which is too large (as $h = 2$). Therefore, `G` is split into external directory pages `G1` and `G2` with partitioning hyperplane `c` serving as the new partition value which is promoted to the parent external directory page `E` (see Figure 52c). Again, this causes `E` to have height 3, which is too large. Therefore, `E` is split into external directory pages `E1` and `E2` with partitioning hyperplane `a` serving as the new partition value which is promoted to the parent who is the internal directory page (see Figure 52d). This causes the internal directory page to overflow as it has pointers to five external directory pages while there is only room for four since $i = 4$.

At this point, the definition of the LSD tree requires that we locate the largest subtree s in the internal directory page such that the length of the path from its root, in terms of the number of external directory pages that are traversed, to a data bucket is a minimum and make s into an external directory page. Continuing our example, this means that we make, in order, external directory pages `A'`, `B'`, and `C'` out of the leaf nodes which are the left sons of the internal nodes `x`, `y`, and `z`, respectively in Figure 52d yielding Figure 52e. We now have the situation that the lengths of all paths from the internal directory page to any of the data buckets are the same. Of course, the internal directory page is still too full. Next, we convert the subtree rooted at internal node `z` in Figure 52e, as it is the largest, into an external directory page as shown in Figure 52f and the insertion process is complete. Notice that the lengths of the paths to any data bucket are the same for any of the nodes in the resulting internal directory page while they differ by just one for the different nodes in the resulting internal directory page.

It should be noted that the example that we described is an example of a worst case in the sense that we had to make external directory pages that were essentially empty (i.e., did not correspond to a partition of the underlying space) when we created external directory pages `A'`, `B'`, and `C'`, as well as `E1` and `G1` in Figure 52e. This worst case is unlikely to arise in practice as the capacity i for the internal directory page is usually chosen to be quite large while the external directory pages and the data buckets are usually the size of a page in the underlying operating system. For example, in experiments reported in [56], the internal directory contained 1000 nodes, while the external directory pages and data buckets were 512 bytes. Assuming an extended binary tree representation for the internal directory page leads to a value of 500 for i as the number of leaf nodes is one greater than the number of nonleaf nodes. Moreover, using 8 bytes per node, the maximum number of nodes that can be stored in an external directory page is 64 which corresponds to an extended binary tree of height $h = 5$ as there are $2^5 = 32$ leaf nodes and $2^5 - 1 = 31$ nonleaf nodes. Thus the page can hold as many as $c = 32$ pointers to other external directory pages and data buckets.

If a cyclic partition order is used, then there is no need to keep track of the partition axes. In this case, the generalized k$^+$-tree in each external directory page is simplified considerably and its extended binary tree representation can be stored using the complete binary tree array representation (see Section **??** of Chapter **??** which is also often used to represent heaps and is sometimes called a sequential heap).

It is interesting to observe (as in the k-d-B-tree) that the structure of the LSD tree is similar to that which would be obtained by taking a binary search tree B of height mh and replacing it by a multiway tree B' of height m where the nodes of B' correspond to aggregates of subtrees of B of height h. Again, this is analogous to the motivating idea for a B-tree with the exception that the fanout at each node of B' is at most 2^h, whereas in a B-tree there is a predetermined minimum fanout as well as a maximum fanout (usually related to the capacity of a page which is also usually the case in the LSD tree [56]). The difference, of course, is that in the case of an LSD tree, the binary search tree is a k-d tree, and each node in the resulting multiway tree represents a sequence of successive binary partitions of a hyper-rectangular region.

Both the k-d-B-tree and the LSD tree result in partitioning the space corresponding to an overflowing region page into two region pages each of which is a hyper-rectangle. The k-d-B-tree permits quite a bit of latitude in the choice of the position of the partition line and identity of the partition axis at the expense of the possibility of further overflow on account of the need for further partitioning of the regions intersected by the partition line. In contrast, there is no further partitioning in the case of the LSD tree as each region page of the k-d-B-tree is partitioned at the root of the generalized k^+-d tree corresponding to the space partition induced by the overflowing region page. The drawback of this approach is that the storage utilization of the region pages resulting from the split may be quite poor.

Exercises

1. Write a procedure, `LSD_INSERT`, to insert a point in an LSD tree.
2. Write a procedure, `LSD_DELETE`, to delete a point from an LSD tree.
3. Write a procedure, `LSD_POINT_QUERY`, to perform a point query in an LSD tree.
4. Write a procedure, `LSD_RANGE_QUERY`, to perform a range query in an LSD tree.

7.1.3 hB-trees

The hB-tree [89, 30, 31] overcomes the low storage utilization drawback of the LSD tree by removing the requirement that the portions of the underlying space spanned by the region pages resulting from the split are hyper-rectangles. Instead, the hB-tree splits each region page into two region pages so that each resulting region page is at least one-third full thereby making the other region page no more than two-thirds full[23] (see Exercises 2). The portions of space spanned by the resulting region pages have the shapes of bricks with holes where the holes correspond to region pages spanning a hyper-rectangular space which has been extracted from the region page that has been split[24]. This analogy serves as the motivation for the name *hB-tree* where *hB* is an abbreviation for a *holey-brick*. Note that the one-third/two-thirds balanced splitting property also holds for the point pages (see Exercise 3).

Each node corresponding to a region page in the hB-tree contains a generalized k^+-d tree[25] describing a partition of a corresponding hyper-rectangular area. However, the space spanned by the node may not cover the entire hyper-rectangle. In this case, some of the generalized k^+-d tree leaf nodes (corresponding to partitions of the hyper-rectangle) are flagged as being external (denoted by `ext`) and thus spanned by other nodes of the hB-tree at the same level (such as its brothers). The best way to understand the hB-tree is to look at how it would deal with the overflowing region page, denoted by `P`, in Figure 51a. It has 10 regions while its capacity is 9. Let Figure 53 be the generalized k^+-d tree corresponding to the region page `P` in Figure 51 which has overflowed. In this case, the space spanned by `P` is a rectangle and thus none of the nodes in the corresponding k^+-d tree are marked as being external. Notice that this example is quite simple as there are no external nodes in the generalized k^+-d trees of any of the child nodes of `P` since they are all also rectangles. We shall look at more complex examples later.

First, we must determine how to split the overflowing region page, say P. This is done by traversing its corresponding generalized k^+-d tree (i.e., Figure 53) in a top-down manner descending at each step the subtree with the largest number of leaf nodes (i.e., regions). The descent ceases as soon as the number of regions in one of the subtrees, say S, lies between one third and two thirds of the total number of regions in the overflowing region page P. The space spanned by S corresponds to the brick which is being extracted from P.

In our example, the stopping condition is satisfied once we have made two left transitions followed by two right transitions in the generalized k^+-d tree of Figure 53. At this point, the brick that we will extract corresponds to the subtree of `P`'s generalized k^+-d tree consisting of the child nodes `C`, `D`, `E`, and `F` (i.e., these

[23] This notion of minimum occupancy is also used in the BD tree [100, 101] (see Section 5.2).
[24] Note that subsequent splits may result in region pages that span areas that are not contiguous but this is not a problem [89]).
[25] We characterize the structure as a *generalized k-d tree* rather than a *k-d tree* because the partitions need not be cyclic.

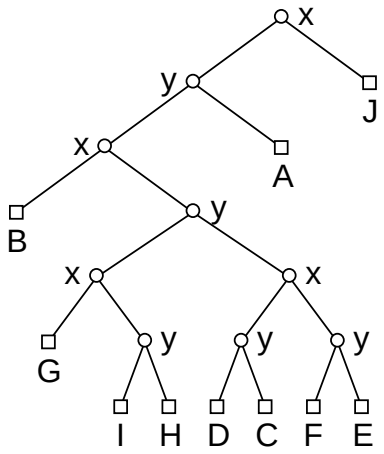


Figure 53: The generalized k^+-d tree of the hB-tree node corresponding to the region page in Figure 51a. The nonleaf nodes are labeled with the name of the key that serves as the discriminator.

are hB-tree nodes at the next lower level in the hB-tree). We let `S` refer to a new hB-tree node consisting of the subtree of `P`'s k^+-d tree and let `Q` refer to the remaining portion of `P`[26] (i.e., the result of extracting `S` from `P`). The generalized k^+-d trees associated with `Q` and `S` are shown in Figures 54a and 54b, respectively. Notice the use of the label `ext` in Figure 54a to mark the portions of the space covered by the generalized k^+-d tree (recall that this space is a hyper-rectangle) that are spanned by another node in the hB-tree. This is needed because the underlying space spanned by its hB-tree node is not a hyper-rectangle. In contrast, no nodes are labeled `ext` in Figure 54b as the underlying space spanned by its hB-tree node is a hyper-rectangle.

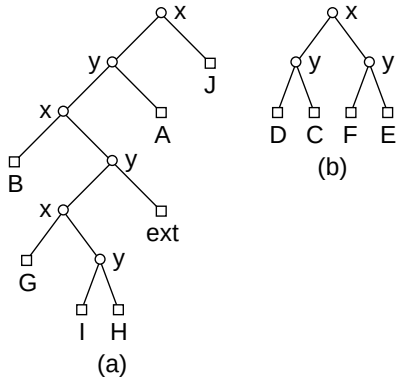


Figure 54: The generalized k^+-d trees associated with the results of splitting the hB-tree node corresponding to the region page in Figure 51a with the generalized k^+-d tree given in Figure 53 into nodes (a) Q and (b) S. The nonleaf nodes are labeled with the name of the key that serves as the discriminator.

We must also propagate the result of the split to the father hB-tree node F of the node P corresponding to the overflowing region page (termed *posting* [89]). In our example, after the split, we post a subtree that discriminates between `Q` and `S` to the father hB-tree node `F`. This results in merging the generalized k^+-d tree in Figure 55 into the generalized k^+-d tree for `F`. Of course, we must also check if F has overflowed which may cause further splitting and posting eventually culminating in the growth of the hB-tree by one level.

Note that as a result of a split we may have the situation that some nodes in the hB-tree will have more than one parent. This means that the directory in the hB-tree is really a directed acyclic graph (i.e., there are

[26] In an actual implementation, `Q` can occupy the same disk page as `P` did.

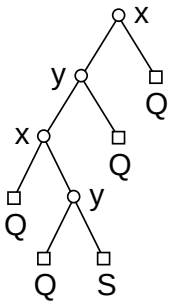


Figure 55: The generalized k$^+$-d tree associated with the father of the hB-tree node corresponding to the region page in Figure 51a with the generalized k$^+$-d tree given in Figure 53 which has been split into nodes Q and S whose generalized k$^+$-d trees are given by Figures 54a and 54b, respectively. The nonleaf nodes are labeled with the name of the key that serves as the discriminator.

no cycles and the pointers are unidirectional) and thus is not actually a tree. As an example of this situation, consider the region page, say P, in Figure 56a whose generalized k$^+$-d tree is given in Figure 56b. Suppose that P has overflowed. Of course, this example is too small to be realistic, but it is easy to obtain a more realistic example by introducing further partition lines into the regions corresponding to A and B in the figure. In this case, we traverse P's generalized k$^+$-d tree looking for a partition that will satisfy our requirement that at least one third of the total number of regions lie in one of the partitions while at most two thirds of the total number regions lie in the other partition. In this example, this is quite easy as all we need to do is descend to the left subtree of the root. The result is the creation of two hB-tree nodes (Figures 56c and 56d) where region A is now split so that it has two parents and the partition line between these parents is posted to the father hB-tree node of P[27]. If subsequent operations result in further splits of region A, then we may have to post the result of these splits to all of the parents of A. The greater the number of such splits that take place, the more complicated the posting situation becomes. In fact, this complexity was later realized to cause a flaw in the original split and post algorithm for the hB-tree [29, 119] and was subsequently corrected as part of the hB$^\pi$-tree [30, 31].

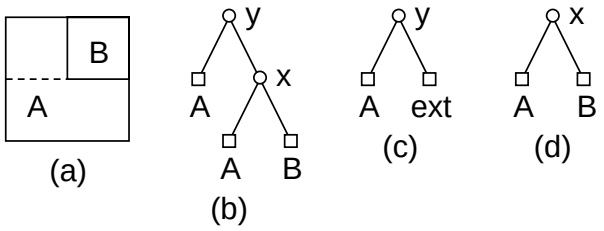


Figure 56: (a) Example region page where (b) is the generalized k$^+$-d tree of its hB-tree node and whose overflow is resolved by partitioning it thereby creating two hB-tree nodes having (c) and (d) as their generalized k$^+$-d trees. The actual partition is of the region page is shown by the broken line in (a). The nonleaf nodes are labeled with the name of the key that serves as the discriminator.

Exercises

1. Give an example point set with $4k$ points in two dimensions which cannot be partitioned by a vertical

[27] The splitting procedure for the hB-tree would actually truncates the generalized k$^+$-d tree in Figures 56c to yield a single leaf node pointing to A. In general, this is done by traversing the generalized k$^+$-d tree from the root and discarding nonleaf nodes until neither of the sons are external markers. However, in the hB$^\Pi$-tree [30, 31], which is an adaptation of the hB-tree designed to provide better support for recovery and concurrency, such a truncation is not performed. This is because in the hB$^\Pi$-tree the external markers are replaced by *sibling pointers* whose value provides useful information. In particular, for a node P, a sibling pointer in its generalized k$^+$-d tree points to the node that was extracted (as a result of splitting) from P whose underlying space spans the corresponding area.

or a horizontal line into two sets each of which has $2k$ points.

2. Prove that it is always possible to split a region page in an hB-tree into two region pages so that each resulting region page is at least one-third full (thereby making the other region page no more than two-thirds full). Recall that the splitting of a region page is achieved by extracting a portion of the k-d tree that describes the partitioning of the region page.

3. Prove that it is always possible to split a point page in an hB-tree into two point pages so that each resulting point page is at least one-third full (thereby making the other point page no more than two-thirds full).

4. Write a procedure, `HB_INSERT`, to insert a point in an hB-tree.

5. Write a procedure, `HB_DELETE`, to delete a point from an hB-tree.

6. Write a procedure, `HB_POINT_QUERY`, to perform a point query in an hB-tree.

7. Write a procedure, `HB_RANGE_QUERY`, to perform a range query in an hB-tree.

7.1.4 K-d-B-tries

Partitions in the k-d-B-tree (as well as the LSD tree and the hB-tree) are usually made in a data-dependent manner[28]. An alternative is to recursively decompose the underlying space from which the data is drawn into halves. The exact identity of the key being partitioned at each level of the decomposition process depends on the partitioning scheme that is used (e.g., whether or not we need to cycle through the keys, etc.). Such decompositions are characterized as being trie-based and have been used earlier in distinguishing between different forms of quadtrees (Sections 4.1 and 4.2) and k$^+$-d trees (Sections 5.1 and 5.2). Some examples of a trie-based decomposition are the PR k-d tree and the generalized k-d trie discussed in Section 5.2.

Trie-based decompositions have the obvious drawback of requiring many decomposition steps when the data is clustered which results in empty subtrees. This is alleviated, in part, by making use of buckets thereby leading to the bucket PR k-d tree and the bucket generalized k-d trie discussed in Section 5.2. The effects of clustering can be further mitigated by increasing the capacity of the buckets. Below, we discuss a number of extensions of the bucket generalized k-d trie (i.e., the multilevel grid file[29] [78] and the buddy-tree [126]) which have additional desirable characteristics. They differ from the bucket generalized k-d trie in that, as in the k-d-B-tree (and the LSD tree), they also aggregate the nonleaf nodes of the bucket generalized k-d trie. We use the term *k-d-B-trie* to refer to this general class of representations.

Our explanation of the k-d-B-trie is heavily influenced by the manner in which two of its variants, namely the multilevel grid file and the buddy-tree, are implemented. They yield a tighter decomposition of the underlying space in the sense that they eliminate some of the empty space that is inherent to the bucket generalized k-d trie. This is achieved by basing the overflow condition for an aggregated nonleaf node (i.e., a region page) on the number of nonempty regions or sons that comprise it. Of course, other variants of the k-d-B-trie are also possible and they often have different definitions. However, what they all have in common is the aggregation of the nonleaf nodes of a bucket generalized k-d trie and their organization using some variant of a k-d trie.

A k-d-B-trie has two types of pages: point pages (capacity b) containing the actual data and region pages (capacity c) that contain pointers to point pages or to other region pages. Initially, we have one empty point page p and one region page r containing p which spans the entire underlying space U[30]. When p becomes full, p is split into two point pages p_1 and p_2 using a variant of a recursive decomposition process described below. We may think of the splitting of p to be the result of applying a sequence of halving operations on the

[28] But see the discussion of distribution-dependent decompositions in LSD tree [56].

[29] Although the qualifier *grid file* is used in the name of the data structure, it has little connection with the grid file as described in Section 7.2.1).

[30] Actually, most implementations dispense with the initial region page. However, in our explanation, we find it useful to assume its existence so that we can see how the underlying space is decomposed once the initial point page, which covered no space at the start, becomes full.

underlying space U. The sequence terminates once the last halving operation results in two nonempty halves, spanning u_1 and u_2. Since the k-d-B-trie is based on the bucket generalized k-d trie, the order in which the halving operations are applied to the domains of the different keys is permitted to vary. This leads to many possible different partitions of the underlying space and configurations of resulting point pages.

We term the result of the application of each halving operation to U a *B-rectangle* [126]. Thus, we see that u_1 and u_2 are B-rectangles. Depending on the configuration of the underlying space, we often find that p_i covers just a very small part of u_i. This will lead to inefficiencies in search and thus we attempt to store a tighter approximation of the data in p_i. Let $M(p_i)$ be the minimum bounding rectangle of p_i[31]. Each minimum bounding rectangle $M(p_i)$ has a minimum bounding B-rectangle $B(M(p_i))$ termed the *B-region* of $M(p_i)$. The B-region of $M(p_i)$ is obtained by continuing the application of recursive halving operations to u_i. With each entry for a point page p_i in the father r, we associate the B-region of $M(p_i)$. Thus we can now see that u_1 and u_2 may be obtained by the application of just one halving operation on the B-region of $M(p)$. In the following, when we speak of the B-region of a node o, we mean the B-region of the minimum bounding rectangle of the data in the subtree at o.

As an example of the tighter approximation that can be obtained, assuming a bucket capacity of 4, consider the insertion of point `X` into the point page in Figure 57a. Figure 57b is one possible result of splitting the overflowing page into two nonoverflowing halves, while Figure 57c shows the B-regions and minimum bounding rectangles that will be stored in the newly-created region page. Thus we see that the k-d-B-trie obtained in Figure 57c has smaller point pages than would have been the case had we not used the B-regions. At a cursory glance we might think that the use of B-regions leads to the creation of more regions by virtue of the presence of the empty regions (e.g., the upper-right quarter of Figure 57c which is marked with diagonal lines). However, this is not a problem as the overflow condition for a region page only depends on the nonempty regions that comprise it.

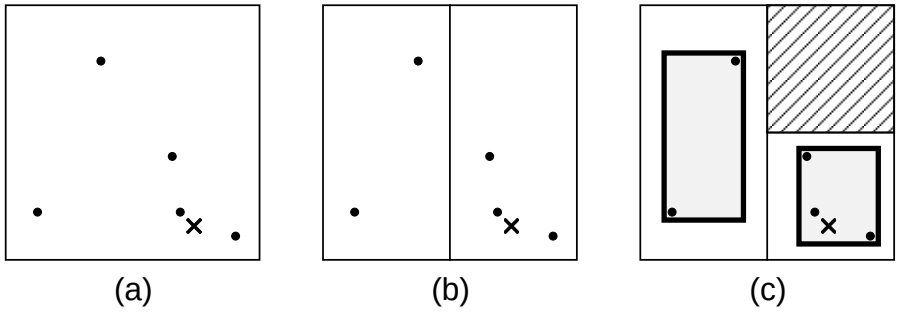


Figure 57: Example showing the insertion of point X into a point page of capacity 4 in k-d-B-trie. (a) Original point page. (b) One possible result of splitting the overflowing point page into two halves. (c) The B-regions and bounding rectangles (shown shaded) that will be stored in the newly-created region page. Empty regions which have no corresponding entry in the region page are marked with diagonal lines.

Each region page r consists of the B-regions of its constituent pages which form a *B-partition* of r. The B-partition has the property that its constituent B-regions are pairwise disjoint. Note that requiring that the B-regions are pairwise disjoint is stronger than just requiring that the minimum bounding rectangles of the son pages (which are point or region pages) are disjoint. For example, although the two minimum bounding rectangles `A` and `B` in Figure 58 are disjoint, their B-regions are not disjoint as they are in fact the same. Empty B-rectangles (corresponding to empty point or region pages) are not represented explicitly in the region page and thus they are not counted when determining if the capacity of r has been exceeded.

Once the initial point page has been split, insertion continues into the existing point pages with splits made as necessary until the number of B-regions (i.e., point pages) exceeds the capacity c of the region page r. At this point, we split r into two region pages r_1 and r_2 by halving the B-region of r and propagate the split up

[31] We assume that the sides of the minimum bounding rectangle are parallel to the axes corresponding to the keys.

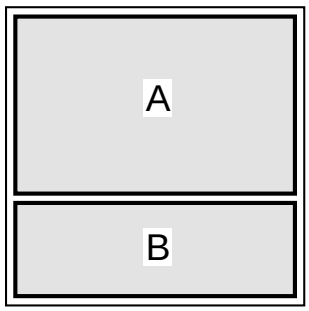


Figure 58: Example pair of disjoint minimum bounding rectangles (shown shaded) whose B-regions are not disjoint.

the tree. If we are at the root of the tree, then we create a new region page s having r_1 and r_2 as its sons. As this process of insertion is continued, s may eventually gain more sons by virtue of splitting r_1 and r_2 until the number of region pages comprising s exceeds c at which time another split operation is applied to s.

Of course, there are times when the newly inserted point q is not contained in any of the existing B-regions of the sons of r. We have two options. The first is to extend the B-region of one of the sons to cover q. The second is to create a new son with a B-region that covers q. Choosing the second option means that we may have to create a chain of new region pages until reaching the leaf level of the k-d-B-trie, unless we allow the k-d-B-trie to be unbalanced (which is allowed in the buddy-tree implementation of the k-d-B-trie).

When a region page is split, we want to avoid the problem that arose in the k-d-B-tree where the partition of a region page could cause the partitioning of some of its constituent region pages. This problem is avoided by requiring that the B-partition that corresponds to each region page forms a generalized k-d trie[32] (such a B-partition is termed *regular* [126]). For example, assuming a region page capacity of 4, the partition given in Figure 59 [126] is not regular. In particular, no hyperplane exists that can partition it into two disjoint sets. The above problem can only arise in the k-d-B-trie when two son pages of a region page are merged, which means that the merging process must ensure that the result of the merge does not violate the regularity requirement (see Exercise 1).

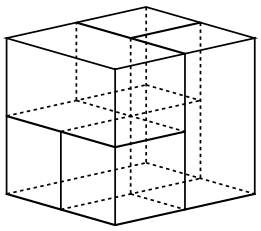


Figure 59: Example of a nonregular B-partition where each of the partitioning hyperplanes will pass through a B-region.

Observe that the generalized k-d trie need not be unique which means that the B-partition need not be unique. For example, consider Figures 60b–d which are three possible generalized k-d tries for the region page in Figure 60a. Observe that in this figure, and for the purposes of facilitating the process of determining whether a B-partition is regular, each leaf node in the generalized k-d trie represents the largest possible bounding B-rectangle for the B-region corresponding to the node thereby obviating the need for empty nodes.

It should be clear that each region page r in a k-d-B-trie is a collection of disjoint B-regions whose union (in terms of the portions of the underlying space that they span) is a subset of the underlying space spanned by r. This property is useful in making the k-d-B-trie a better structure for the execution of search queries

[32] We could have also used the term *generalized k^+-d trie* to describe this k-d trie to reinforce the similarity of the representation of the nodes of the k-d-B-trie to those of the k-d-B-tree.

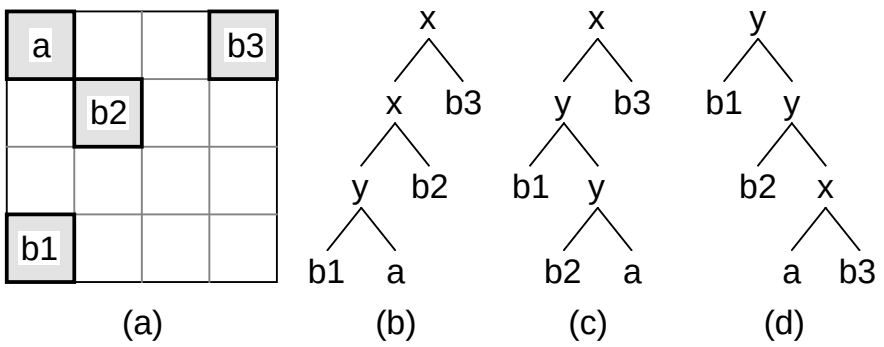


Figure 60: (a) A region page and the generalized k-d tries corresponding to three possible partitions upon overflow given in (b), (c), and (d). The nonleaf nodes are labeled with the name of the key that serves as the discriminator.

(e.g., range queries) as we can avoid examining regions which cannot possibly contain data (e.g., notice the empty region in the upper-right corner of Figure 57c). Thus the B-regions have similar properties to minimum bounding boxes that form the basis of the R-tree. B-regions differ from minimum bounding boxes by virtue of the fact that their sizes and positions are restricted to being a result of the application of a recursive halving process to the underlying space.

Since each B-partition can be represented by a generalized k-d trie, we can represent the B-regions making up both the point and region pages of the k-d-B-trie compactly by just recording the minimum number of bits (i.e., the paths in the generalized k-d trie, one per key, to the B-region, with left and right transitions being represented by 0 and 1, respectively[33]) necessary to differentiate them from the remaining B-regions. Note that in this case we are representing the full generalized k-d trie in contrast to the one depicted in Figure 60 which employed just enough partitions to distinguish the different minimum bounding B-rectangles for the pages making up the B-partition. In contrast, in the k-d-B-tree, we need to store the whole partition value for each key[34].

The k-d-B-trie has great flexibility in determining which nonfull point and region pages are to be merged when deletions take place, as well as when page splitting results in pages that are underfull. This permits the k-d-B-trie to have good dynamic behavior. In particular, assuming d-dimensional data, if we used a PR k-d tree instead of a generalized k-d trie, then a page a could be merged with just one page (i.e., the brother of a). In contrast, in the k-d-B-trie, a can be merged with up to d pages corresponding to the d possible keys whose domain axes could have been split in half when a was created (i.e., as a result of a split). The pages with whom a can be merged are termed its *buddies*. In fact, the definition of a buddy could be made less restrictive by stipulating that two son pages g and h in region page p are buddies if the intersection of the B-region of their union with the B-regions of all remaining pages in p is empty. This means that the maximum number of pages with which a can be merged (i.e., buddies) is even greater than d (see [125, 126]). To see why this is true, recall that the B-regions of the son pages of a region page p can be represented by a number of different generalized k-d tries (whereas there is only one way to do so with a PR k-d tree). Therefore, a page b_i is a buddy of page a if a and b_i occur in sibling leaf nodes in some generalized k-d trie representing the partition of p. For example, in Figure 60, a can be merged with $\mathtt{b}_1$, $\mathtt{b}_2$, or $\mathtt{b}_3$.

There are a number of variants of the k-d-B-trie. The most notable are the multilevel grid file [78] and the buddy-tree [126]. The key difference between the multilevel grid file and the buddy-tree is the fact that in the buddy-tree, a minimum bounding rectangle is stored for each of the son pages (instead of their B-regions, which may be computed from the minimum bounding rectangles as needed), while the multilevel grid file just stores the B-regions with the bit string approach described above (see Figure 61). Another difference is that

[33] A separate path for each key is needed instead of just one sequence of interleaved 0s and 1s because the attributes are not being tested in cyclic order and thus the correspondence between the identity of the tested attribute and the result of the test must be recorded.

[34] In fact, for some implementations of the k-d-B-tree, associated with each region entry in a region page is a rectangle, requiring two values for each key, representing the region.

the buddy-tree exploits the less restrictive definition of a buddy thereby permitting a page to be merged with a larger range of pages. In addition, the buddy-tree is not always balanced in order to prevent the existence of region pages with only one son page (see below for more details). Finally, the multilevel grid file employs a hashing function which is applied to the various keys prior to the application of the trie-like decomposition process. This enables indexing on a wider variety of key types, such as strings of characters, etc.

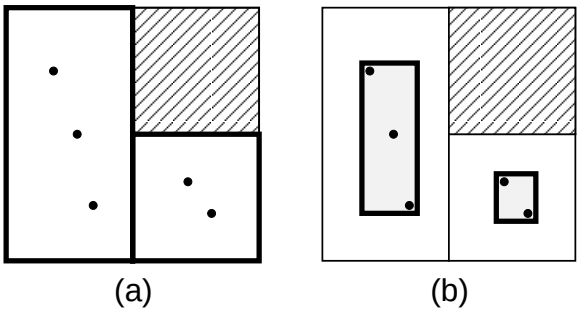


Figure 61: Example illustrating the difference between a (a) multilevel grid file and (b) a buddy-tree. Notice the use of minimum bounding rectangles (shown shaded) in (b).

The fact that empty B-rectangles are not represented in a region page r leads to an interesting situation when a point q being inserted into r does not lie in the B-regions of any of the son pages of r. In other words, q is in an empty B-rectangle. Recall that the point insertion procedure is top-down in the sense that we start at the root of the k-d-B-trie and search for the point page containing q. The search ceases once we encounter such a point page or if we encounter an empty B-rectangle e in a region page r. In the latter case, we have two options. The first option is to attempt to enlarge one of the existing B-regions to accommodate the point. If this is not possible, then the second option is to create a new page for q. Let us look more closely at how we attempt to enlarge one of the existing B-regions. This is achieved by checking whether the B-region of q (i.e., the B-region of the minimum bounding rectangle of q) can be merged with the B-region of one of its buddies in r. If such a buddy a exists, and if a is not already filled to capacity, then we enlarge the B-region of a to accommodate q in the corresponding entry of region page r and continue by inserting q into a. If no nonfull buddy exists, then we resort to the second option and create a new point page v of one point.

The newly-created point page should be at the deepest level of the k-d-B-trie. However, this would lead to a long sequence of region pages with just one nonempty entry. This is quite inefficient, and thus in some implementations of the k-d-B-trie (e.g., in the buddy-tree but not in the multilevel grid file) an unbalanced structure is used where the new point page v is added at the depth i of the empty B-rectangle t in which q fell. However, if this new point page v at depth i overflows at some later instance of time, then v is pushed one level down in the k-d-B-trie to depth $i+1$ and then split. This action is taken instead of splitting v at depth i and possibly having to split the region page w at depth $i-1$ when w overflows as a result of the split at depth i.

Note that when the keys are clustered, methods such as the k-d-B-trie do not yield a very good partition of the underlying space. However, results of experiments with the buddy-tree [126] show that the fact that empty B-rectangles are not represented explicitly ameliorates this drawback to such an extent that it is no longer significant. Nevertheless, the definition of the multilevel grid file does take this into consideration by virtue of the hashing functions which, we recall, are applied to the various keys prior to the application of the trie-like decomposition process. However, it should be noted that, in general, it is quite difficult to devise an order-preserving hashing function that significantly reduces clustering. Moreover, the use of a nonorder preserving hashing function (which can be devised to spread the hash values virtually uniformly) is a drawback for range queries.

The BANG file [42, 46, 47, 43] is a variant of the k-d-B-trie which requires that the underlying space be decomposed by a sequence of partitions that cycle through the keys in a fixed order. Moreover, unlike other variants of the k-d-B-trie (such as the buddy tree and multilevel grid file), the BANG file removes the

requirement that the portions of the underlying space that are spanned by the region pages are hyper-rectangles. Instead, the BANG file only requires that the space spanned by the set of regions that comprise each region page be capable of being described by a PR k-d tree where the individual regions are unions of the blocks corresponding to collections of leaf nodes of the PR k-d tree. In this sense, the region pages are similar to those in the hB-tree except that the BANG file makes use of regular decomposition. In particular, as mentioned above, in the BANG file, the underlying space is decomposed by cycling through the various keys in a fixed and predetermined cyclic order (e.g., $xyxyx\cdots$). Thus the BANG file is a variant of the k-d-B-trie in a similar way as the hB-tree is a variant of the k-d-B-tree.

As an example of the type of space decomposition permitted by the BANG file, consider the region page in Figure 62a and its corresponding PR k-d tree in Figure 62b which consist of the four regions `A`, `B`, `C`, and `D`. Observe that using the parlance of hB-trees, regions `A` and `B` play the role of holey bricks while regions `B` and `D` play the role of the bricks that are removed from `A`, and region `C` plays the role of the brick that is removed from `B`. In the BANG file, the various regions prior to the removal of the bricks are represented by a bit string corresponding to the path from the root of the PR k-d tree of the region page to the nearest common ancestor of all leaf nodes belonging to the same region. For example, letting `0` correspond to branches to the left and below the partition line while letting `1` correspond to branches to the right and above a partition line, we have that `A` is represented by the empty bit string (as the nearest common ancestor is the root of the PR k-d tree), while `B` is represented by `10`.

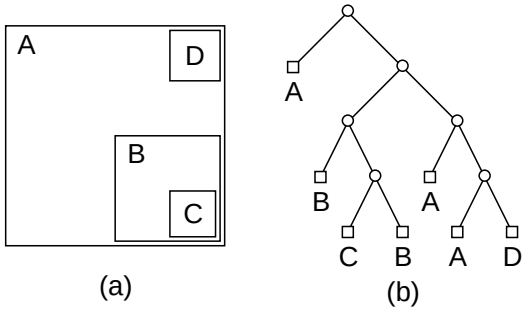


Figure 62: (a) Example illustrating a region page in the BANG file and (b) its corresponding PR k-d tree.

The execution of operations such as a point query using the BANG file is facilitated by sorting the regions corresponding to each region page in the BANG file in order of increasing size of their blocks. The point query is performed by testing the regions for containment of the query point p in this order and stopping the search as soon as a region is found to contain p. This will guarantee that the smallest region enclosing p is selected. The process of searching for the block containing p can be implemented by sorting the bit strings in lexical order (e.g., empty, 0, 011, 1, 101, etc.). Notice that this will order the bit strings of larger blocks before the bit strings of small blocks contained in them. The search itself is done by performing a prefix bit matching in the resulting list in reverse order (i.e., starting from the end) for the bit string describing p.

The use of bit strings to represent the various regions in the BANG file is similar to that used in the BD tree [100, 101][35], while quite different from the one used in the multilevel grid file [78][36]. Nevertheless, a key difference between the BANG file and the BD tree is the fact that the BD tree does not aggregate the nonleaf nodes of the structure into buckets. In particular, in the BD tree, only the leaf nodes are aggregated into buckets. Therefore, the BANG file is related to the BD tree in a way similar to the way that the k-d-B-tree is related to the bucket generalized pseudo k-d tree. An even more precise analogy is that the BANG file

[35] The difference is that in the BANG file the bit strings corresponding to the regions are kept in a sorted list according to their size while in the BD tree they are kept in a binary tree. This results in a different search process, possibly visiting different regions, although, of course, the end result for both representations is the same!

[36] Recall that in the multilevel grid file, a set of bit strings, one per key, is associated with each page since the keys are partitioned in arbitrary order rather than a cyclic order as in the BANG file, thereby precluding the possibility of using just one bit string, which is what is done in the BANG file. However, more importantly, in the multilevel grid file, the bit strings represent rectangular regions as in the BANG file, but all the regions in a given page are disjoint rather than being allowed to be nested as is the case for the BANG file.

is related to the BD tree in the same way as the k-d-B-trie (e.g., the buddy tree or the multilevel grid file) is related to the bucket PR k-d tree as they all also make use of regular decomposition.

Unfortunately, the BANG file has a serious flaw [45] in that the possibility of nodes having multiple parents was not taken into account in the original design. This may occur, just as it does in the hB-tree, when the only way of splitting a node a in a balanced manner via the extraction of a hyper-rectangle from the region covered by a is to split a so that the region covered by some child node intersects the regions of both of the nodes resulting from the split. For an example of such a situation, recall Figure 56. Of course, it could be argued that this example is not realistic in the sense that we can usually find an alternative partition which avoids this problem. However, in order to demonstrate a situation where no alternative partition is possible, a very complex example must be constructed, which we do not do here.

Exercises

1. Give an algorithm to test whether two pages in a k-d-B-trie can be merged. Recall that two pages can be merged only if the resulting page is not too full and if the result is a B-partition.
2. Do we need to test if the result of a page split in a k-d-B-trie is a B-partition?
3. Write a procedure, `BUDDY_INSERT`, to insert a point in a buddy-tree.
4. Write a procedure, `BUDDY_DELETE`, to delete a point from a buddy-tree.
5. Write a procedure, `BUDDY_POINT_QUERY`, to perform a point query in a buddy-tree.
6. Write a procedure, `BUDDY_RANGE_QUERY`, to perform a range query in a buddy-tree.
7. What is the maximum number of possible buddies in a buddy-tree?

7.1.5 BV-trees

The lifting of the requirement that the portions of the underlying space spanned by the region pages are hyper-rectangles is a central principle behind the development of both the hB-tree and the BANG file. Recall that both the hB-tree and the BANG file are designed to overcome the cascading split problem that is common to the k-d-B-tree in the sense that the insertion of a split line may cause a recursive application of the splitting process all the way to the point pages. This means that we do not have a bound on the number of nodes in the directory structure as a function of the number of data points. Moreover, as we saw, the directories of neither the hB-tree nor the BANG file are trees (recall that they are directed acyclic graphs) since it may be impossible to avoid the situation that some of the regions in the directory node (i.e., region page) being split will fall partially into both of the directory nodes resulting from the split.

The occurrence of this problem in the BANG file is addressed by Freeston [45] by use of a clever innovation. The innovation is to decouple the hierarchy inherent to the tree structure of the directory (made up of point and region pages) from the containment hierarchy associated with the process of the recursive decomposition of the underlying space from which the data is drawn. Below, we use the term *BV-tree* to denote the tree structure of the directory obtained from applying the technique described in [45]. The goal is for the BV-tree to have properties similar to the desirable properties of the B-tree while avoiding the problems described above. In particular, a bounded range of depths (and hence a maximum) can be guaranteed for the BV-tree based on the maximum number of data points, and capacity (i.e., fanout) of the point and region pages (they may differ). These bounds are based on the assumption that each point page is at least one-third full [45]. Moreover, in spite of the fact that the BV-tree is not quite balanced, search procedures always proceed level to level in the decomposition hierarchy. For exact match point search, this means that the the number of steps to reach a point page is equal to the depth of the decomposition hierarchy.

In order for the BV-tree technique to be applicable, the regions in the containment hierarchy are required to be disjoint within each level. Furthermore, the boundaries of regions at different levels of the containment

hierarchy are also required to be non-intersecting. These requirements hold when the decomposition rules of both the BANG file and the hB-tree are used for the containment hierarchy. However, in the case of the BANG file, their satisfaction when splitting and merging nodes is almost automatic due to the use of regular decomposition whereas in the case of the hB-tree we need to make use of a k-d tree to represent the history of how the regions were split. In contrast, in the case of the BANG file, the splitting history is implicitly represented by the bit string corresponding to each region.

As in the BANG file and in the hB-tree, there are two ways in which regions in the BV-tree are created that satisfy these properties. In particular, the underlying space (e.g., Figure 63a) is recursively decomposed into disjoint regions either by splitting them into what we term *distinct regions* (e.g., Figure 63b) or *contained regions* (e.g., Figure 63c). Splitting into contained regions is achieved via the extraction of smaller-sized regions thereby creating one region with holes. The disjointness of the regions at a particular level in the decomposition hierarchy ensures that the outer boundaries of the regions do not intersect (although they may touch or partially coincide). The BV-tree is built in a bottom-up manner so that data is inserted in point pages (of course, the search for the point page to insert into is done in a top-down manner). Overflow occurs when the point page is too full at which time the point page is split thereby leading to the addition of entries to the region page that points to it which may eventually need to be split when it points to more point pages than its capacity.

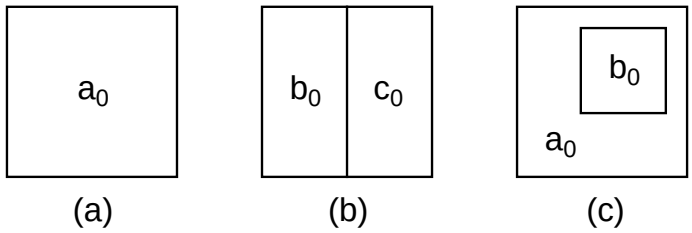


Figure 63: (a) Example region and the result of splitting it into (b) distinct regions or (c) contained regions.

Before proceeding further, let us define some of the naming and numbering conventions that we use in our presentation. For a BV-tree of height h (i.e., with h levels of decomposition), the levels in the decomposition hierarchy are numbered from 0 to $h-1$, where 0 is the lowest level (i.e., corresponding to point pages). Each level has a number of regions associated with it that span the entire underlying data space. The levels in the directory hierarchy are numbered from 1 to h, with level 1 containing the leaf nodes. The reason for the apparent discrepancy in the numbering is to enable a node at level v in the directory hierarchy to be (normally) associated with a region at level v in the decomposition hierarchy. For example, we can view the region representing the entire data space as being at level h in the decomposition hierarchy, and the root node (at directory level h) also represents the entire data space. As we see, the use of the term *level* can be quite confusing. Unless qualified by explicitly using the full expression *directory level*, when we talk about the *level* of a directory node n, we are actually referring to the level in the containment hierarchy of the region that corresponds to n. It is important to distinguish between a level in the decomposition hierarchy and a level in the directory hierarchy, since these two hierarchies are decoupled in the BV-tree. In fact, the result of the decoupling is that a node at (decomposition) level v may actually reside in the directory at a higher (i.e., shallower) directory level, but not lower (i.e., deeper), than v. Note, however, that this stipulation is equivalent to one that states that the region associated with the node at level v appears as an entry of a node at directory level $v+1$ or higher. The full ramification of the decoupling innovation will become clearer as our explanation unfolds.

As mentioned above, the containment relationship (i.e., the nonintersection of the outer boundaries) holds for the boundaries of all pairs of regions including those at the same level of decomposition. However, the containment relationship does not require that regions at deeper levels of decomposition (i.e., smaller level numbers) be contained in regions at shallower levels of decomposition (i.e., larger level numbers).

The containment condition can be restated as stipulating that the interiors of any two regions a and b (regardless of whether or not their holes are taken into account) are either disjoint (with boundaries that are per-

mitted to partially coincide) or one region is completely contained in the other region (i.e., a is in b or b is in a). From the standpoint of the structure of the directory hierarchy, the containment condition implies that if a region page n (i.e., a directory node) is split (on account of having overflowed) into region pages a and b such that the area spanned by one of the child region pages r of n (which must be redistributed into a and b) intersects the areas spanned by both a and b, then the area spanned by r must completely contain either the area spanned by a or the area spanned by b. In the BV-tree, when this happens, r (and most importantly the area spanned by it) will be 'associated' with the resulting region page (termed its *guard*) whose area it completely contains (see below for more details about the mechanics of this process).

The BV-tree definition does not stipulate how the underlying space spanned by the elements of the directory structure (i.e., region pages) is to be decomposed nor the shape of the resulting regions, provided that their outer boundaries do not intersect. In particular, the underlying space may be decomposed into regions of arbitrary shape and size — that is, neither the space spanned by the regions need be hyper-rectangular nor, as is the case for the hB-tree and BANG file, must the faces of the regions be parallel to the coordinate axes. However, since the space decomposition generated by the BANG file satisfies the nonintersecting outer boundary requirement, the BV-tree method may be used to obtain a variant of the BANG file (i.e., a regular decomposition of the underlying space) that does not suffer from the multiple parent flaw.

The fact that regions created at certain levels of decomposition of the underlying space may be associated with region pages (i.e., directory nodes) at shallower directory levels in the directory structure means that the union of the regions associated with all of the region pages found at directory level i of the directory structure of the BV-tree does not necessarily contain all of the data points. This is not a problem for searching operations because in such a case each region page (i.e., directory node) also contains entries that correspond to regions resulting from splits at deeper levels of the decomposition hierarchy. These node entries are present at the shallower directory level because otherwise they and their corresponding regions would have had to be split when the node corresponding to the region page in which they were contained was split. Instead, they were promoted to the next shallower directory level along with the newly-created region page that they contained. The directory structure entries that correspond to space decompositions whose corresponding region pages were created at deeper levels of the decomposition hierarchy (i.e., the ones that were promoted) serve as *guards* of the shallower directory level region pages and are carried down the directory structure as the search takes place.

Since promotion can take place at each directory level, a situation could arise where guards are promoted from level to level. For example, suppose that region a has been promoted from directory level i to directory node x at directory level $i+1$ where a serves as a guard of region b. Next, suppose that x overflows, such that its region c is split into regions d and e, which replace c in the parent node y of x. Now, it is possible that region b contains either d or e, say e. In this case, b is promoted to y at directory level $i+2$ where it serves as a guard of e, and thus a must be promoted to directory level $i+2$ as it is a guard of b. Subsequent splits can result in even more promotion for a. Thus we see that every unpromoted entry in a region page at (decomposition) level i (which may be as deep in the directory hierarchy as directory level i) can have as many as $i-1$ guards where we once again assume that the deepest region pages are at level 1 (but they contain entries that correspond to regions at level 0).

The best way to see how the BV-tree works is to consider an example of how it is built. In our description we will use the term *directory node* instead of *region page* to make it clearer that we are dealing with entries in the directory. As mentioned above, each directory node a corresponds to a region in the underlying space. The entries in a contain regions and pointers to their corresponding point page or directory node (i.e., entries with regions at level 0 correspond to point pages but regions at a higher level correspond to directory nodes). When a is at level i (i.e., its corresponding region is at decomposition level i), its entries may be at level 0 to $i-1$. The entries whose corresponding regions are at levels lower than $i-1$ have been *promoted* to the node a which is at a higher level in the directory where they serve as *guards* of some of the entries in a. Each entry in a also contains information about the decomposition level of the region it contains, so that we may distinguish promoted and unpromoted entries.

a contains a maximum of m entries where m is the fanout of a. a is split if it overflows — that is, if the total number of entries in a (not distinguishing between promoted and unpromoted entries) exceeds m. In our

example, we assume a fanout value of four for the directory nodes. When a directory node is split, the BV-tree definition [45] claims that it should be possible to distribute its entries in the newly-created directory nodes a and b so that each of a and b is at least one-third full as this is one of the properties of the BV-tree that enables it to guarantee an upper bound on the maximum depth of the directory hierarchy. As we will see, this may not always be possible for all fanout values thereby requiring some adjustments in the definition of the fanout.

Let us start with a large set of points which exceeds the capacity (i.e., fanout) of the region so that we split it into two regions $\mathtt{a}_0$ and $\mathtt{b}_0$ having a decomposition and directory hierarchy given in Figures 64a and 64b, respectively.

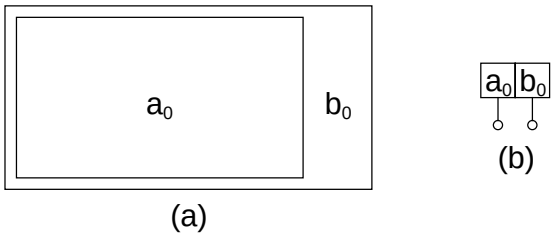


Figure 64: Initial step in creating a BV-tree. (a) Decomposition hierarchy. (b) Directory hierarchy.

Next, enough points are added so that the underlying space is split three times thereby resulting in five regions $\mathtt{a}_0$, $\mathtt{b}_0$, $\mathtt{c}_0$, $\mathtt{d}_0$, and $\mathtt{e}_0$ as shown in Figure 65a. However, this causes the single directory node to overflow and it is split into two nodes corresponding to regions $\mathtt{a}_1$ and $\mathtt{b}_1$ shown in Figure 65b. Since region $\mathtt{a}_0$ is included in both $\mathtt{a}_1$ and $\mathtt{b}_1$, while completely containing $\mathtt{a}_1$, $\mathtt{a}_0$ is promoted to the newly-created directory node where it serves as a guard on region $\mathtt{a}_1$. The resulting directory hierarchy is shown in Figure 65c. Note that $\mathtt{a}_0$ is the region which, if it were not promoted, we would have to split along the boundary of $\mathtt{a}_1$. By avoiding this split, we are also avoiding the possibility of further splits at lower levels of the directory hierarchy. Thus the fact that $\mathtt{a}_0$ appears at this shallower level in the directory hierarchy ensures that $\mathtt{a}_0$ will be visited during any search of the BV-tree for a point on either side of the branch that would have been made if we would have split $\mathtt{a}_0$ which we would have had to do if we did not promote it.

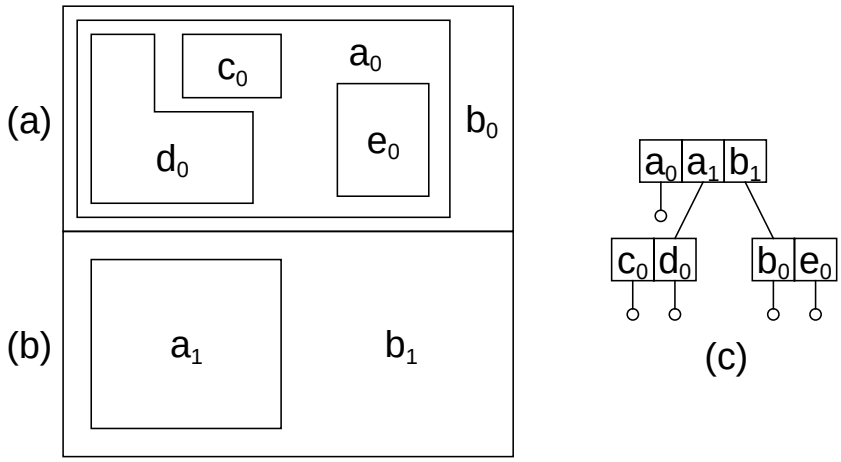


Figure 65: Second step in creating a BV-tree. (a) Decomposition hierarchy at the lowest level. (b) Decomposition hierarchy at the highest level. (c) Directory hierarchy.

Now, suppose that more points are added so that the underlying space is split a number of times. We do not go into as much detail as in our explanation except to indicate one possible sequence of splits.

1. Split $\mathtt{d}_0$ into $\mathtt{d}_0$, $\mathtt{f}_0$, and $\mathtt{g}_0$. This is followed by splitting $\mathtt{d}_0$ into $\mathtt{d}_0$ and $\mathtt{h}_0$. This results in the directory

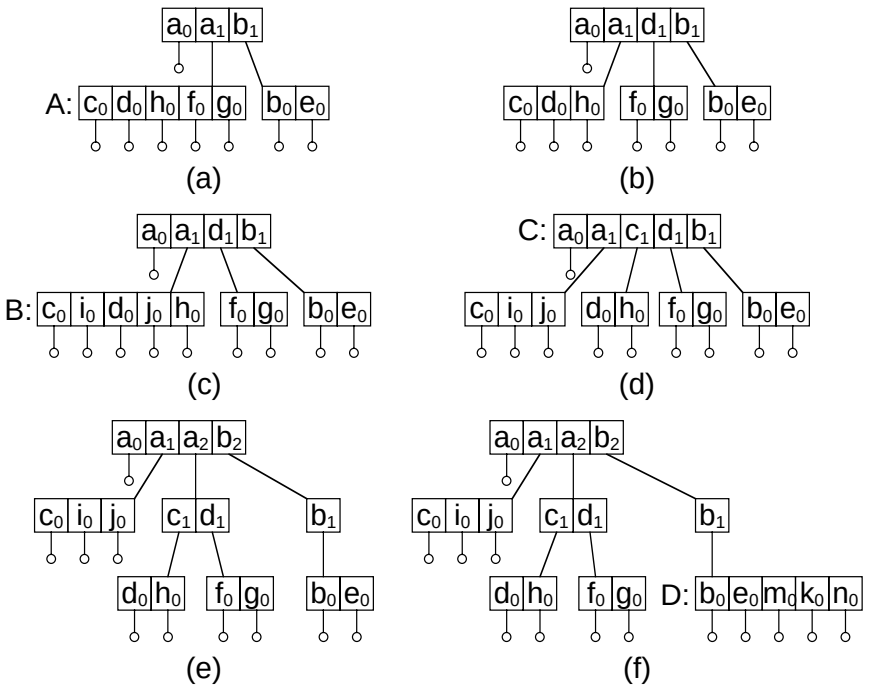


Figure 66: Sequence of intermediate directory hierarchies illustrating the node splitting operations taking place when points are added to the BV-tree of Figure 64 until obtaining the BV-tree of Figure 67.

hierarchy given in Figure 66a where the directory node corresponding to a_1, labeled A, must be split as it overflows. A is split into two directory nodes corresponding to regions a_1 and d_1 resulting in the directory hierarchy given in Figure 66b.

2. Split c_0 into c_0 and i_0. This is followed by splitting d_0 into d_0 and j_0. This results in the directory hierarchy given in Figure 66c where the directory node corresponding to a_1, labeled B, must be split as it overflows. B is split into two directory nodes corresponding to regions a_1 and c_1 resulting in the directory hierarchy given in Figure 66d where the directory node corresponding to the root, labeled C, must be split as it overflows.

3. C is split into two directory nodes corresponding to regions a_2 and b_2 resulting in the directory hierarchy given in Figure 66e. A number of items are worthy of note.

   (a) The directory node corresponding to the root also contains region a_1 as a_1 is included in both a_2 and b_2, while completely containing a_2. Thus a_1 becomes a guard of a_2.

   (b) Since a_0 originally guarded a_1, a_0 continues to do so, and we find that a_0 is also promoted to the root of the directory hierarchy.

   (c) The directory node corresponding to region b_2 only contains the single region b_1. This means that our requirement that each directory node be at least one-third full is violated. This cannot be avoided in this example and is a direct result of the promotion of guards. This situation arises primarily at shallow levels in the directory when the fanout value is low compared to the height of the directory hierarchy. It is usually avoided by increasing the value of the fanout as described later in our discussion.

4. Split e_0 into e_0, k_0, and m_0. This is followed by splitting a_0 into a_0 and n_0. This results in the directory hierarchy given in Figure 66f where the directory node corresponding to b_1, labeled D, must be split as it overflows. There are a number of ways to split D. One possibility, and the one we describe, is to split D into two directory nodes corresponding to regions b_1 and e_1 resulting in the directory hierarchy

given in Figure 67d. Since region e_0 is included in both e_1 and b_1, while completely containing e_1, e_0 is promoted to the directory node containing b_1 and e_1 where it serves as a guard on region e_1. The three levels that make up the final decomposition hierarchy are shown in Figure 67a–c. Notice that we could have also split D so that e_1 contains e_0, k_0, and m_0. In this case, e_0 would not be needed as a guard in the directory node containing b_1 and e_1.

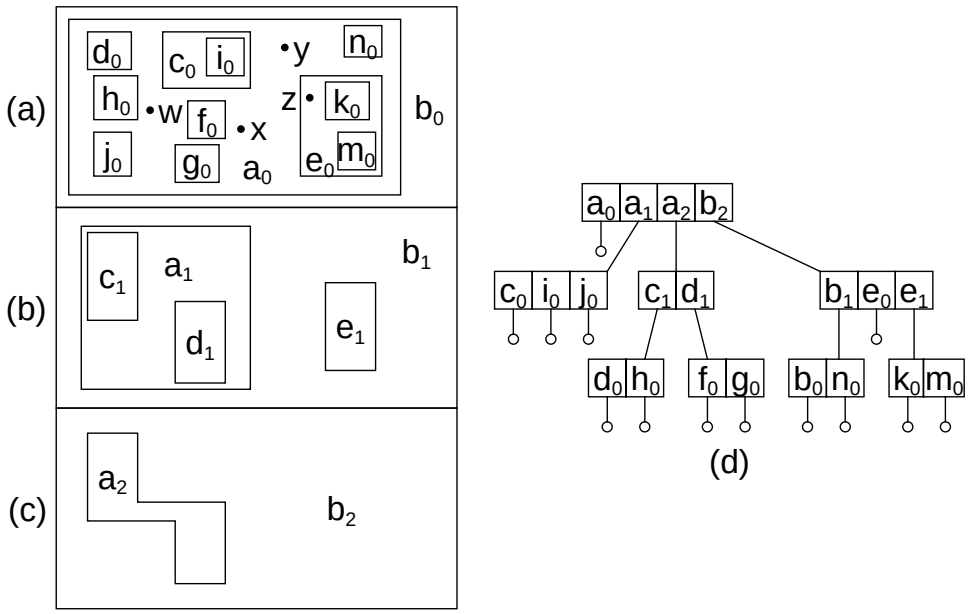


Figure 67: Final step in creating a BV-tree. (a) Decomposition hierarchy at the lowest level. (b) Decomposition hierarchy at the intermediate level. (c) Decomposition hierarchy at the highest level. (d) BV-tree.

Having shown how to construct the BV-tree, we now examine the process of searching a BV-tree for the region that contains a point p (termed an *exact-match search* here). The key idea behind the implementation of searching in the BV-tree is to perform the search on the decomposition hierarchy rather than on the directory hierarchy. At a first glance this appears to be problematic as the BV-tree only represents the directory hierarchy explicitly, while the decomposition hierarchy is only represented implicitly. However, this is not a problem at all as the decomposition hierarchy can be, and is, reconstructed on-the-fly as necessary with the aid of the guards when performing the search. In particular, at each level of the search, the relevant guards are determined and they are carried down the directory hierarchy as the search proceeds. Note that all exact-match searches are of the same length and always visit every level in the decomposition hierarchy (see Exercise 2).

We explain the search process in greater detail below. Initially, we differentiate between the search depending on whether the directory node being searched is the root of the directory hierarchy. Recall that all unpromoted entries in the root correspond to regions at level $v-1$ in the decomposition hierarchy. The search starts at the root of the directory hierarchy at level v (assuming that the deepest directory node is at level 1 and describes regions at level 0 in the decomposition hierarchy). It examines the regions corresponding to all of the entries (i.e., including the guards) and selects the smallest region r (corresponding to entry e_r) that contains the query point p. e_r is termed the *best match*. The level $v-1$ directory node at which the search is continued depends on whether e_r is a guard.

1. e_r is not a guard: Continue the search at the directory node pointed at by e_r with a guard set formed by the current set of guards of e_r (as many as $v-1$ guards).

2. e_r is a guard: Find the smallest region s containing r whose corresponding entry e_s is not a guard. e_s must exist (see Exercise 3). The area spanned by s may also contain other regions, say the set O,

corresponding to guards whose regions contain r. Continue the search at the directory node pointed at by e_s with a guard set consisting of e_r, the set of entries corresponding to the regions that comprise O, and the guards that correspond to regions that contain s. The guards that correspond to regions that contain s are from levels for which O does not contain guards. For each of these levels, choose the guard which has the smallest region that contains p. In other words, the guard set can contain at most one guard per level and a maximum of $v - 1$ guards.

Searching at directory node a at level u other than the root uses a similar procedure as at the root. Again, we wish to determine the level $u - 1$ directory node at which the search is to be continued. This is done by finding the smallest region r (termed the *best match*) containing p. The difference is that the best match is obtained by examining the guard g at level $u - 1$ (if it exists) of the u guards at levels 0 through $u - 1$ that comprise the guard set that was brought down from the previous search (i.e., from level $u + 1$) and the entries in a that are not guards (note that it may be the case that none of the entries in a are guards as can be seen by examining point w and the directory node corresponding to $\mathtt{a}_2$ in Figure 67a). There are two options:

1. If the best match is the region corresponding to g, then continue the search at the level $u - 1$ directory node pointed at by g. The guard set must also be updated as described below.

2. Otherwise, the best match is r, which corresponds to an entry in a (say e_r), and continue the search at level $u - 1$ at the directory node pointed at by e_r. Use a guard set obtained by merging the matching guards found at level u with those brought down from level $u + 1$. Discard g, the level u guard brought down from level $u + 1$ (if it exists), as g cannot be a part of a directory node at level $u - 1$ since each entry at level i of the directory hierarchy can contain at most i guards. Note that two guards at a given level are merged by discarding the one that provides the poorer match.

3. Otherwise, the best match is r, which corresponds to an entry in a (say e_r), and we continue the search at level $u - 1$ at the directory node pointed at by e_r. Use a guard set obtained by merging the matching guards found in a at level u with those brought down from level $u + 1$. Discard g, the level $u - 1$ guard brought down from level $u + 1$ (if it exists), as the search will next be on regions at level $u - 2$. Note that two guards at a given level are merged by discarding the one that provides the poorer match.

Interestingly, there is really no need to differentiate the treatment of root nodes from that of nonroot nodes. In particular, in both cases, only the unpromoted nodes (i.e., entries whose corresponding regions are not guards) need to be examined. In the case of the root node, initially no guard set is brought down from the previous level. Thus it is initially empty. In this case, the new guard set is formed by finding the guard for each level that has the smallest region that contains p.

In order to understand the algorithm better, let us see how we find the regions in the BV-tree of Figure 67a that contain the points w, x, y, and z which are $\mathtt{a}_0$, $\mathtt{a}_0$, $\mathtt{a}_0$, and $\mathtt{e}_0$, respectively. Notice that each region has been labeled with subscripts denoting the decomposition level where it occurs. This is also two more than the deepest directory level at which it can serve as a guard, and one more than the deepest directory level at which it will get carried down in the guard set.

1. The search for w first examines the root, at level 3, and determines $\mathtt{a}_2$ to be the best match. The search continues at the directory node pointed at by $\mathtt{a}_2$ at level 2 (i.e., containing $\mathtt{c}_1$ and $\mathtt{d}_1$) with a guard set consisting of $\mathtt{a}_0$ and $\mathtt{a}_1$. There is no match in the entries in $\mathtt{a}_2$ (i.e., neither $\mathtt{c}_1$ nor $\mathtt{d}_1$ contain w) and thus we use the guard $\mathtt{a}_1$ brought down from level 3 as the best match. The search continues at the directory node pointed at by $\mathtt{a}_1$ at level 1 (i.e., containing $\mathtt{c}_0$, $\mathtt{i}_0$, and $\mathtt{j}_0$) with a guard set consisting of $\mathtt{a}_0$. Again, there is no best match in the entries in $\mathtt{a}_1$ (i.e., neither $\mathtt{c}_0$ nor $\mathtt{i}_0$ nor $\mathtt{j}_0$ contain w) but the guard $\mathtt{a}_0$ contains w, and, since we are at level 0, we are done. Therefore, w is in $\mathtt{a}_0$.

2. The search for x first examines the root, at level 3, and determines $\mathtt{a}_1$ to be the best match. However, $\mathtt{a}_1$ is a guard and thus it finds the smallest nonguard region containing $\mathtt{a}_1$ which is $\mathtt{b}_2$. The search continues at the directory node pointed at by $\mathtt{b}_2$ at level 2 (i.e., containing $\mathtt{b}_1$, $\mathtt{e}_0$, and $\mathtt{e}_1$) with a guard set consisting of $\mathtt{a}_0$ and $\mathtt{a}_1$. The best match in this node is $\mathtt{b}_1$. However, the level 1 guard in the guard set is $\mathtt{a}_1$ and its

corresponding region is a better match. The search continues in the directory node pointed at by a_1 at level 1 (i.e., containing c_0, i_0, and j_0) with a guard set consisting of a_0. There is no best match in this node but the guard a_0 contains x and, since we are at level 0, we are done. Therefore, x is in a_0.

3. The search for y first examines the root, at level 3, and determines a_0 to be the best match. However, a_0 is a guard and thus it finds the smallest nonguard region containing a_0 which is b_2. The search continues at the directory node pointed at by b_2 at level 2 (i.e., containing b_1, e_0, and e_1) with a guard set consisting of a_0. The best match in this node is b_1. There is no level 1 guard in the guard set and the search continues at the directory node pointed at by b_1 at level 1 (i.e., containing b_0 and n_0) with a guard set consisting of a_0 (we ignore e_0 as it does not contain y). There is no best match in b_1 but the guard a_0 contains y and since we are at level 0, we are done. Therefore, y is in a_0.

4. The only difference in searching for z from searching for y is that the search is continued at the directory node pointed at by b_1 (i.e., containing b_0 and n_0) with a guard set consisting of e_0 as it provides the better match (i.e., both e_0 and a_0 contain z but a_0 contains e_0). Therefore, z is in e_0.

One of the main reasons for the attractiveness of the BV-tree is its guaranteed performance for executing exact-match queries, its storage requirements in terms of the maximum number of point pages and directory nodes, and the maximum level of the directory structure that are necessary for a particular volume of data given the fanout of each directory node. The fanout is equal to the quotient of the number of bits in each node (i.e., page size) and the number of bits need for each directory entry. The reason that these bounds exist for the BV-tree while they may not exist for other representations such as the hB-tree and the BANG file is that in the BV-tree there is no need to worry about the multiple posting problem. Thus each region is pointed at just once in the BV-tree.

The BV-tree performance guarantees are based in part on the assumption that each point page is at least one-third full [45]. This is justified by appealing to the proof given for the hB-tree's satisfaction of this property [89] (and likewise for the BANG file). The presence of the guards complicates the use of the assumption that each directory node is at least one-third full (which holds for the hB-tree and the BANG file). For example, while illustrating the mechanics of the BV-tree insertion process, we saw an instance of a directory node split operation where the nodes resulting from the split were not at least one-third full (recall Figure 66e). This situation arose because of the presence of guards whose number was taken into account in the node occupancy when determining if a directory node overflows. However, the guards are not necessarily redistributed among the nodes resulting from the split. Instead, they are often promoted thereby precipitating the overflow of directory nodes at shallower levels of the directory hierarchy.

At this point, let us re-examine the situation that arise when a directory node overflows. As mentioned earlier, a directory node at level v (assuming that the deepest directory node is at level 1), may have a maximum of $v-1$ guards for each entry in it which is not a guard. An overflowing directory node a at level v has $F+1$ entries. Let a be split into directory nodes b and c. Now, suppose that after the split, entry d in a serves as a guard for directory node c and that d has $v-1$ guards in a. Therefore, d will be promoted as a guard to the directory node e at level $v+1$ which serves as the father of a. In addition, we have to promote all of the $v-1$ guards of d in a. This means that we only have $F+1-v)$ entries to distribute into b and c which must each be at least one-third full (i.e., contain at least $F/3$ entries each). Thus we must be careful in our choice of F. In particular, if $F < 3v-3$, then it is not possible to guarantee the one-third full condition (e.g., when the promoted node has $v-1$ guards so that a total of v nodes get promoted to level $v+1$).

Clearly, choosing F to be large is one way to increase the likelihood of avoiding such problems. An alternative method of avoiding this problem altogether is to permit the directory nodes to be of variable size and only take the number of unpromoted entries into account when determining how to split an overflowing node. In this case, we can avoid most problems by increasing the capacity of a directory node at (decomposition) level v to be $F \cdot v$ thereby guaranteeing enough space to store the maximum number of guards (i.e., $v-1$) for each unpromoted entry in the directory node (see Exercises 12– 14). Interestingly, this has little effect on the overall size of the directory since so few of the directory nodes are at the maximum level [45] (see Exercise 15). An added benefit of permitting the directory nodes to vary in size and only taking the number of unpromoted entries into account is that now the execution time for the search query is the same as it would be

for a balanced tree with the same number of point pages (assuming the same fullness percentage guarantee).

The number of guards that are present also has a direct effect on the number of point pages and directory nodes in the BV-tree of level v. It can be shown that regardless of the number of guards present, given the same fanout value F for both point pages and directory nodes, the ratio of directory nodes to point pages is $1/F$ [45] (see Exercise 11). Assuming that the deepest directory nodes are at level 1, the maximum number of point pages in a BV-tree of height v is F^v and arises when there are no guards (see Exercise 8 for a related result). This is the case when each directory node split results in two disjoint halves as no guards need to be promoted. The minimum number of point pages for a BV-tree of height v and fanout F is $F^v/v!$ where $F \gg v$ and arises when each directory node at level i contains $i-1$ guards for each unpromoted entry in the node (see Exercises 9 and 10). From a practical standpoint, in order to accommodate the same number of point pages, for a fanout value of 24, a BV-tree of height 3 will have to grow to height 4. Similarly, BV-trees of heights 4 and 5 will have to grow to heights 6 and 10, respectively. For a fanout value of 120, the effect is even smaller. In particular, for this fanout value, BV-trees of heights 4 and 6 will have to grow to heights 5 and between 8 and 9, respectively. Thus when the data pages are of size 1024 bytes, a 200GB file will contain about 200M pages. These 200M pages will fill up a BV-tree of height 4, and in the worst case will require the BV-tree to grow by one level to be of height 5.

The implementation of the searching process demonstrates the utility of the innovation in the design of the BV-tree. The directory is stored economically by reducing the number of nodes necessary due to the avoidance of needless splitting of regions. Recall that this was achieved by decoupling the directory hierarchy from the decomposition hierarchy[37]. Nevertheless, in order to perform searching efficiently (i.e., to avoid descending nodes needlessly as is the case in the R-tree where the space containing the desired data can be covered by several nodes due to the nondisjoint decomposition of the underlying space), the regions must be split, or at least examined in a way that simulates the result of the split, thereby posing apparently contradictory requirements on the data structure. In the BV-tree, this is avoided by the presence of the guards which are associated with the relevant nodes of the directory hierarchy. In fact, the guards are always associated with the directory node that corresponds to the nearest common ancestor of the directory nodes with whom they span some common portion of the underlying space. This is very similar to the notion of fractional cascading [19, 20] (see Section **??** of Chapter **??**) which is used in many computational geometry applications.

Although up to now, we have looked at the BV-tree as an attempt to overcome the problems associated with variants of the k-d-B-tree, the BV-tree can also be viewed as a special case of the R-tree which tries to overcome the shortcomings of the R-tree. In this case, the BV-tree is not used in its full generality since in order for the analogy to the R-tree to hold, all regions must be hyper-rectangles. Of course, although these analogies and comparisons are matters of interpretation, closer scrutiny does reveal some interesting insights.

The main drawback of the k-d-B-tree and its variants is the fact that region splits may be propagated downwards. This is a direct result of the fact that regions must be split into disjoint subregions where one subregion cannot contain another subregion. The hB-tree and the BANG file attempt to overcome this problem but they still suffer from the multiple posting problem. In this case, instead of splitting point pages into several pieces, directory nodes are referenced several times. In terms of efficiency of the search process, the multiple posting problem is analogous to the multiple coverage problem of the R-tree. In particular, the multiple coverage problem of the R-tree is that the area containing a specific point may be spanned by several R-tree nodes since the decomposition of the underlying space is not disjoint. Thus just because a point was not found in the search of one path in the tree, does not mean that it would not be found in the search of another path in the tree. This makes search in an R-tree somewhat inefficient.

At a first glance, the BV-tree suffers from the same multiple coverage problem as the R-tree. This is true when we examine the directory hierarchy the the BV-tree. However, the fact that the search process in the BV-tree carries the guards with it as the tree is descended ensures that only one path is followed in any search, thereby compensating for the multiple coverage. Notice that what is really taking place is that the search proceeds by levels in the decomposition hierarchy, even though it may appear to be backtracking in the directory hierarchy. For example, when searching for point `x` in the BV-tree of Figure 67a, we immediately make use

[37] See [26] for a related approach to triangular decompositions of surfaces such as terrain data.

of the guards $\mathtt{a}_0$ and $\mathtt{a}_1$ in our search of region $\mathtt{b}_2$. We see that these three regions have quite a bit in common. However, as we descend the BV-tree, we find that some of the regions are eliminated from consideration resulting in the pursuit of just one path. One way to speed the searches in the BV-tree is to organize each directory node into a collection of trees corresponding to the containment hierarchy represented by the node.

The use of the containment hierarchy in the BV-tree can be thought of as leading to a special case of an R-tree in the sense that the containment hierarchy serves as a restriction on the type of bounding rectangles that can be used. In particular, an R-tree could be built subject to the constraint that any pair of bounding rectangles of two sons a and b of node r must be either disjoint or one son is completely contained in the other son (i.e., a is in b or b is in a). In addition, if we were to build an R-tree using the BV-tree rules, then we would have to modify the rules as to the insertion of a point that does not lie in the areas spanned by any of the existing bounding rectangles. In particular, we must make sure that the expanded region does not violate the containment hierarchy requirements. Overflow must also be handled somewhat differently to ensure the promotion of guards so that we can avoid the multiple coverage problem. Of course, once this is done, the structure no longer satisfies the property that all leaf nodes are at the same level. Thus the result is somewhat like the S-tree [4] (see Section **??** of Chapter **??**) which is a variant of the R-tree designed to deal with skewed data.

Exercises

1. Suppose that a region r at decomposition level v in a BV-tree serves as a guard g for a region at level $v-1$, and the directory node corresponding to r overflows. How would you handle the situation? In particular, what are the ramifications of splitting r into two disjoint regions in contrast to one where one region contains the other?

2. Prove that all exact-match searches in a BV-tree have the same length and always visit every level in the decomposition hierarchy.

3. Consider an exact-match search for a point p in a BV-tree. This requires traversing the directory hierarchy of the tree while applying a process that finds the best match entry in a particular directory node a. When processing the root node, if the best match e_r corresponding to region r is a guard, then we need to find the smallest region s containing r whose corresponding entry e_s is not a guard. Prove that e_s exists.

4. Continuing Exercise 3, prove that any matching guards enclosed by region s also enclose r.

5. Let us try to generalize Exercise 3 to nodes other than the root. In particular, suppose that during the search for a point p in a BV-tree we have reached a nonroot node a at level v. Note, that this means that the nonguard entries in a correspond to regions at level $v-1$ in the decomposition hierarchy. Let e_r corresponding to region r be the best match among the entries in a plus the level $v-1$ entry in the guard set brought down from higher levels. If e_r is a guard in a, then we need to find a region s containing r whose corresponding entry e_s is either a nonguard entry in a or is the level $v-1$ member of the guard set (to see that e_s can be a nonguard consider the search for $\mathtt{z}$ in Figure 67 where the best match in the directory node pointed at by $\mathtt{b}_2$ is $\mathtt{e}_0$ while being enclosed by $\mathtt{b}_1$ which is not a guard). Prove that e_s exists.

6. Suppose that you modify the exact-match search procedure in a BV-tree so that when a search of a directory node a at level v finds that the best match is a guard g at level $v-2$, then the search is continued in the level $v-2$ directory node corresponding to g. This is instead of the regular search process which leads to the following actions:

    1. Finding the best match d in the entries in a that are not guards and applying the same process to the level $v-2$ entries in d obtaining e as the best match.
    2. Checking if e is a better match than the guard g at level $v-2$ (if one exists) in the guard set that was brought down from the previous search (i.e, from level v).

In other words, if this modification is valid, then it would appear that there is no need to do the search at the level $v-1$ directory node. once we found that the best match at level v is a guard. Prove or disprove the validity of this modification. If it is not valid, then construct a counter-example using the BV-tree of Figure 67.

7. How would you deal with deletion in a BV-tree

8. Prove that the maximum number of directory nodes in a BV-tree of height v and fanout F is approximately F^{v-1} for $F \gg 1$.

9. Prove that the minimum number of point pages in a BV-tree of height v and fanout F is approximately $F^v/v!$ for $F \gg v$.

10. Prove that the maximum number of directory nodes in the BV-tree of height v and fanout F for which the number of point pages is a minimum is approximately $F^{v-1}/v!$ for $F \gg v$.

11. Prove that the ratio of directory nodes to point pages in a BV-tree is $1/F$ regardless of whether a minimum or maximum number of guards are present.

12. Suppose that nodes in the BV-tree are of varying size. In particular, assume that a node at level i of a BV-tree with fanout F is large enough to contain F unpromoted entries and $F \cdot (i-1)$ guards (i.e., unpromoted entries). Show that the total number of point pages we can now have in the worst-case of a BV-tree of height v and fanout F is $\approx F^v$ for $F \gg 1$.

13. Using the variable BV-tree node size assumption of Exercise 12, show that the total number of directory nodes we can now have in the worst-case of a BV-tree of height v and fanout F is $\approx F^{v-1}$ for $F \gg 1$.

14. Using the variable BV-tree node size assumption of Exercises 12 and 13, Prove that the ratio of directory nodes to point pages in a BV-tree is $1/F$ when the maximum number of guards are present.

15. Suppose that you use the variable BV-tree node size assumption of Exercises 12 and 13. This means that the directory nodes are not all the same size. Assuming that each node at level v originally required B bytes, the variable node size means that each node will be of size Bv bytes. Show that this modification leads to a very nominal growth in the number of bytes needed to represent the index nodes. In particular, given a BV-tree of height v and fanout F, prove that the storage requirements for the index nodes is approximately BF^{v-1} — that is, B times the number of directory nodes as derived in Exercise 13. Thus the increased size of the nodes close to the root has a negligible effect on the total size of the directory in that when we used B bytes for each directory node we also obtain an index size of BF^{v-1} in the best case.

7.1.6 Static Methods

The LSD tree, k-d-B-tree, hB-tree, k-d-B-trie, BANG file, and BV-tree are all dynamic structures where data is inserted at the leaf nodes and splits are propagated upwards. Thus they are often characterized as bottom-up data structures. A pair of related data structures are the VAMSplit k-d tree (denoting *variance approximate median split*) [143] and the VAMSplit R-tree [143]. They differ from the LSD tree, k-d-B-tree, hB-tree, and BANG file in that they are static structures where the partitions (i.e., splits) are made in a top-down manner on the basis of knowledge of the entire data set.

The VAMSplit k-d tree [143] is really a bucket variant of an adaptive k-d tree in the sense that each leaf node corresponds to a disk bucket or block of capacity b and thus we only split a set of elements if its cardinality exceeds b. Notice that the tree need not be complete. However, the depths of any two leaf nodes differ by at most one. The partitioning axis is chosen corresponding to the dimension with the maximum variance. The partitioning position is chosen with the goal that the leaf nodes (i.e., buckets) are as full as possible. Thus

the partitioning position is not necessarily the median value along the partitioning axis, although it is usually within a relatively close range of the median (hence the use of the qualifier *approximate median split*) [143].

In particular, assuming a set of N nodes, if $N \leq 2b$, then we choose the median value as the partitioning position so that the left son has $\lfloor N/2 \rfloor$ elements while the right son has $N - \lfloor N/2 \rfloor$ elements. Otherwise, $N > 2b$, and we choose a partitioning position so that the left son contains mb elements where mb is the largest multiple of b which is less than $N/2$, while the right son contains the remaining elements. Such an action will minimize the number of leaf nodes in the final k-d tree by maximizing the number of completely full nodes so that at most two will not be full (see Exercise 3).

As an example, suppose that we have a set with 22 elements and bucket capacity of 5 for which we want to build a VAMSplit k-d tree. There are many ways to partition this set. Below, we only consider the way in which the partitioning positions are chosen thereby assuming that the partitioning axes have already been determined. Figure 68a shows the size of the subtrees in the resulting k-d tree when the partitioning position is the median with 6 leaf nodes (i.e., buckets), while Figure 68b shows the resulting VAMSplit k-d tree when the partitioning position is chosen so as to minimize the total number of leaf nodes which is 5 here. Note that although the difference in the number of leaf nodes is just one in this case, examples can be constructed with more elements where the difference in the number of leaf nodes is considerably higher.

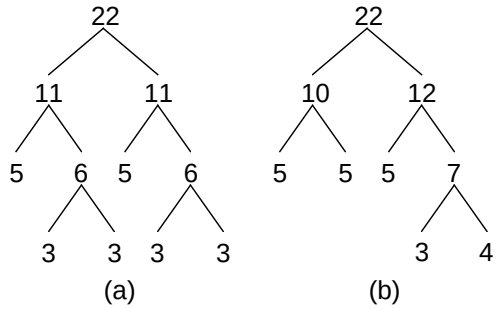


Figure 68: k-d trees with 22 elements and a bucket capacity of 5 which are built by using a partitioning position (a) at the median of the elements in the subtree, and (b) at the position which minimizes the total number of leaf nodes (i.e., a VAMSplit k-d tree with no aggregation of the internal nodes). The nonleaf nodes indicate the number of elements in the corresponding subtree.

Once the partitioning positions have been chosen, the internal nodes of the tree are aggregated into buckets of capacity c proceeding from the root downwards in a breadth-first traversal of the subtrees until enough nodes have been accumulated to fill the bucket (i.e., c)This process is applied recursively to the rest of the tree until encountering the leaf nodes. The similarity to a top-down LSD tree and and a top-down k-d-B-tree should be clear. The search performance of the VAMSplit k-d tree is enhanced by associating with each node i in the VAMSplit k-d tree a minimum bounding box of the part of the underlying space that is spanned by the internal nodes which have been merged into i. The result is termed a *VAMSplit R-tree* [143]. However, unlike the real R-tree, minimization of overlap or coverage plays no role in deciding which internal nodes should be aggregated. This decision is based purely on the k-d tree splits. Thus we see that the VAMSplit R-tree is related to the VAMSplit k-d tree in the same way that the R^+-tree [36, 127, 129] is related to the k-d-B-tree (see Section **??** of Chapter **??**). In fact, a more appropriate name for the VAMSplit R-tree is a VAMSplit R^+-tree.

Exercises

1. Assuming a bucket size of 5, can you come up with a simpler example than that given in Figure 68 to illustrate the advantage of not using a partitioning position at the median of the set of elements in an adaptive k-d tree such as the VAMSplit k-d tree?

2. Give an algorithm to build a VAMSplit k-d tree.

3. Prove that the number of nonfull leaf nodes in a VAMSplit k-d tree is at most two.

4. Give an algorithm to build a VAMSplit R-tree.

7.2 Grid Directory Methods

There are a number of methods of decomposing the underlying space into a grid of cells termed *grid cells*. The contents of the grid cells are stored in pages (termed *buckets*) in secondary memory. Some of the methods (e.g., grid file and EXCELL) permit a bucket to contain the contents of several grid cells, but do not permit a grid cell to be associated with more than one bucket. Other methods (e.g., linear hashing and spiral hashing) do not permit a bucket to contain the contents of more than one grid cell, but do permit a grid cell to be associated with more than one bucket. When more than one grid cell is associated with a bucket, then the union of these grid cells must form a d-dimensional hyper-rectangle. The actual buckets associated with the grid cells are usually accessed with the aid of a directory in the form of an array (termed *grid directory*) which can change as the data volume grows or contracts.

Below, we focus on what to do if the data volume grows to such an extent that either one of the grid cells or buckets is too full. There are a number of possible solutions. When a bucket is too full, we just split it. When a grid cell is too full, we usually need to make a new grid partition. In this case, the most extreme solution is to make a new grid by refining the entire grid of g d-dimensional grid cells by doubling it along all keys (i.e., halving the width of each side of a grid cell) thereby resulting in $2^d \cdot g$ grid cells. This is quite wasteful of space and is not discussed further here.

Whatever action we take, we must maintain the grid while also resolving the issue of the overflowing grid cell (usually by splitting it). This can be achieved by refining the partition along just one of the keys through the introduction of of one additional partition. The drawback of such an action is that the grid cells are no longer equal-sized. This means that given a point, we cannot determine the grid cell in which it is contained without the aid of an additional data structure (termed a *linear scale* and discussed briefly in Section 1) to indicate the positions of the partitions. If there are originally $\prod_{i=1}^{d} g_i$ grid cells with $g_i - 1$ partition positions along key i, then, without loss of generality, refining key d results in $(g_d + 1) \cdot \prod_{i=1}^{d-1} g_i$ grid cells. This is the basis of the grid file [99, 57, 58].

We can avoid the need for the linear scales by uniformly partitioning all grid cells in the same way rather than just the cells in the $(d-1)$-dimensional hyperplane that passes through the overflowing grid cell. In essence, what we do is halve the width of just one of the sides of each grid cell thereby doubling the granularity of the decomposition along one of the axes. This results in doubling the total number of grid cells to yield $2 \cdot g$ grid cells. This is the basis of the EXCELL [131] method.

An alternative approach is to order the grid cells according to one of a subset of the one-dimensional orderings of the underlying space described in Section 6 and use linear hashing (recall Section **??** of Chapter **??**) to resolve an overflowing grid cell. In this case, we start with a grid of equal-sized grid cells (initially containing just one grid cell) which are split into two halves in succession in a consistent manner (i.e., along the same key) according to the particular one-dimensional ordering that is used until all grid cells have been split at which point the process is restarted with a split across the same or another key. The drawback of this method is that the grid cell that is split is not necessarily the one that has become too full. This is resolved by making use of what are termed *overflow buckets*.

It is important to observe that for linear hashing the grid directory is a one-dimensional array instead of a d-dimensional array as for the grid file and EXCELL. This is because the result of the hashing function is just an address in one dimension although the underlying space is partitioned into sets of ranges of values which for most variants of linear hashing (with the notable exception of spiral hashing) correspond to grid cells in d dimensions. For most variants of linear hashing, the ranges have a limited set of possible sizes. In fact, if there is a one-to-one correlation between the numbering of the grid cells and the bucket labels, then there is no need for a grid directory in linear hashing.

The rest of this section is organized as follows. Section 7.2.1 describes the grid file. Section 7.2.2 presents EXCELL. Section 7.2.3 reviews linear hashing and explains how it can be adapted to handle multidimensional point data. Section 7.2.4 discusses some variants of linear hashing that address the drawback that the grid cell that has been split most recently is not necessarily the one that is full. This discussion also includes an explanation of how to adapt the related method of spiral hashing to multidimensional point data. Section 7.2.5 contains a brief comparison of the various bucket methods that make use of a grid directory.

7.2.1 Grid File

The *grid file* of Nievergelt, Hinterberger, and Sevcik [99, 57, 58] is a variation of the fixed-grid method, which relaxes the requirement that grid subdivision lines be equidistant. Its goal is to retrieve records with at most two disk accesses, and to handle range queries efficiently. This is done by using a grid directory in the form of an array to the grid cells. All records in one grid cell are stored in the same bucket. However, several grid cells can share a bucket as long as the union of these grid cells forms a d-dimensional hyper-rectangle in the space of the data. Although the regions of the buckets are piecewise disjoint, together they span the entire underlying space.

The purpose of the grid directory is to maintain a dynamic correspondence between the grid cells and the buckets. The grid directory consists of two parts. The first is a dynamic d-dimensional array, containing one entry for each grid cell. The values of the elements are pointers to the relevant buckets. Usually buckets will have a capacity of 10 to 1000 points. Thus, the entry in the grid directory is small in comparison to a bucket. We are not concerned with how the points are organized within a bucket (e.g., linked list, tree, etc.). The grid directory is usually stored on disk as it may be quite large especially when d is large.

The second part of the grid directory is a set of d one-dimensional arrays called *linear scales*. These scales define a partition of the domain of each key. They enable access to the appropriate grid cells by aiding the computation of their address based on the value of the relevant keys. The linear scales are kept in core. It should be noted that the linear scales are useful in guiding a range query, by indicating the grid cells that overlap the query range. The linear scales can also be implemented using binary trees (see Sections 7.2.3 and 7.2.4 as well as Exercise 1).

Thus, we see that the two goals of the grid file are met. Any record is retrieved with two disk accesses: one disk access for the grid cell and one for the bucket. Range queries are efficient, although since the sizes of the grid cells (i.e., the ranges of their intervals) are not related to the query range, it is difficult to analyze the amount of time necessary to execute a range query.

As an example, consider Figure 69, which shows the grid file representation for the data in Figure 1. Once again, we adopt the convention that a grid cell is open with respect to its upper and right boundaries and closed with respect to its lower and left boundaries. The bucket capacity is 2 records. There are $d = 2$ different keys. The grid directory consists of 9 grid cells and 6 buckets labeled `A-F`. We refer to grid cells as if they are array elements: grid cell (i, j) is the element in column i (starting at the left with column 1) and row j (starting at the bottom with row 1) of the grid directory.

Grid cells `(1,3)`, `(2,2)`, and `(3,3)` are empty; however, they do share buckets with other grid cells. In particular, grid cell `(1,3)` shares bucket `B` with grid cells `(2,3)` and `(3,3)`, while grid cells `(2,2)` and `(3,2)` share bucket `E`. The sharing is indicated by the broken lines. Figure 70 contains the linear scales for the two keys (i.e., the x and y coordinate values). For example, executing a point query with $x = 80$ and $y = 62$ causes the access of the bucket associated with the grid cell in row `2` and column `3` of the grid directory of Figure 69.

The grid file is attractive, in part, because of its graceful growth as more and more records are inserted. As the buckets overflow, we apply a splitting process, which results in the creation of new buckets and a movement of records. To understand the splitting process, let us examine more closely how the grid file copes with a sequence of insertion operations. Again, we assume a bucket capacity of 2 and observe how a grid file is constructed for the records of Figure 1 in the order in which they appear there: `Chicago`, `Mobile`, `Toronto`,

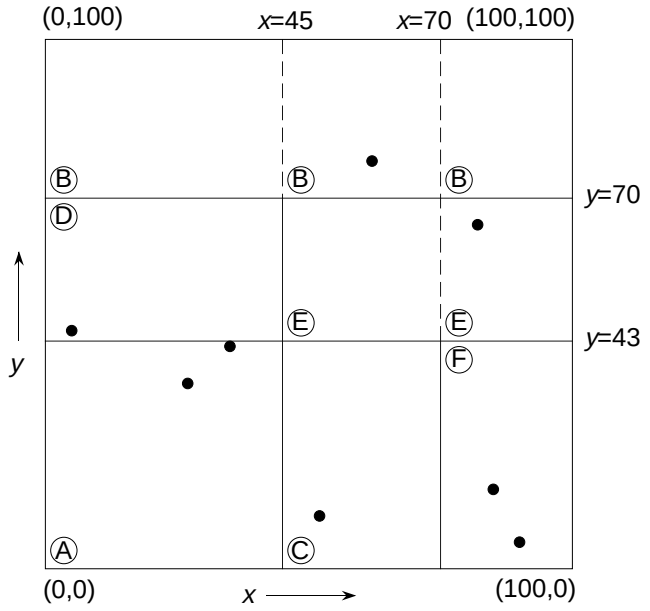


Figure 69: Grid file partition for the data corresponding to Figure 1.

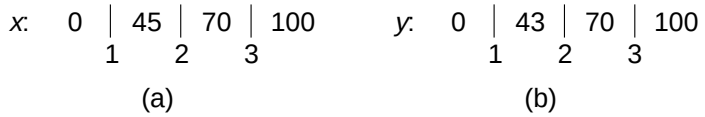


Figure 70: Linear scales for (a) x and (b) y corresponding to Figure 69.

`Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`.

The insertion of `Chicago` and `Mobile` results in bucket `A` being full. Insertion of `Toronto` leads to an overflow of bucket `A` causing a split. We arbitrarily split along the y axis at $y = 70$ and modify the linear scale for key y accordingly. `Toronto` is inserted in `B`, the newly allocated bucket (see Figure 71a).

Next, we try to insert `Buffalo` and find that the bucket in which it belongs (i.e., `A`) is full. We split the x axis at $x = 45$ and modify the linear scale for key x. This results in the insertion of `Buffalo` in bucket `C` and the movement of `Mobile` from bucket `A` to bucket `C` (see Figure 71b). Note that as a result of this split, both grid cells `(1,2)` and `(2,2)` share bucket `B`, although grid cell `(1,2)` is empty. Alternatively, we could have marked grid cell `(1,2)` as empty when the x axis was split. This has the disadvantage that should we later wish to insert a record in grid cell `(1,2)` we would have to either allocate a new bucket or search for a neighboring bucket that is not full and whose grid cells form a convex region with grid cell `(1,2)`.

The insertion of `Denver` proceeds smoothly. It is placed in bucket `A`. `Omaha` also belongs in bucket `A` which means that it must be split again. We split the y axis at $y = 43$ and modify the linear scale for key y. In addition, we create a new bucket, `D`, to which `Denver` is moved while `Omaha` and `Chicago` remain in bucket `A` (see Figure 71c). Note that as a result of this split both grid cells `(2,1)` and `(2,2)` share bucket `C`, contributing `Buffalo` and `Mobile`, respectively.

Attempting to insert `Atlanta` finds it belonging in bucket `C`, which is full yet grid cell `(2,1)` is not full. This leads to splitting bucket `C` and the creation of bucket `E`. Bucket `C` now contains `Mobile` and `Atlanta` and corresponds to grid cell `(2,1)` while bucket `E` now contains `Buffalo` and corresponds to grid cell `(2,2)` (see Figure 71d). Note that we did not have to partition the grid in this case. Thus, no change needs to be made to the linear scales; however, the grid directory must be updated to reflect the association of grid cell `(2,2)` with bucket `E` instead of bucket `C`.

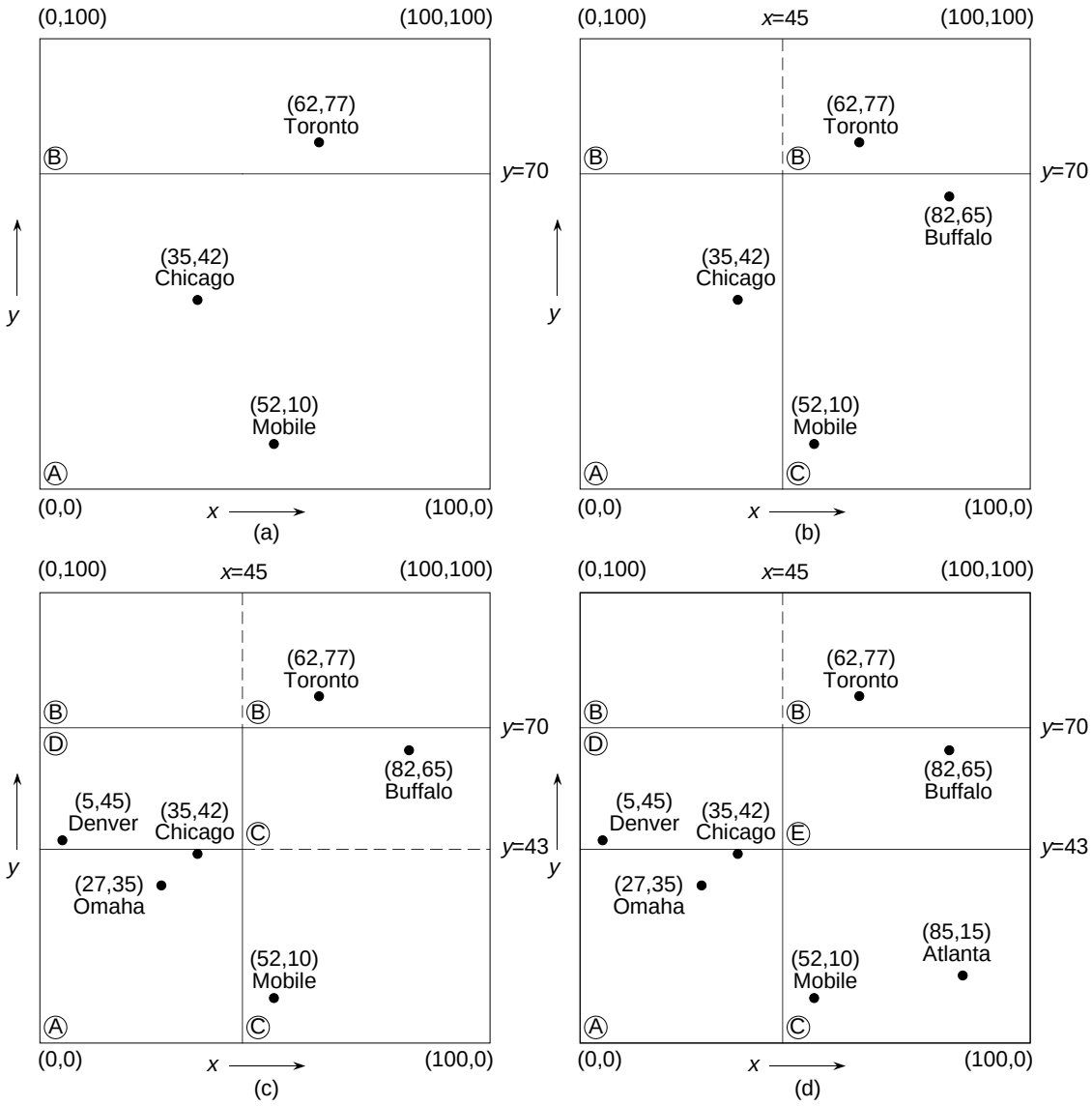


Figure 71: Sequence of partial grid partitions demonstrating the result of the insertion of (a) Chicago, Mobile, and Toronto, (b) Buffalo, (c) Denver and Omaha, and (d) Atlanta.

Finally, insertion of `Miami` finds it belonging to bucket `C`, which is full. We split the x axis at $x = 70$ and modify the linear scale for key x. A new bucket, `F`, is created to which `Atlanta` and `Miami` are moved (see Figure 69). Once again, we observe that as a result of this split, grid cells `(2,3)` and `(3,3)` share bucket `B`, although grid cell `(3,3)` is empty. Similarly, grid cells `(2,2)` and `(3,2)` share bucket `E`, although grid cell `(2,2)` is empty.

At this point, we are ready to elaborate further on the splitting process. From the above we see that two types of bucket splits are possible. The first, and most common, arises when several grid cells share a bucket that has overflowed (e.g., the transition between Figures 71c and 71d upon the insertion of `Atlanta`). In this case, we merely need to allocate a new bucket and adjust the mapping between grid cells and buckets.

The second type of a split arises when we must refine a grid partition. It is triggered by an overflowing bucket, all of whose records lie in a single grid cell (e.g., the overflow of bucket `A` upon insertion of `Toronto`

in Figure 71a). In this case, we have a choice with respect to the dimension (i.e., axis) and the location of the splitting point (i.e., we do not need to split at the midpoint of an interval). Without any external knowledge or motivation, a reasonable splitting policy is one that cycles through the various keys (e.g., first split on key x, then key y, key x, etc., as was done in Figure 71) and uses interval midpoints. This is the approach used in the grid file implementation [99].

An alternative splitting policy is an adaptive one favoring one key over others. This is akin to a favored key in an inverted file. It results in an increase in the precision of answers to partially specified queries where the favored key is specified. Such a policy is triggered by keeping track of the most frequently queried key (in the case of partial match queries), and by monitoring dynamic file content, thereby increasing the granularity of the key scales. Splitting at locations other than interval midpoints is also an adaptive policy.

The grid refinement operation is common at the initial stage of constructing a grid file. However, as the grid file grows it becomes relatively rare in comparison with the overflowing bucket that is shared among several grid cells. Nevertheless, the frequency of grid refinement can be reduced by varying the grid directory as follows. Implement the grid directory as a d-dimensional array whose size in each dimension is determined by the shortest interval in each linear scale (i.e., if the linear scale for key x spans the range 0 through 64 and the shortest interval is 4, then the grid directory has 16 entries for dimension x).

This variation is a multidimensional counterpart of the directory used in extendible hashing [33] and is the basis of EXCELL [131] (see Section 7.2.2). Its advantage is that a refinement of the grid partition will only cause a change in the structure of the directory if the shortest interval is split, in which case the grid directory will double in size. Such a representation anticipates small structural updates and replaces them by a large one. It is fine as long as the data is uniformly distributed. Otherwise, many empty grid cells will arise.

The counterpart of splitting is merging. There are two possible instances where merging is appropriate: bucket merging and directory merging. Bucket merging, the more common of the two, arises when a bucket is empty or nearly empty. Bucket merging policy is influenced by three factors. First, we must decide which bucket pairs are candidates for merging. This decision can be based on a *buddy system* or a *neighbor system*.

In a *buddy system* [72], each bucket, say X, can be merged with exactly one bucket, say B_i, in each of the d dimensions. Ideally, the chosen bucket, say B_j, should have the property that at some earlier point it was split to yield buckets X and B_j. We call this buddy the 'true' buddy. For example, consider the grid directory of Figure 72 which contains buckets `A`–`K` and `X`. Assume that when a bucket is split, it is split into two buckets of equal size. Since $d = 2$, the only potential buddies of bucket `X` are `I` and `J`. We can keep track of the 'true' buddies by representing the splitting process as a binary tree, thereby maintaining the buddy relationships.

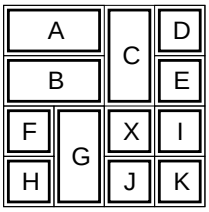


Figure 72: Example grid directory illustrating bucket merging when using the grid file.

In a *neighbor system*, each bucket can be merged with either of its two adjacent neighbors in each of the d dimensions (the resulting bucket region must be convex). For example, in the grid directory of Figure 72, bucket `X` can be merged with any one of its neighbors, buckets `C`, `I`, or `J`. Note that `X` can not be merged with neighbor bucket `G` since the resulting bucket region would not be convex.

The second factor influencing bucket merging policy deals with the ordering among possible choices should there be more than one candidate. In the case of the buddy system, we give priority to the 'true' buddy. Otherwise, this factor is only relevant if it is desired to have a varying granularity among keys. In particular, if the splitting policy favors some keys over others, then the merging policy should not undo it.

The third factor is the merging threshold: when should a candidate pair of buckets actually be merged? It should be clear that the sum of the bucket occupancy percentages for the contents of the merged bucket should not be too large, as otherwise it would soon have to be split. Nievergelt et al. [99] conducted simulation studies showing the average bucket occupancy to be 70% and suggest that this is an appropriate merging threshold for the occupancy of the resulting bucket.

Directory merging arises when all the grid cells in two adjacent cross sections (i.e., slices along one axis) in the grid directory are associated with the same bucket. For example, consider the two-dimensional grid directory of Figure 73, where all grid cells in column `2` are in bucket `C` and all grid cells in column `3` are in bucket `D`. In such a case, if the merging threshold is satisfied, then buckets `C` and `D` can be merged and the linear scales modified to reflect this change.

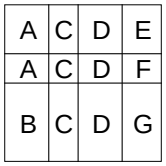


Figure 73: Example grid directory illustrating directory merging when using the grid file.

Generally, directory merging is of little practical interest since even if merging is allowed to occur, it is probable that splitting will soon have to take place. Nevertheless, there are occasions when directory merging is of use. First, directory merging is necessary in the case of a shrinking file. Second, it is appropriate when the granularity of certain keys is being changed to comply with the access frequency of the key. Third, it is a useful technique when attempting to get rid of inactive keys. In such a case, the key could be set to a 'merge only' state. Eventually, the partition will be reduced to one interval and the corresponding dimension in the grid directory can be removed or assigned to another key.

Exercises

1. The linear scales are implemented as one-dimensional arrays. They could also be implemented as binary trees. What would be the advantage of doing so?
2. Implement a database that uses the grid file to organize 2-dimensional data.
3. The grid file is considered to be an instance of the general bucket method of fanout 2 (which also includes the B-tree, EXCELL, and EXHASH) with 0.69 average storage utilization [99]. The *twin grid file* [65, 66] is a representation which makes use of two grid files which has been observed to result in improving the average storage utilization of the grid file to 90%. Give an intuitive explanation of why this is so.
4. How can you improve the performance of the twin grid file discussed in Exercise 3?
5. Calculate the expected size of the grid directory for uniformly distributed data.

7.2.2 EXCELL

The *EXCELL* method of Tamminen [131] is similar in spirit to the grid file in that it also makes use of a grid directory and retrieves all records with at most two disk accesses. The principal difference between them is that grid refinement for the grid file splits only one interval in two and results in the insertion of a $(d-1)$-dimensional cross section. In contrast, a grid refinement for the EXCELL method splits all intervals in two (the partition points are fixed) for the particular dimension and results in doubling the size of the grid directory. This means that all grid cells are of the same size in EXCELL while this is not the case for the grid file.

The result is that the grid directory grows more gradually when the grid file is used, whereas use of EXCELL reduces the need for grid refinement operations at the expense of larger directories, in general, due to a sensitivity to the distribution of the data. However, a large bucket size reduces the effect of nonuniformity unless the data consists entirely of a few clusters. The fact that all grid cells define equal-sized regions (and convex as well) has two important ramifications. First, it means that EXCELL does not require a set of linear scales to access the grid directory and retrieve a record with at most two disk accesses, as is needed for the grid file. Thus, grid directory access operations are considerably faster for EXCELL. Second, it means that the partition points are fixed and are not chosen on the basis of the data, as is the case for the grid file. Therefore, range queries are efficient with an execution time that is proportional to the number of buckets corresponding to the grid cells that comprise the range being searched (i.e., to the size of the range). In contrast, the number of grid cells that comprise a range in the grid file is not proportional to the size of the range.

An example of the EXCELL method is given in Figure 74, which shows the representation for the data in Figure 1. Here the x axis is split before the y axis. Again, the convention is adopted that a rectangle is open with respect to its upper and right boundaries and closed with respect to its lower and left boundaries. The capacity of the bucket is 2 records. There are $d = 2$ different keys. The grid directory is implemented as an array and in this case it consists of 8 grid cells (labeled in the same way as for the grid file) and 6 buckets labeled `A`–`F`. Note that grid cells `(3,2)` and `(4,2)` share bucket `C` while grid cells `(1,2)` and `(2,2)`, despite being empty, share bucket `D`. The sharing is indicated by the broken lines. Furthermore, when a bucket size of 1 is used, the partition of space induced by EXCELL is equivalent to that induced by a PR k-d tree [103], although the two structures differ by virtue of the presence of a directory in the case of EXCELL.

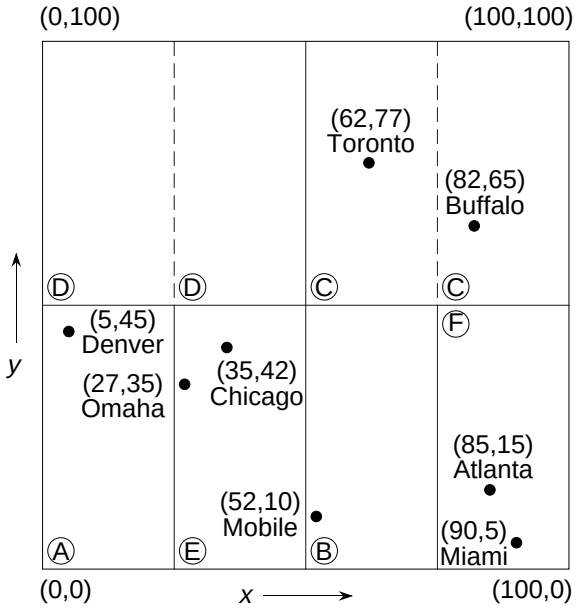


Figure 74: EXCELL representation for the data corresponding to Figure 1 when the x axis is split before the y axis.

As a database represented by the EXCELL method grows, buckets will overflow. This leads to the application of a splitting process, which results in the creation of new buckets and a movement of records. To understand the splitting process, we examine how EXCELL copes with a sequence of insertion operations corresponding to the data of Figure 1. Again, we assume a bucket capacity of 2 and that the records are inserted in the order in which they appear in Figure 1, that is, `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`.

The insertion of `Chicago` and `Mobile` results in bucket `A` being full. Insertion of `Toronto` leads to an overflow of bucket `A`, which compels us to double the directory by splitting along key x. We split bucket `A` and move `Mobile` and `Toronto` to `B`, the newly allocated bucket (see Figure 75a). Next, we insert `Buffalo`

and find that the bucket in which it belongs (i.e., B) is full. This causes us to double the directory by splitting along key y. We now split bucket B and move Toronto and Buffalo to C, the newly allocated bucket (see Figure 75b). Note that bucket A still contains Chicago and overlaps grid cells (1,1) and (1,2).

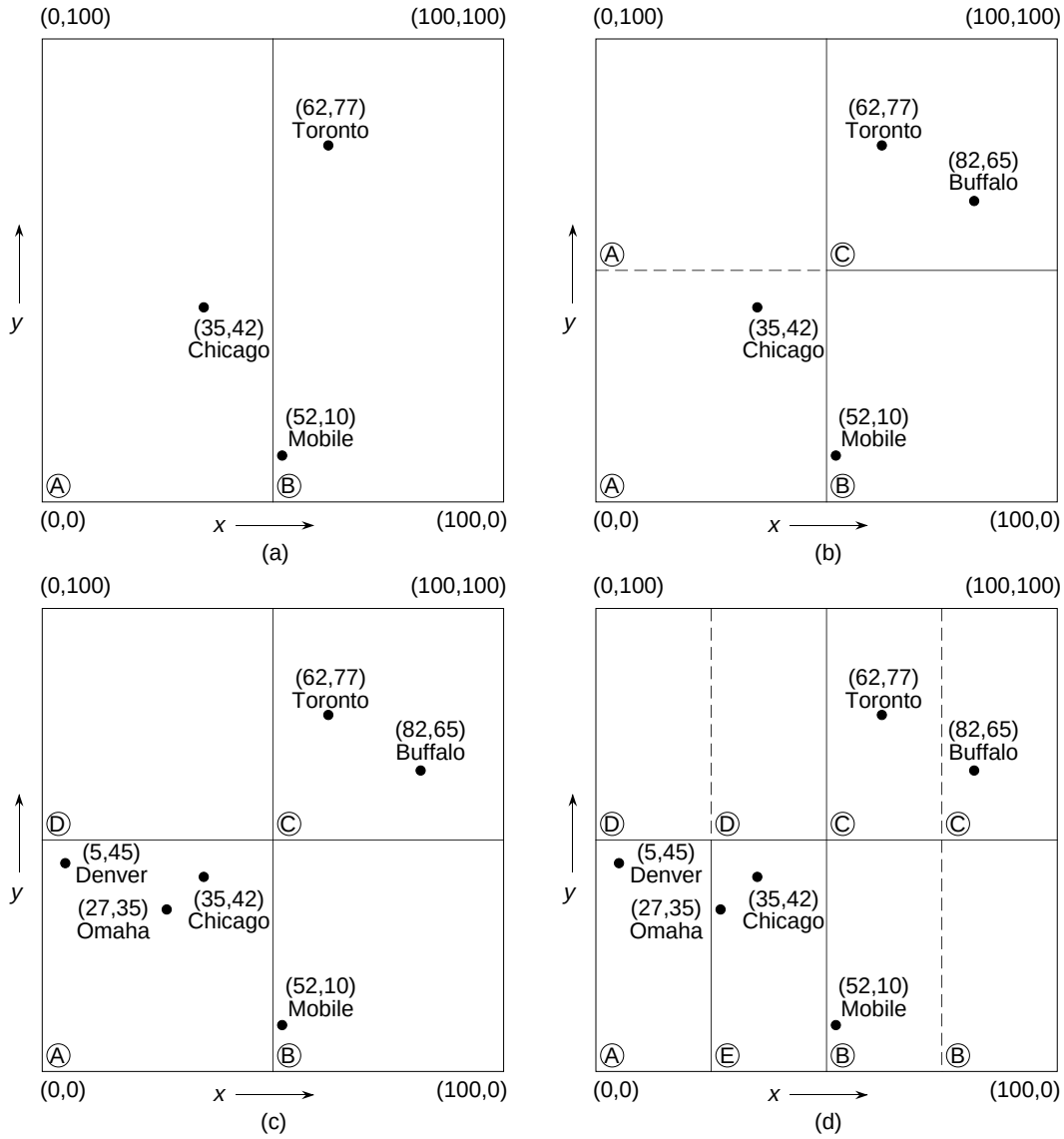


Figure 75: Sequence of partial EXCELL partitions demonstrating the result of the insertion of (a) Toronto, (b) Buffalo, and (c-d) Omaha.

The insertion of Denver proceeds smoothly and it is placed in bucket A. Omaha also belongs in bucket A, which has now overflowed. Since bucket A overlaps grid cells (1,1) and (1,2), we split it thereby allocating a new bucket, D, such that buckets A and D correspond to grid cells (1,1) and (1,2), respectively (see Figure 75c). However, neither Denver, Chicago, nor Omaha can be moved to D, thereby necessitating a directory doubling along key x. We now split bucket A and move Chicago and Omaha to E, the newly allocated bucket (see Figure 75d). Note that buckets B, C, and D retain their contents, except that now each bucket overlaps two grid cells.

Next, Atlanta is inserted in bucket B. Insertion of Miami causes bucket B to overflow. Since B overlaps grid cells (3,1) and (4,1), we split it, thereby allocating a new cell, F, such that buckets B and F correspond to grid cells (3,1) and (4,1), respectively. Atlanta and Miami are moved to F (see Figure 74).

From the discussion, we see that two types of bucket splits are possible. The first, and most common, is when several grid cells share a bucket that has overflowed (e.g., the transition between Figures 75d and 74 caused by the overflow of bucket `B` as `Atlanta` and `Miami` are inserted in sequence). In this case, we allocate a new bucket and adjust the mapping between grid cells and buckets.

The second type of a split causes a doubling of the directory and arises when we must refine a grid partition. It is triggered by an overflowing bucket that is not shared among several grid cells (e.g., the overflow of bucket `A` upon insertion of `Toronto` in Figure 75a). The split occurs along the different keys in a cyclic fashion (i.e., first split along key x, then y, then x, etc.).

For both types of bucket splits, the situation may arise that none of the elements in the overflowing bucket belong to the newly created bucket, with the result that the directory will have to be doubled more than once. This is because the splitting points are fixed for EXCELL. For example, consider bucket `A` in Figure 76a with bucket capacity 2 and containing points `X` and `Y`. Insertion of point `Z` (see Figure 76b) leads to overflow of `A` and causes 3 directory doublings (see Figure 76c). In contrast, the fact that the splitting point is not fixed for the grid file means that it can be chosen so that only one grid refinement is necessary. Thus, we see that the size of the EXCELL grid directory is sensitive to the distribution of the data. However, a large bucket size reduces the effect of nonuniformity unless the data consists entirely of a few clusters.

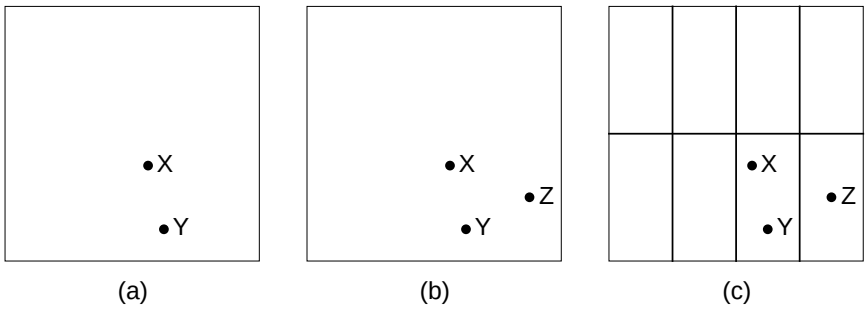


Figure 76: Example showing the need for three directory doubling operations using EXCELL when inserting point Z, assuming a bucket capacity of 2.

The counterpart of splitting is merging. However, it is considerably more limited in scope for EXCELL than for the grid file. Also, it is less likely to arise because EXCELL has been designed primarily for use in geometrical applications where deletion of records is not so prevalent. There are two cases where merging is appropriate.

The first, and most common, is bucket merging, which arises when a pair of buckets is empty or nearly empty. The buckets that are candidates for merging, say X and Y, must be buddies in the sense that at some earlier point in time one of them was split, say X, to yield X and Y (e.g., `A` and `E` in Figure 74 but not `A` and `C`). Once such a bucket pair has been identified, we must see if its elements satisfy the merging threshold. In essence, the sum of the bucket occupancy percentages for the contents of the merged bucket should not be too large, as otherwise it might soon have to be split.

The second instance where merging is possible, is directory merging. It arises when each bucket-buddy pair in the grid directory either meets the merging threshold (e.g., `B` and `C`, and `D` and `E` in Figure 77), or the bucket overlaps more than one grid cell (e.g., bucket `A` overlaps grid cells `(1,2)` and `(2,2)`, and bucket `F` overlaps grid cells `(3,1)` and `(4,1)` in Figure 77). This is quite rare.

Exercises

1. Implement a database that uses EXCELL to organize 2-dimensional data.

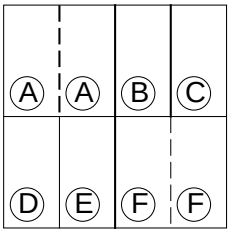


Figure 77: Example demonstrating the possibility of directory merging when using EXCELL.

7.2.3 Linear Hashing

Linear hashing [80, 87] is a technique for avoiding drastic growth in the size of a hash table when it becomes too full (i.e., when a hash bucket overflows). In essence, use of linear hashing enables the hash table to grow by just one hash bucket instead of doubling the number of hash buckets as is the case with some hashing methods such as EXHASH [33], the extendible hashing function that forms the basis of EXCELL [132]. Although linear hashing has been described in Section **??** of Chapter **??**, we review its basic principles here so that our discussion of its adaptation to multidimensional data can be meaningful.

Our adaptation is based on starting with a grid of equal-sized grid cells, ordering the grid cells using a mapping from d dimensions to one dimension (perhaps one of the ones discussed in Section 6), and then, if necessary, splitting the grid cells one-at-a-time in a manner consistent with the chosen order until all grid cells have been split. This results in creating a new grid of equal-sized grid cells, at which time the process is reapplied. Each bucket is associated with just one grid cell.

The key idea behind linear hashing is the decoupling of the decision as to which grid cell is to be split from the identity of the grid cell whose bucket has overflowed. This is a direct result of the pre-determined order in which the grid cells are split. Since we must still deal with the fact that a bucket is too full, we have two types of buckets: primary and overflow. Each grid cell has a different primary bucket. When attempting to insert a data point p whose grid cell c's primary bucket b is full, then p is inserted into an overflow bucket associated with c. Generally, the overflow buckets are chained to the primary buckets. Assuming that there are m grid cells, we use a one-dimensional directory addressed from 0 to $m-1$ to access the primary buckets associated with the grid cells. However, if there is a one-to-one correlation between the numbering of the grid cells and the primary bucket labels (as we will assume in our example), then there is no need for a grid directory.

Formally, we treat the underlying space as having a complete grid T imposed on it, said to be at *level* n, and a partial grid, T', said to be at level $n+1$, where some of the grid cells in T have been split into two equal-sized parts[38]. n denotes the number of total grid partitions that have been applied where a *total grid partition* means that the number of grid cells has doubled in size. Given a complete grid at level n, we have m ($2^n \leq m \leq 2^{n+1}-1$) disjoint grid cells that span the entire underlying space. $2^{n+1}-m$ of the grid cells are d-dimensional hyper-rectangles of size v (both in volume and orientation), while the remaining $2 \cdot (m-2^n)$ grid cells are d-dimensional hyper-rectangles of size $v/2$. The set of grid cells of size v are the ones that have not been split yet.

In order to make this method work, we need an ordering for the grid cells. In addition, we need a way to determine the grid cell associated with each data point. Moreover, if we do not assume a one-to-one correlation between the numbering of the grid cells and the primary bucket labels, then we also need a way to determine the directory element associated with each grid cell so that the relevant bucket can be accessed. Since we are using an ordering, we will need to make use of at least one mapping from d dimensions to one dimension.

In the following, we describe the approach that we use. Identify each grid cell c by the location of the

[38] Note that in this case the term *level* has the same meaning as *depth* for representations that make use of a tree or trie access structure.. As we will see, in many other applications (e.g., the region quadtree as well as the BV-tree in Section 7.1.5), the term *level* has a different meaning in the sense that it indicates the number of aggregation steps that have been performed.

point in the underlying space at the corner of c that is closest to the origin (e.g., the lower-leftmost corner for two-dimensional data with an origin at the lower-leftmost corner of the underlying space) and denote it as $l(c)$. Also, let $u(c)$ be the location of the point in the underlying space at the corner of c that is furthest from the origin (e.g., the upper-rightmost corner for two-dimensional data with an origin at the lower-leftmost corner of the underlying space).

We need two mappings. First, we need a mapping g from d dimensions to one dimension that assigns a unique number to each data point in d dimensions (i.e., an ordering). For this mapping we use one of the orderings discussed in Section 6 that satisfies the properties given below. The ordering is implemented in such a way that the first element in the ordering corresponds to the point closest to the origin.

1. The position in the ordering of each data point p in grid cell c is greater than or equal to that of $l(c)$ (i.e., after) — that is, $g(p) \geq g(l(c))$.
2. The position in the ordering of each data point p in grid cell c is less than or equal to that of $u(c)$ (i.e., before) — that is, $g(p) \leq g(u(c))$.
3. The position in the ordering of any data point not in grid cell c is either before that of $l(c)$ or after that of $u(c)$.

All of these properties are satisfied for both bit interleaving and bit concatenation. Observe that these properties ensure that the ordering also applies to the grid cells. For both bit interleaving and bit concatenation, the ordering also indicates the order in which the grid cells are partitioned. In particular, the partition order is equal to the order from left to right in which the bits of the binary representation of the keys are combined to form the position of the point in the ordering.

Second, we need a mapping h from the one-dimensional representative k of each point p and grid cell c in a complete grid at level n (i.e., $k = g(p)$ and $k = g(l(c))$, respectively) to a number a in the range 0 to $m-1$. The result of this mapping $a = h(k)$ serves as an index in the directory to access the buckets which is a one-dimensional array. The mapping h is such that the $2^{n+1} - m$ grid cells of size v are associated with directory addresses $m - 2^n$ through $2^n - 1$, while the remaining $2 \cdot (m - 2^n)$ grid cells of size $v/2$ are associated with directory addresses 0 through $m - 2^n - 1$ and 2^n through $m - 1$. Each directory element at address a contains the identity of the bucket corresponding to the grid cell associated with a.

Linear hashing implements the function h as two hash functions h_n and h_{n+1}. The function $h_n(k) = k \bmod 2^n$ is used to access the buckets associated with the grid cells at directory addresses $m - 2^n$ through $2^n - 1$, while $h_{n+1}(k) = k \bmod 2^{n+1}$ is used to access the buckets associated with the grid cells at directory addresses 0 through $m - 2^n - 1$ and those at directory addresses 2^n through $m - 1$. Such a file is said to be of level $n, n+1$ so that the grid cells accessed by h_n are at level n while those accessed by h_{n+1} are at level $n + 1$. Note that when $m = 2^n$, no grid cells are accessed by h_{n+1}.

Our adaptation of linear hashing to multidimensional data requires a function g which has the property that all of the data points associated with a particular directory element (i.e., grid cell) are within a given range (recall properties 1–3). Unfortunately, this is not the case when we use the hash function $h_n(k) = k \bmod 2^n$ where $k = g(p)$. In particular, for any binding of k, $h_n(k)$ has the property that all of the points in a given directory element agree in the n least significant bits of k. This is fine for random access; however, it does not support efficient sequential file access (in terms of spatial proximity) since different directory elements and hence different buckets, must be accessed. On the other hand, if h_n would discriminate on the most significant bits of k, then all of the points mapped to a given directory element would be within a given range and hence in the same grid cell thereby satisfying to properties 1–3 of the mapping. Assuming k is of fixed length (i.e., the number of bits is fixed), this can be achieved by redefining the hashing function h_n to be $h_n(k) = reverse(k) \bmod 2^n$. An implementation of linear hashing that satisfies this property is termed *order preserving linear hashing (OPLH).*

In one dimension, OPLH is analogous to a trie. For multidimensional data (e.g., d dimensions), the same effect is obtained by combining the bits from the various keys (e.g., via interleaving, concatenation, etc.),

reversing the result, and then performing a modulo operation. When the combination is based on bit interleaving, the result is analogous to a k-d trie and its behavior for range searching was discussed in some detail in Section 6. In the rest of this section, we use the combination of bit interleaving as the function g with reverse to form a hashing function that we term *reversed bit interleaving* which we use in conjunction with linear hashing.

Reversed bit interleaving results in partitioning the underlying space in a cyclic manner. For example, assuming a bit interleaving order of $yxyxy\ldots$ (i.e., the y coordinate is the most significant), then the first partition splits every grid cell in two using the y coordinate, the second partition splits every grid cell in two using the x coordinate, the third partition splits every grid cell in two using the y coordinate, etc. After each partition, the number of grid cells doubles. After each cycle of partitions (i.e., one partition per key), the number of grid cells grows by a factor of 2^d. Of course, a different partitioning order could be obtained by combining the keys in a different order. For example, we could have defined a similar function termed *reversed bit concatenation* which would complete all partitions along one key before starting a partition on another key. However, reversed bit concatenation has the drawback of favoring some of the keys over the others, and thus we do not discuss its use further here.

It is interesting to note that the use of reversed bit interleaving with linear hashing seems to have been proposed independently by Burkhard [18] (who terms it *shuffle order*), Orenstein [104], and Ouksel and Scheuermann [109] (see also [135]). This combination is applied to range searching by Burkhard [18] and Orenstein and Merrett [105], although they each used different search algorithms.

Now, let us briefly review the mechanics of linear hashing and then give an example of its use with multidimensional data. Recall that a file implemented using linear hashing has both primary and overflow buckets. One primary bucket is associated with each grid cell. The storage utilization factor, τ, is defined to be the ratio of the number of records (i.e., points) in the file to the number of positions available in the existing primary and overflow buckets.

When the storage utilization factor exceeds a predetermined value, say α, then one of the grid cells is split. When grid cell c associated with the directory element at address b is split, its records are rehashed using h_{n+1} and distributed into the buckets of the grid cells associated with directory elements at addresses b and $b + 2^n$. The identity of the directory element corresponding to the next grid cell to be split is maintained by the pointer s that cycles through the values 0 to $2^n - 1$. When s reaches 2^n, all of the grid cells at level n have been split which causes n to be incremented and s to be reset to 0. It is important to observe that a grid cell split does not necessarily occur when a point is inserted that lies in a grid cell whose primary bucket is full nor does the primary bucket of the grid cell that is split have to be full.

As an example of the partitioning that results when g is bit interleaving, consider, again, the database of Figure 1 after the application of the mapping $f(z) = z \div 12.5$ to the values of both the x and y coordinates. The result of this mapping is given in columns 2 and 3 of Figure 78. Next, apply reversed bit interleaving to its keys to yield the mappings given in columns 4 and 5 of the Figure. Two mappings are possible since we can either take the value of the x coordinate as the most significant (column 4 and labeled `CODE`(x)) or the y coordinate (column 5 and labeled `CODE`(y)). In our example, we treat the y coordinate value as the most significant and thus make use of the values in column 5.

Assume that the primary and overflow buckets are both of size 2 and that a bucket will be split whenever τ, the storage utilization factor, is greater than or equal to 0.66 (i.e., $\alpha = 0.66$). We also assume that there is a one-to-one correlation between the numbering of the grid cells and the primary bucket labels and thus there is no need for a grid directory. Therefore, we use numbers to refer to both the grid cells and the primary buckets associated with them. Inserting the cities in the order `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami` yields the partition of the database shown in Figure 79. which has six primary buckets, labeled `0`–`5`, and two overflow buckets. To understand the splitting process, we examine more closely how Figure 79 is constructed.

Initially, the file consists of just grid cell `0` and bucket `0` which is empty. The file is of level 0,1. The pointer to the next grid cell to be split, s, is initialized to `0`. `Chicago` and `Mobile` which lie in grid cell `0`, are inserted in bucket `0` which yields $\tau = 1.00$. This causes grid cell `0` to be split and bucket `1` to be allocated. s retains

Name	x	y	CODE(x)	CODE(y)
Chicago	2	3	44	28
Mobile	4	0	1	2
Toronto	4	6	11	7
Buffalo	6	5	39	27
Denver	0	3	40	20
Omaha	2	2	12	12
Atlanta	6	1	37	26
Miami	7	0	21	42

Figure 78: Reversed bit interleaving applied to the result of f(z)=z÷12.5.

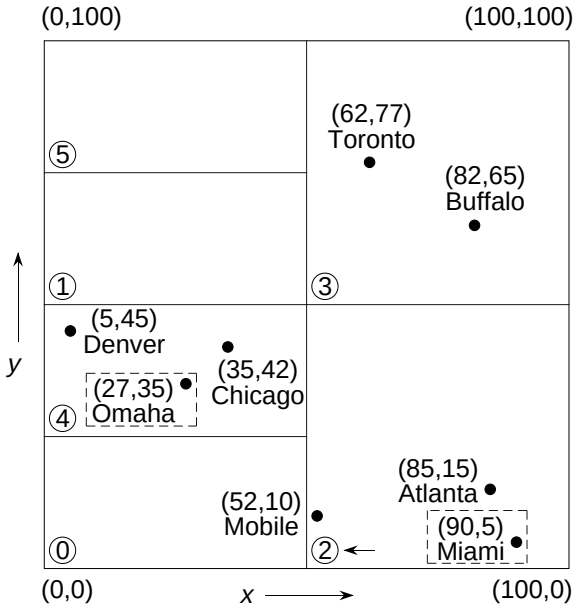


Figure 79: The representation resulting from applying linear hashing using OPLH with reversed bit interleaving corresponding to the Figure 1. The y coordinate value is assumed to be the most significant. Overflow bucket contents are enclosed by broken lines. A leftward-pointing arrow indicates the next bucket to be split.

its value of 0 and both `Chicago` and `Mobile` remain in bucket 0 (see Figure 80a).

`Toronto` lies in grid cell 1 and is inserted in bucket 1. However, now $\tau = 0.75$ thereby causing grid cell 0 to be split, s to be set to 1, bucket 2 to be allocated, and `Mobile` to be placed in it (see Figure 80b). Our file is now of level 1,2.

Next, we try to insert `Buffalo` which lies in grid cell 1 and hence is inserted in bucket 1. However, now $\tau = 0.67$ thereby causing grid cell 1 to be split, s to be reset to 0, bucket 3 to be allocated, and `Toronto` and `Buffalo` to be placed in it (see Figure 80c).

`Denver` lies in grid cell 0 and is inserted in bucket 0. `Omaha` lies in grid cell 0 and hence it is inserted in bucket 0. However, now $\tau = 0.75$ thereby causing grid cell 0 to be split, s to be set to 1, bucket 4 to be allocated, and `Denver` and `Chicago` to be moved into it. Our file is now of level 2,3. Unfortunately, `Omaha` also lies in grid cell 4 but bucket 4 is full and thus an overflow bucket must be allocated, attached to grid cell 4, and `Omaha` is placed in it (see Figure 80d).

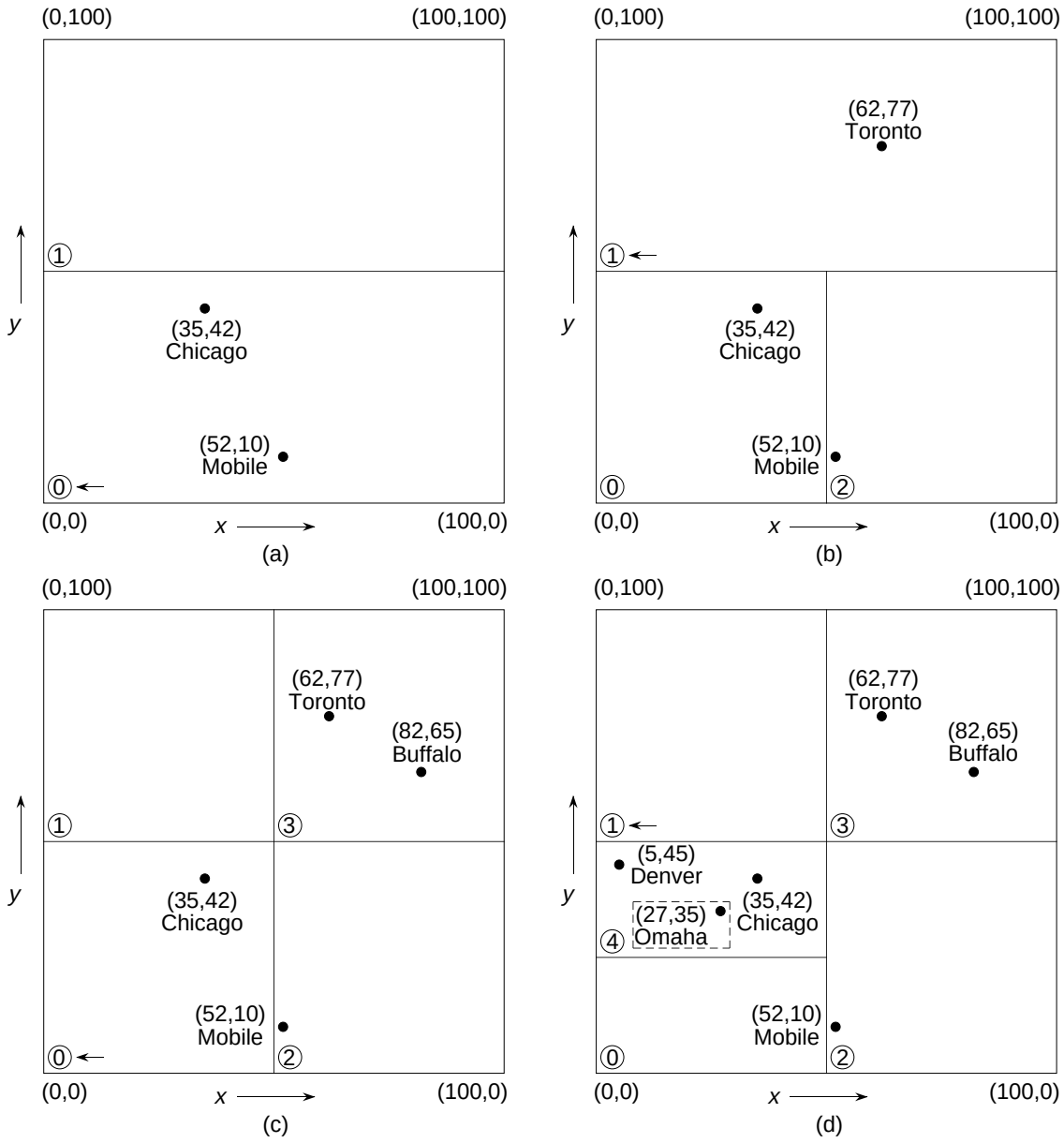


Figure 80: Snapshots of the representation resulting from applying linear hashing using OPLH with reversed bit interleaving after the insertion of (a) Chicago and Mobile, (b) Toronto, (c) Buffalo, and (d) Denver, and Omaha. The y coordinate value is assumed to be the most significant. Overflow bucket contents are enclosed by broken lines. A leftward-pointing arrow indicates the next bucket to be split.

`Atlanta` lies in grid cell 2 and is placed in bucket 2. `Miami` also lies in grid cell 2 and is placed in bucket 2. However, now $\tau = 0.67$ thereby causing grid cell 1 to be split, s to be set to 2, and bucket 5 to be allocated. After the grid cell split we still have too many items in bucket 2 and thus an overflow bucket must be allocated, attached to grid cell 2, and `Miami` is placed in it (see Figure 79).

The drawback of using OPLH with reversed bit interleaving as the hashing function h is that the order in which the grid cells are split is not consistent with a splitting policy that introduces a $(d-1)$-dimensional partitioning hyperplane l (as in the grid file), and then splits the group of grid cells intersected by l (termed a

slice thereby creating an additional slice) before attempting to split other grid cells[39]. Such a splitting order is more in line with the manner in which the grid file grows and is frequently desired as it preserves some locality, at least across the key k that is being split. In particular, these grid cells are spatially contiguous within the range of k that has been split and hence are more likely to be full when the split is made (this is especially true when we use the PLOP hashing variant of linear hashing described in Section 7.2.4). Thus we need another hashing function f. f must also stipulate the order in which the grid cells in the slice intersected by l are to be split and buckets allocated for them (as well as deallocated, if necessary, should there have been overflow buckets associated with some of the corresponding grid cells in l).

f results in partitioning the axes corresponding to the d keys in a cyclic manner starting with one partition for each of the keys thereby resulting in 2^d grid cells after completing the first cycle of partitions. Next, f cycles through the keys partitioning each key's axis twice, each key's axis four times on the next cycle, etc.[40] As grid cells are partitioned, new ones are created which are assigned numbers in increasing order, assuming that initially the entire set of data is in grid cell `0`. The central property of f is that all the grid cells in slice a are split in two before any grid cells in any of the other slices, while splitting the grid cells in a in the order in which they were created[41]. In contrast, the reversed bit interleaving hashing function h uses a different split policy in that it does not introduce a $(d-1)$-dimensional partitioning hyperplane. Instead, h results in splitting the grid cells in the order in which they were created, regardless of the identity of the slices in which they are contained.

In order to see this distinction between f and h, assume that the partitions are made on a cyclic basis $yxyxy\ldots$ (i.e., the y coordinate value is more significant than the x coordinate value). Figures 81 and 82 show the order in which the first 16 grid cells (starting with the initial grid cell `0`) are partitioned for both h and f (also known as *MDEH* as described below), respectively. The numbers associated with the cells correspond to the order in which they were created. Assume that the underlying space is a unit-sized square with an origin at the lower-left corner and that it is partitioned by a recursive halving process along the range of the corresponding key so that after each cycle of partitions through all of the keys, the result is a grid of cells having a uniform width across all keys. The labels along the axes in Figure 82 indicate the relative time at which the partition was performed along the axis (subsequently referred to as the *partition number*).

Note that in the transition from Figure 81b to Figure 81c, grid cells `0` and `1` are split thereby creating grid cells `4` and `5`. However, these grid cells (i.e., `0` and `1`) are not partitioned by the same one-dimensional hyperplane (which would be a line in two dimensions as is the case here). In contrast, the transition from Figure 82b to Figure 82c does proceed by completely partitioning one slice before partitioning another slice. Thus we find that the first two grid cells that are split (i.e., `0` and `2`) are partitioned by the same hyperplane (i.e., the line $y = 0.25$) thereby creating grid cells `4` and `5`.

Figure 83 shows the correspondence between the partition numbers in Figure 82 (i.e., the labels along the axes) and the numbers of the grid cells that are created by the first three partitions along each key (i.e., two cycles for each key) when $d = 2$. This correspondence can be calculated by using MDEH (denoting *Multidimensional Extendible Hashing* [108][42]) which serves as the desired hashing function f thereby obviating the need for the directory shown in Figure 83.

To convince ourselves that the MDEH function really does what it is supposed to do, let us determine the grid cell that contains the point ($x = 0.3, y = 0.6$). Once again, we assume again that the underlying space of Figure 82 is a unit-sized square with an origin at the lower-left corner which is partitioned by a recursive halving process along the range of the corresponding key so that after each cycle of partitions through all of the keys, the result is a grid of cells having a uniform width across all keys. The labels along the axes in Figure 82 indicate the partition number that causes the creation of the $(d-1)$-dimensional region of grid

[39] This problem also holds for reversed bit concatenation. We do not discuss it here as it favors some keys over others.

[40] We shall see later that the cyclic partitioning order is not a requirement on the definition of f. In other words, other partition orders are also possible as long as the number of partitions along the axis of each key doubles for each subsequent partitioning along that key. However, for the present, we assume a cyclic partition order.

[41] The grid cells in a that are being split were not necessarily created in immediate succession (i.e., they are not necessarily numbered in consecutive order). In other words, grid cell c_i in a is split before grid cell c_j in a if grid cell c_i was created earlier than grid cell c_j although, of course, other grid cells in different slices than a could have been created in the time between the creation of c_i and c_j.

[42] Also referred to as the access function for a uniform extendible array of exponential varying order (UXAE).

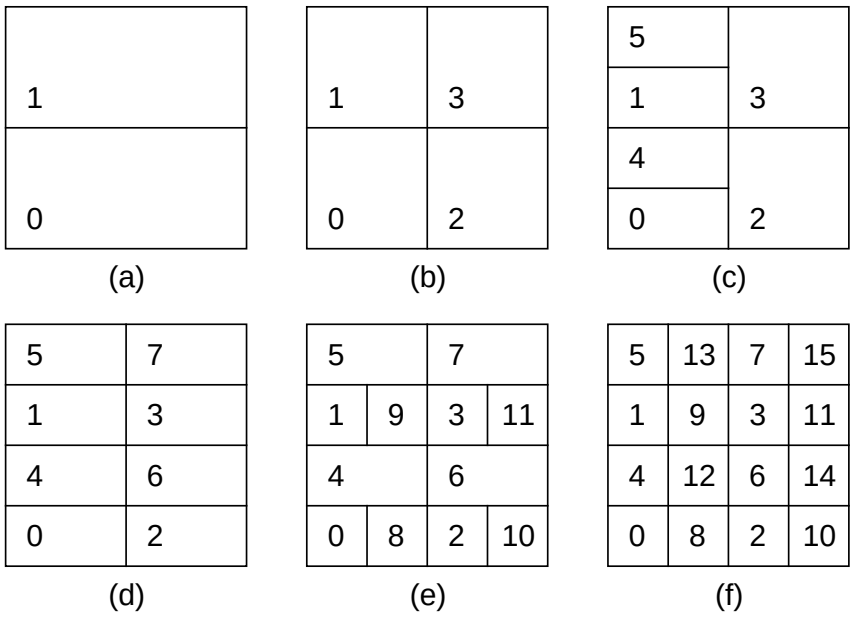


Figure 81: The order in which the underlying space is partitioned while creating the first 16 grid cells when applying linear hashing using OPLH with reversed bit interleaving (i.e., h) where the y coordinate is the most significant. The grid cells are numbered in the order in which they were created.

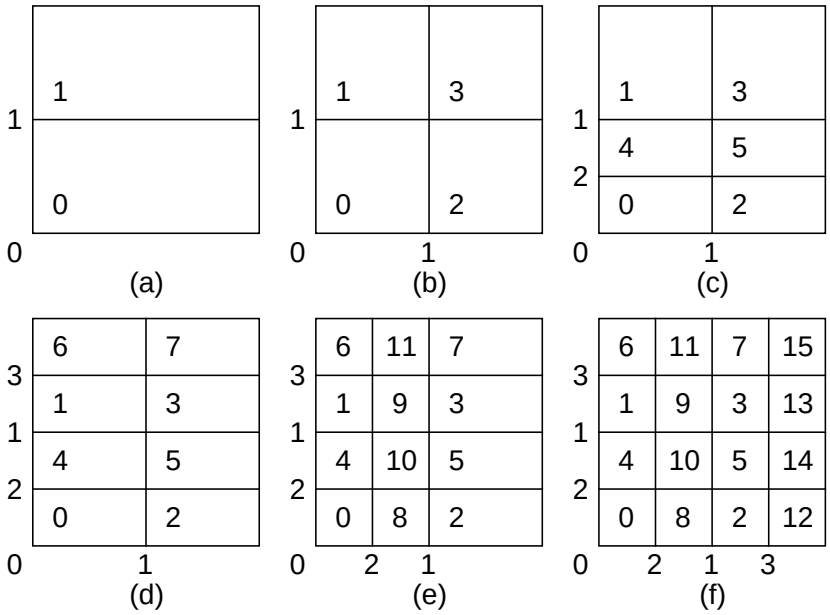


Figure 82: The order in which the underlying space is partitioned while creating the first 16 grid cells when using a hashing function that splits all grid cells within a slice before splitting other grid cell (i.e., f which is also MDEH) where the y coordinate is the most significant. The grid cells are numbered in the order in which they were created. The labels along the axes indicate the relative time at which the partition was performed along the axis.

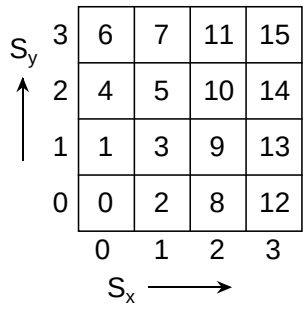


Figure 83: Correspondence between partition numbers (shown as horizontal and vertical axes of the table) and the number of the grid cells that are created by the first three partitions of the keys when all grid cells in one slice are split before splitting other grid cells. Assume that d=2 where the y coordinate is the most significant. This table is the same as the function MDEH.

cells immediately to its right (for the x coordinate) or above (for the y coordinate). We also assume that the partitioning process has cycled through both keys twice thereby resulting in Figure 82f. This information is used to determine from Figure 82f that our query point is in the grid cell created by the second partition along the x axis and the first partition along the y axis. Looking up the entry in Figure 83 corresponding to $S_x = 2$ and $S_y = 1$ yields grid cell number 9 which is indeed the number associated with the grid cell containing the point in Figure 82f.

Our explanation of how to determine the grid cell c that contains a point p at location r with r_i ($1 \leq i \leq d$) as the values for key i omitted one crucial step. In particular, we need to find a way to compute the partition numbers corresponding to the different partitioning hyperplanes for the keys so that they can be used as parameters to MDEH. In our explanation, the existence of d one-dimensional directories such as those found along the axes of Figure 82f which correlate the locations of the partitioning hyperplanes with the partition numbers was implicitly assumed, where in fact such directories do not exist. When the partitions are obtained by a recursive halving process of the range of the keys that comprise the underlying space (as is the case in this example), the equivalent effect of these one-dimensional directories can be achieved by applying the following five-step process to each key value r_k. Assume that key k has been partitioned q_k times such that $2^n \leq q_k < 2^{n+1}$ where n is the number of times a full cycle of partitions have been made through k.

1. Determine the range t_k of values for key k.
2. Calculate the width w_k of a grid cell for key k (i.e., $w_k = t_k/2^{n+1}$).
3. Compute the position u_k ($0 \leq u_k < 2^{n+1}$) of the grid cell containing r_k (i.e., $u_k = t_k \div w_k$).
4. Reverse the $n+1$ least significant bits of the binary representation of u_k to obtain v_k which is the desired partition number.
5. If $v_k > q_k$, then recalculate v_k using steps 2–4 with 2^n instead of 2^{n+1} in steps 2 and 3.

An alternative to the five-step process described above is to use a set of d one-dimensional directories in the form of linear scales (as in the grid file), one per key, implemented as binary trees (see Exercise 3). The nonleaf nodes of the binary trees indicate the key value corresponding to the partition[43], and the leaf nodes indicate the partition number. For example, Figure 84 shows the linear scales for the x and y axes corresponding to the partition of the underlying space given in Figure 82f. Note that after each full cycle of partitions through the d keys, the linear scales are identical (as is the case in our example).

[43] There is no need to store the key value in the nonleaf nodes, as they can always be determined from knowledge of their ranges and the path followed when descending from the root of the binary tree corresponding to the linear scale to the appropriate leaf node.

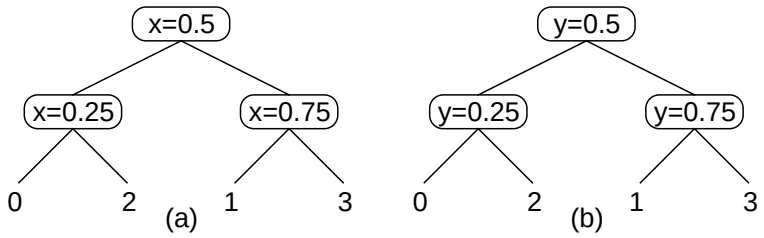


Figure 84: Linear scales showing the correspondence between coordinate values and partition numbers for the (a) x and (b) y axes corresponding to the partition of the underlying space given in Figure 82f. This can also be obtained by use of a five-step process outlined in the text.

The method outlined above for determining the grid cell c that contains a point p given its location r demonstrates the second principal way in which f (i.e., MDEH) differs from the hashing functions used with linear hashing. In particular, c is not obtained by directly applying f to r (or to the result of reversing the bit interleaved or bit concatenated representation of the key values that make up r as in linear hashing with OPLH). Instead, c is determined by applying f to the partition numbers along the axes corresponding to the values of its keys that caused c to be created. The actual correspondence between partition numbers and key values is obtained by use of either the linear scales or the five-step process outlined above given the number of times each key has been partitioned.

Upon some reflection, it should be apparent that f, in conjunction with the partition numbers, plays the same role as the table given in Figure 82f to indicate the two-dimensional correspondence between key values and grid cell numbers. Unfortunately, we do not have the luxury of having space to store this table which is why we make use of f. It should be clear that the lack of this of this table is what sets linear hashing apart from the grid file (for which the table exists), which, after all, is the point of this discussion (i.e., to avoid the table). Note that the use of the linear scales or the five-step process to calculate the partition numbers which are then supplied as parameters to the hash function f to determine the grid cell containing the query point obviates the need for the hashing function f to satisfy the ordering properties 1–3 outlined earlier in this section. In contrast, when using a function such as OPLH, we usually apply the hash function directly to the result of interleaving or concatenating the bits of the query point rather than computing the partition numbers corresponding to the coordinate values of the query point as is the case when using MDEH.

Grid cells are split whenever the storage utilization factor τ exceeds a predetermined value, say α. We use a split index (termed an *expansion index* [76]) in the form $E = (e_1, e_2, \ldots, e_d)$ to indicate which grid cell c is to be split next. The identity of c is determined by applying the MDEH hash function to the d values of e_i where e_i is the partition number corresponding to key i. The actual split occurs by passing the $(d-1)$-dimensional hyperplane corresponding to partition e_k through c. The split procedure is as follows. Assume that we are in the j^{th} split cycle (meaning that we have cycled through the d keys $j-1$ times) for key k. Thus we must now apply 2^{j-1} partitions to k. We first determine the slice that is being partitioned (i.e., e_k and termed the *expansion slice*). Next, we check if we are splitting the first cell in slice e_k (i.e., $e_i = 0$ for all i where $1 \leq i \leq d$ and $i \neq k$). In this case, we compute a partition point along key k which partitions the slice in two. Next, we split the cell.

Once a cell has been split, we check if all of the grid cells intersected by the current partition (i.e., in the expansion slice) have been split. If no, then we determine the next cell to be split which is done in increasing lexicographic order of E while holding e_k fixed. If yes, then, assuming that linear scales are used[44], we update the linear scale of k to contain the partition value as well as the new partition numbers, and reset the expansion index. We then proceed to the next partition on key k by incrementing e_k by 1 unless we are processing the final slice (i.e., the $2^{j-1\,th}$ slice in the current split cycle) in which case we proceed to the next key in cyclic order and possibly increase the cycle number j if $k = d$.

[44] In the rest of this discussion we assume that linear scales are used as this will make the process general and enables it to be used with quantile hashing and PLOP hashing as described in Section 7.2.4.

At this point, let us describe the process of determining the grid cell c associated with a query point $p = (p_1, p_2, \ldots, p_d)$ in more detail. As mentioned above, we look up the coordinate values of p in the linear scales (or use our five-step procedure) and obtain the partition numbers for the different keys, say s_i ($1 \leq i \leq d$). We must now determine if c lies inside or outside the expansion slice as this indicates whether we can apply f directly to s_i to obtain the value of c. Assume that we are currently partitioning on key k. Therefore, from the expansion index E we know that e_k denotes the partition number thereby enabling the identification of the expansion slice for key k. We can determine whether c will be inside or outside the expansion slice by examining the values of s_i ($1 \leq i \leq d$).

1. If $s_k = e_k$ and if the sequence s_i ($1 \leq i \leq d$) is lexicographically less than E, then the grid cell containing p has already been expanded. This means that we have to recompute the partition number s_k, say q, for key k by including the expansion partition in the lookup process that uses the linear scales of key k (or use our five-step procedure). The function f is now applied to s_i ($1 \leq i \leq d$) with q replacing s_k to yield the grid cell corresponding to c.
2. If c is outside the expansion slice, then we apply f to s_i ($1 \leq i \leq d$) to yield the grid cell corresponding to c.

For example, consider the grid partition given in Figure 85a where the x coordinate is currently being partitioned (again assuming a cyclic order $yxyx\ldots$). In particular, we have cycled through the y axis twice and through the x axis once. We are currently in the second partition on the second cycle of the x axis with an expansion index value of $E = (2, 1)$ (denoting that the next grid cell to be split is the one formed by the second partition on y and the first partition on x). The corresponding linear scales are given to the left of the y axis and below the x axis. Let us first search for the point ($x = 0.3$,$y = 0.8$). From the linear scales we have that $S_y = 3$ and $S_x = 2$. Since $S_x \neq 1$, the grid cell containing ($x = 0.3$,$y = 0.8$) is not in the expansion slice and thus the grid cell is obtained by applying MDEH (i.e., grid cell 11).

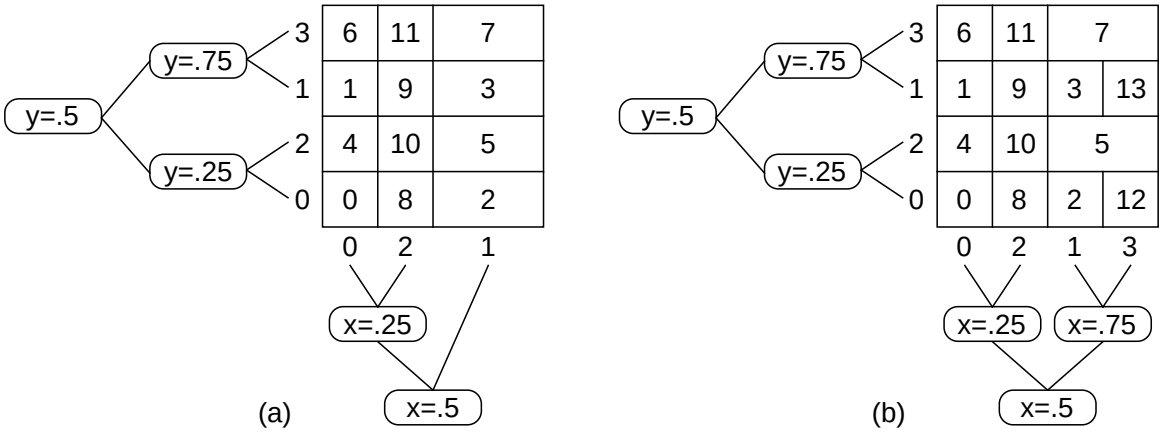


Figure 85: (a) Example grid partition illustrating the use of MDEH with the y coordinate being the most significant. Linear scales are given to the left of the y axis and below the x axis. The expansion index has a value E=(2,1). (b) The modified linear scale for x to reflect the fact that the third partition on the x axis has not yet been completed but is used when the query point is in a grid cell in the expansion slice which has already been expanded. The broken lines indicate that the grid cell has not been split yet.

As another example, suppose that we search for the point ($x = 0.8$,$y = 0.6$). From the linear scales we have that $S_y = 1$ and $S_x = 1$. Since $S_x = 1$, the grid cell containing ($x = 0.8$,$y = 0.6$) is in the expansion slice. Moreover, since $(S_y, S_x) = (1, 1) <^L (2, 1) = E$ (where $<^L$ denotes the lexicographic ordering), the grid cell containing ($x = 0.8$,$y = 0.6$) has already been expanded. Therefore, we have to include this expansion in the linear scales (see Figure 85b) when looking up the value of S_x which is now 3. Applying

MDEH to $S_y = 1$ and $S_x = 3$ yields grid cell `13`. Note that had we looked for the point $(x = 0.8, y = 0.8)$ we would have $S_y = 3$ and $S_x = 1$ which is also in the expansion slice. However, $(S_y, S_x) = (3, 1)$ is not lexicographically less than $E = (2, 1)$ and thus we can apply MDEH directly to obtain grid cell `5`.

Both OPLH and MDEH have the shortcoming that grid cells at consecutive addresses (i.e., buckets) are not usually in spatial proximity. Equivalently, grid cells that are in spatial proximity are not usually in consecutive addresses (i.e., buckets). If the numbers of the grid cells correspond to disk addresses (not an unreasonable assumption), then execution of queries such as multidimensional range queries (e.g., a hyper-rectangular window query) usually results in a large number of seek operations. In order to see this, consider Figure 86 which shows the correlation of grid cells to the underlying space for OPLH with reversed bit interleaving (Figure 86a), MDEH (Figure 86b), and the Morton order whch is the same as conventional bit interleaving (Figure 86c) where the y coordinate is the most significant after two complete cycles of splitting along the two keys. In particular, let us examine the grid cells that overlap the 2×2 search window in the upper half of each part of the figure (shown with a heavy line). Notice that only for conventional bit interleaving (i.e., the Morton order) are any of the covering grid cells at consecutive addresses. The goal is to make this number as large as possible.

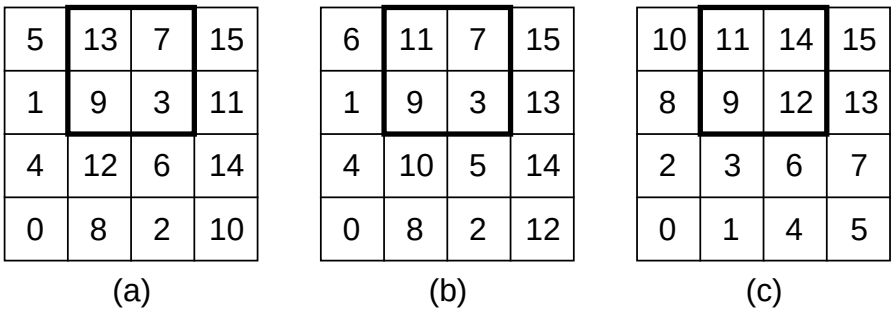


Figure 86: Correlation of grid cell addresses to the underlying space for (a) OPLH with reversed bit interleaving, (b) MDEH, and (c) Morton order (i.e., conventional bit interleaving) where the y coordinate is the most significant after two complete cycles of splitting along the two keys. The heavy lines denote the boundaries of a 2×2 search window in the upper half of the underlying space.

The problem is in the order in which the newly created grid cells are created. In particular, in OPLH and MDEH, when a grid cell is split, the grid cells resulting from the split are not stored in consecutive addresses. Thus we need a method that preserves proximity by having the property that whenever grid cell at address c is split, it must result in the creation of two grid cells at consecutive addresses, say at a and $a + 1$. Since the grid cell at address $c + 1$ is usually occupied, we cannot split c to create c and $c + 1$. Moreover, when the grid cell at address $c + 1$ is split, we would like its result to be near the result of splitting the grid cell at address c (i.e., a and $a + 1$). Interestingly, the Morton order does satisfy these requirements in that it has the property that a grid cell at address c is split into two grid cells at addresses $2c$ and $2c + 1$ (see Exercise **??**). The Morton order is the basis of the *z$^+$-ordering* [64]. The difference from the Morton order is that the z$^+$-ordering enables the representation of the partition at times other than when a key is fully expanded (i.e., when we have doubled the number of grid cells as a result of splitting all of the grid cells for a particular key). When a key is fully expanded, the z$^+$-ordering is equivalent to the Morton order.

The central idea behind the z$^+$-ordering is to determine a sequence in which the grid cells are to be split so that the ordering is preserved. Assume that we have just completed L expansions thereby having 2^L grid cells with addresses $0, 1, \ldots, 2^L - 1$, and we are now ready to start on the $L + 1^{st}$ expansion. Clearly, when we split the grid cell at address c to yield grid cells at addresses $2c$ and $2c + 1$, these grid cells must not be occupied. This property does not hold if we try to split the grid cell at address 0. One way to ensure that this does not occur is to split the grid cells in decreasing order starting with the grid cell at address $2^L - 1$ thereby creating grid cells at addresses $2^{L+1} - 2$ and $2^{L+1} - 1$. The problem with this approach is that it creates a large gap of unused grid cells which contradicts the purpose of linear hashing which is to extend the range of addresses in an incremental manner as needed. Moreover, it results in low storage utilization.

Ideally, if there are currently $4t$ grid cells in use (at addresses $0, 1, \ldots, 4t-1$), then we want to place the newly-created grid cells at addresses $4t$ and $4t+1$. This means that we split the grid cell at address $2t$ thereby creating a gap (i.e., an unused grid cell) at $2t$. This is fine as long as we do not create too many gaps and if the gaps are small. Thus any technique that we use must fill up the gaps as soon as possible, as well as minimize their number. The gaps are filled by splitting the predecessor grid cells. For example, suppose that after splitting the grid cell at address $2t$, we split the grid cell at address $2t+1$ thereby creating grid cells at addresses $4t+2$ and $4t+3$. This means that we now have a gap at addresses $2t$ and $2t+1$ which means that the next grid cell to be split is the one at address t. This process is continued recursively until all grid cells have been split and we have 2^{L+1} grid cells at addresses $0, 1, \ldots, 2^{L+1}-1$. This splitting method is termed *dynamic z hashing* [64].

As an example of dynamic z hashing, consider Figure 86c which corresponds to the mapping of grid cells and addresses when using the Morton order for the fourth expansion (i.e., $L=4$). During the fifth expansion (i.e., $L=5$), the grid cells are split in the following order where we use the notation $(a:bc)$ to denote that the grid cell at address a has been split to create grid cells at addresses b and c: `(8:16 17)`, `(9:18 19)`, `(4:8 9)`, `(10:20 21)`, `(11:22 23)`, `(5:10 11)`, `(2:4 5)`, `(12:24 25)`, `(13:26 27)`, `(6:12 13)`, `(14:28 29)`, `(15:30 31)`, `(7:14 15)`, `(3:6 7)`, `(1:2 3)`, `(0:0 1)`.

As mentioned above, the cost of using the z^+-ordering is a lower storage utilization due to the existence of unused grid cells (i.e., gaps). The storage efficiency can be obtained by calculating the average number of gaps during a full expansion. Assuming that L full expansions have been carried out, it can be shown that the $L+1^{st}$ expansion has an average of $L/2$ gaps (see Exercise 9) [64]. Experiments comparing the z^+-ordering with a variant of MDEH have shown that use of z^+-ordering for hyper-rectangular range queries results in a significant reduction in the number of sequences of consecutive grid cells that need to be examined thereby lowering the number of disk seek operations that will be needed and thus yielding faster execution times.

A general drawback of linear hashing is that during each expansion cycle, the load factor of the buckets corresponding to the grid cells that have been split is only one half of the load factors of the buckets corresponding to the grid cells that have not yet been split in the cycle. Thus the records are not uniformly distributed over all of the buckets thereby leading to worse query performance (e.g., search). The performance is worse because there are more overflow buckets to be searched. Larson [79] proposes a variant of linear hashing termed *linear hashing with partial expansions (LHPE)* to overcome this problem. The key idea is that the number of grid cells is doubled in a more gradual manner through the application of a sequence of partial expansions. This technique has been described for conventional linear hashing in Section **??** of Chapter **??**. In this section, we restrict our discussion to the case of just two partial expansions.

In essence, assuming that a file contains $2N$ buckets, the doubling can be achieved in a sequence of two partial expansions where the first expansion increases the file size to 1.5 times the original size while the second expansion increases it to twice the original size. In particular, the original file is subdivided into N pairs of buckets, say 0 and N, 1 and $N+1$, ..., $N-1$ and $2N-1$. Now, instead of splitting a bucket as in linear hashing, the first partial expansion creates a new bucket $2N$ corresponding to the first pair 0 and N and rehashes their contents into buckets 0, N, and $2N$. This process is repeated until we have $3N$ buckets at which time a similar second partial expansion is applied to N trios of buckets where a new bucket is created for each trio of buckets until we have $4N$ buckets. The next time we need to allocate a new bucket, we start a new sequence of two partial expansions resulting in the file growing to $6N$ and $8N$ buckets after each partial expansion. If more that two (say q) partial expansions are used, then the number of buckets after each partial expansion would be increased by a factor of $1/q$ times the number of buckets at the beginning of each full expansion cycle.

Partial expansions can be easily used with OPLH. Using partial expansions with MDEH is a bit more tricky as the expansion unit is a slice rather than a grid cell. This means that instead of expanding groups of individual grid cells, we must expand groups of slices. Kriegel and Seeger [74] take this into consideration by proposing to split two adjacent slices into three on the first partial expansion for key k followed by a split of three adjacent slices into four on the second partial expansion for k. They term the result *Multidimensional Order-preserving Linear Hashing with Partial Expansion (MOLHPE)* and give a function for its computation.

Exercises

See also the exercises in Section **??** of Chapter **??**.

1. Does the Peano-Hilbert ordering (see Section **??** of Chapter **??**) satisfy properties 1–3?
2. Write a procedure `LIN_RANGE_SEARCH` to perform a range search for a rectangular region in a 2-dimensional database implemented with linear hashing and OPLH using reversed bit interleaving as the hashing function.
3. What is the drawback of implementing the linear scales as one-dimensional arrays?
4. Give an algorithm for computing the value of MDEH for a set of partition numbers s_i ($1 \leq i \leq d$) where we have made j cycles through each key.
5. Give an algorithm for computing the value of MDEH for a set of partition numbers s_i ($1 \leq i \leq d$) where we have made j cycles through each key and that uses two partial expansions (i.e., MOLHPE).
6. Write a procedure `MDEH_POINT_SEARCH` to perform a point search a 2-dimensional database implemented with MDEH as the hashing function.
7. Write a procedure `MDEH_RANGE_SEARCH` to perform a range search for a rectangular region in a 2-dimensional database implemented with MDEH as the hashing function.
8. Give an algorithm to implement dynamic z hashing.
9. Show that the average number of unused grid cells (i.e., the sizes of the gaps) during the $L + 1^{st}$ expansion when using the z^+-ordering is $L/2$.
10. One of the main properties that enables the Morton order to be used as the basis of dynamic z-hashing is that a grid cell at address c is split into two grid cells at addresses $2c$ and $2c + 1$. Does this property also hold for the Peano-Hilbert order? If it does not hold, can you state and prove how often it does hold? Also, if it does not hold, can you find a similar property that does hold for the Peano-Hilbert order?
11. The z^+-ordering [64] was devised in order to ensure that when a grid cell is split in linear hashing, the newly created grid cells are in spatial priximity. The z^+-ordering makes use of a Morton order. Can you define a similar ordering termed *p^+-ordering* which makes use of a Peano-Hilbert order instead of a Morton order? If yes, give an algorithm for the splitting method that is analogous to dynamic z hashing (i.e., *dynamic p hashing*).
12. Compare the z^+-ordering and the p^+ ordering (i.e., the Morton and Peano-Hilbert orderings as described in Exercise 11) in terms of their storage efficiency by examining the average number of gaps. The comparison should be analytic. If this is not possible, then use an experimental comparison.
13. Compare the z^+-ordering and the p^+ ordering (i.e., the Morton and Peano-Hilbert orderings as described in Exercise 11) in terms of their efficiency in answering range queries by examining the average number of sequences of consecutive grid cells that are examined. The comparison should be analytic. If this is not possible, then use an experimental comparison. Do the results match the general comparison of the efficiency of the Morton and Peano-Hilbert orders described in [68]?

7.2.4 Variants of Linear Hashing

Regardless of whether OPLH or MDEH are used as the hashing functions, linear hashing has the drawback that the grid cell that has been split most recently is not necessarily the one that is full. Three approaches, quantile hashing [74, 75, 77], PLOP (denoting *piecewise linear order preserving*) hashing [76], and spiral hashing [90, 80, 95] (recall Section **??** of Chapter **??**), are discussed below that try to overcome this drawback by, in part, varying the range of values (i.e., in the underlying data domain) that can be associated with some

of the grid cells (true for all three approaches), and by allowing some flexibility in choosing the next grid cells to be partitioned (true for PLOP hashing). Two of the approaches (i.e., quantile hashing and PLOP hashing) make use of MDEH as the underlying hashing function, while spiral hashing makes use of OPLH.

These approaches have the effect of permitting greater variability in the size of some of the grid cells thereby differing from conventional linear hashing, as described in Section 7.2.3, where the number of possible sizes of the grid cells is quite limited[45]. Moreover, in both quantile and PLOP hashing, the space U spanned by the grid cell is still a hyper-rectangle although subsets of the individual grid cells no longer need to be similar. In contrast, in spiral hashing, U can have an arbitrary shape and, in fact, need not always be spatially contiguous.

A central difference between these three approaches is the action taken when the storage utilization factor is exceeded (i.e., the determination of which cell or group of cells is to be partitioned; or, if there is no choice, then where to place the partition).

1. Quantile hashing partitions a group of grid cells on the basis of a stochastic approximation of what the future distribution of the incoming data might look like for the associated quantile which is always a power of $1/2$.
2. PLOP hashing partitions a group of grid cells on the basis of which one contains the maximum number of data items.
3. Spiral hashing partitions the grid cell for which U is a maximum.

Quantile hashing [74, 75, 77] is just an implementation of the MDEH hashing function where instead of partitioning each expansion slice of key k in half, we compute a partition point along key k which partitions the quantile associated with the slice so that approximately one half of the incoming values that are in the range of the expansion slice will fall in each of the two resulting slices. For example, for the partition associated with the $1/2$ quantile, its position is based on the estimate of the median along the key. Linear scales in the form of binary trees described in Section 7.2.3 are used to keep track of the actual partition points for each key thereby enabling the correlation of the partitioning hyperplane with the partition numbers which are used as parameters to MDEH so that we can determine the grid cell associated with a particular point. The grid cell split procedure is the same as that used for the implementation of linear hashing with the MDEH hashing function (i.e., it makes use of the expansion index).

Figure 87 is an example of the type of partitioning of the underlying grid that is supported by quantile hashing. In essence, we have taken the partition of the underlying space given by Figure 82f and varied the positions of the partition lines from being the result of a recursive halving process. The actual partition points are recorded by the linear scales which are given to the left of the y axis and below the x axis. Notice that in the figure we have also identified the quantiles in the nonleaf nodes by use of the notation '`q: quantile value`' although this is not necessary if we always use quantiles that are powers of two. In particular, we can determine the quantile by noting the path followed when descending from the root of the linear scale to the appropriate leaf node.

The drawback of quantile hashing is that it assumes that the distribution of the incoming data does not change over time (i.e., it is stable). For example, if the data comes in a sorted order, say increasing without loss of generality, for a particular key (or lexicographically for all keys), then we have a problem as soon as we perform the first partition. In particular, the partition associated with the $1/2$ quantile results in the first slice being repeatedly partitioned in future cycles while no future data will ever be encountered that falls into its range. As another example, suppose that the data tends to cluster around a few points in the underlying space (e.g., areas of dense population). If the cluster centers suddenly change, then quantile hashing will perform

[45] In particular, when OPLH is the hashing function (as well as in EXCELL) there are two possible sizes for the grid cell depending on whether or not the cell has been split in the current cycle of splits across the different keys. When MDEH is the hashing function, the range of grid cell sizes is larger as it depends on the number of times a particular slice has been split in the current cycle of splits across the different keys. However, the sizes of the grid cells only differ by factors that are powers of two. Note that no matter which hashing function is used, for linear hashing, at any instance of time after a complete cycle through all of the keys, all grid cells are of equal size.

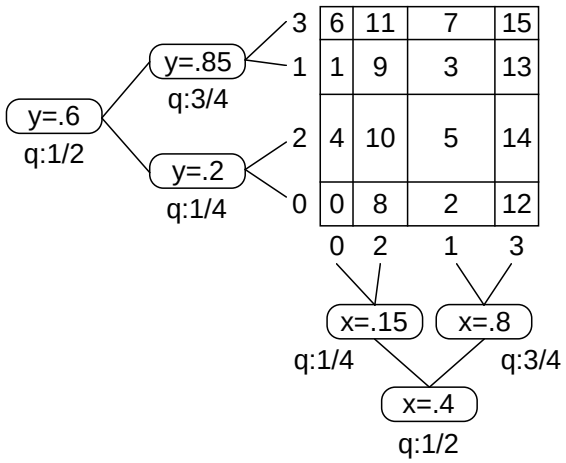


Figure 87: Example grid partition illustrating quantile hashing with the y coordinate being the most significant. Linear scales are given to the left of the y axis and below the x axis. The quantiles are also identified in the non leaf nodes of the linear scales in the form of 'q: qantile number'.

poorly (at least for awhile) as it had adjusted the positions of its partitions according to the old and known cluster centers.

The solution to the problems mentioned above is to apply repartitioning. Unfortunately, this is a costly process as it involves moving the positions of the partitions and, more importantly, moving the data in the corresponding grid cells. The alternative is to base the partitioning on the known data and to split the slices that contain the most data points. This is the basis of PLOP hashing [76]. PLOP hashing is similar to quantile hashing in the sense that it cycles through the keys and doubles the number of partitions along each key's axis for each key on successive cycles[46]. However, the difference is that where in each cycle for a given key, quantile hashing splits each slice into two new slices, PLOP hashing may repeatedly split the same slice during the cycle if it turns out that it is the one that contains the most data points. Thus PLOP hashing is much more suited to a dynamically changing distribution of data. Of course, both methods create the same number of new slices for key k during each cycle (i.e., they are doubled thereby also doubling the number of grid cells). However, whereas in quantile hashing each grid cell is split during each cycle, in PLOP hashing some grid cells may never be split during a cycle while other grid cells may be split many times by virtue of the fact that their containing slice is split many times during the cycle.

PLOP hashing is implemented by modifying the binary trees corresponding to the linear scales associated with the axes of the keys in quantile hashing as well as the implementation of the MDEH hashing function. Once again, the nonleaf nodes record the positions of the partitions. However, the fact that the partitions may be applied to the same slice several times during a cycle means that the resulting binary tree is no longer "almost complete" (recall Section **??** of Chapter **??**) in the sense that there is no longer a difference of at most one between the depths of all the leaf nodes as is the case with quantile hashing. Each leaf node corresponds to a slice i and contains the number of the partition along the axis of key k which resulted in the creation of i. This number is used as the parameter to the MDEH function when attempting to determine the grid cell that contains a particular query point p at location l. In addition, each leaf node contains the number of data points in the corresponding slice. This number is used to determine which slice to partition next for key k when the storage utilization factor τ exceeds α and all grid cells in the slice that is currently being partitioned have already been expanded. Of course, whenever a slice is partitioned, a new nonleaf node t is created in the binary tree for the new partition, a new slice allocated (with a partition number one higher than the current maximum for key k), and the count fields of the sons of t are set. Note that the grid cell c associated with a particular point is determined in the same manner as with quantile hashing in the sense that the linear scales in

[46] As mentioned in the original definition of f (and MDEH), the requirement that we cycle through the axes is not necessary. Thus both quantile hashing and PLOP hashing can be implemented without this requirement and this is pointed out at the end of this section.

conjunction with the expansion index are used to determine the appropriate partition numbers which are then used as arguments to the MDEH function.

Figure 88a is an example of the type of partitioning of the underlying grid that is supported by PLOP hashing. The actual partition points are recorded by the linear scales which are given to the left of the y axis and below the x axis. In essence, we have taken the partition of the underlying space given by Figure 82f and varied the positions of the partition lines from being the result of a recursive halving process. The linear scales in our figure are somewhat incomplete as their leaf nodes should also contain the number of data points in the corresponding slice. We have omitted this information here in the interest of reducing the complexity of the figure.

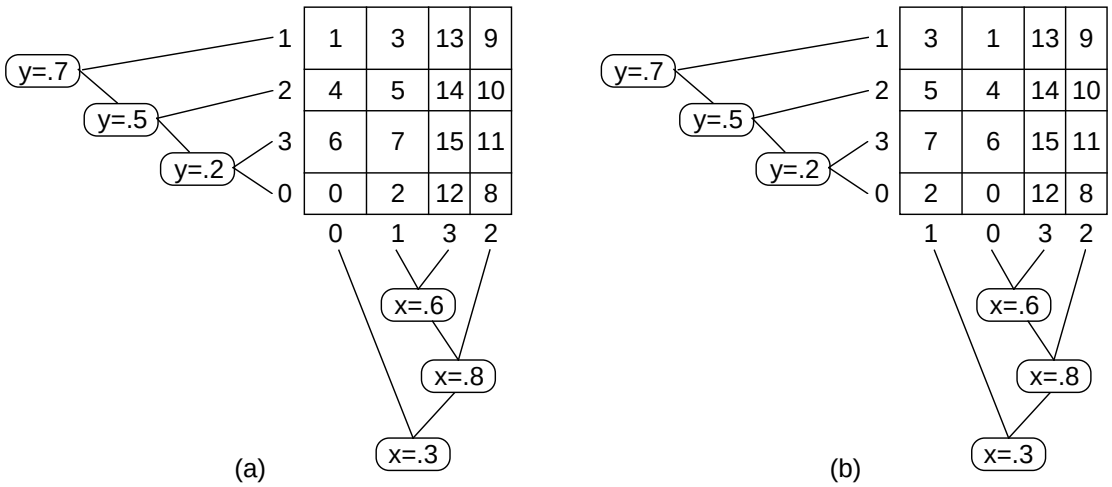


Figure 88: Example grid partition illustrating PLOP hashing with the y coordinate being the most significant. Linear scales are given to the left of the y axis and below the x axis. When slice 0 along the x axis is split in the first cycle, (a) the newly-created slice 1 is associated with the right half of the underlying space, while in (b) it associated with the left half of the underlying space. The leaf nodes of the linear scales should also contain the number of data points in the corresponding slice although this is not shown here.

There are number of interesting observations that can be made by looking at this example (i.e., Figure 88a). First, note that some slices have been partitioned several times in a cycle while others were not partitioned at all during some cycles. Here we see that on the first cycle we partitioned the y axis at $y = 0.7$ and the x axis at $x = 0.3$ Second, notice that during the second cycle, slice 0 along the y axis was partitioned twice and likewise for slice 1 along the x axis. Third, we observe that when slice 0 along the x axis was split in the first cycle, the newly-created slice 1 was associated with the right half of the underlying space. However, we could have also associated the newly-created slice with the left half of the underlying space as shown in Figure 88b. Clearly, this is permissible as the linear scales provide the association of partition numbers with regions of the grid.

Locating the grid cell containing a particular point with PLOP hashing is accomplished in the same way as for linear hasing with the MDEH hashing function. Once again, we find the leaf nodes in the binary trees of the linear scales that contain the region corresponding to the values of the keys, and then use the partition numbers recorded in the leaf nodes as arguments to MDEH. For example, let us first search for the point ($x = 0.4$,$y = 0.6$). Using Figure 88a, from the linear scales we have that $S_x = 1$ and $S_y = 2$. There is no expansion index here as we have completed two full cycles. Applying MDEH yields the grid cell `5`. Using Figure 88b, from the linear scales we have that $S_x = 0$ and $S_y = 2$. There is no expansion index here as we have completed two full cycles. Applying MDEH yields the grid cell `4`.

Spiral hashing is the third approach that we discuss. In this approach, we apply the same mapping g from

d dimensions to one dimension as in linear hashing. The difference is that the range of the mapping must be in $[0, 1)$. This is achieved by dividing the result k of the mapping by 2^s where s is the total number of bits that make up the bit interleaved or bit concatenated value. Such an action has the same effect as the function $h_n(k) = reverse(k) \bmod 2^n$ where *reverse* corresponds to reversed bit interleaving or reversed bit concatenation. Recall that this definition of $h_n(k)$ ensures that all records in the same grid cell or bucket agree in the values of the n most significant bits of k and hence they are within the same range.

Spiral hashing also decomposes the underlying values into disjoint ranges of k so that all points within a given range are stored in the same bucket (and possibly overflow buckets). Thus the extent of the range is analogous to the concept of a grid cell in linear hashing. Moreover, as in linear hashing, a primary bucket of finite capacity is associated with each range, as well as overflow buckets. The difference from linear hashing is that in spiral hashing the ranges are not equal-sized after a full cycle of splits for all keys. Again, as in linear hashing, buckets are split when the storage utilization factor τ exceeds a predetermined value α. Another difference from linear hashing is that the bucket b that is split is the one that corresponds to the largest-sized range. b's range is split into r ($r \geq 2$) unequal-sized ranges[47] that are smaller than any of the existing ranges. As with linear hashing, establishing a one-to-one correlation between the values of the spiral hashing function[48] $y(k) = \lfloor r^{x(k)} \rfloor$ and the primary bucket labels obviates the need for the one-dimensional array analog of the grid directory.

Using such a splitting policy, assuming a uniform distribution of data, the bucket corresponding to the range of values that is split is said to be more likely to have been full than the buckets that correspond to the smaller-sized ranges. Unfortunately, since the result of the one-dimensional mapping of the underlying space is no longer partitioned into equal-sized ranges (or a limited set of possible sizes at an instance which does not immediately follow a full cycle of splits for all keys) of values, there is less likelihood that the one-dimensional ranges correspond to a grid cell in d dimensions. This has the undesirable effect that the ranges of values in each partition are not always spatially contiguous as is the case for linear hashing. This limits the utility of this technique.

For example, Figure 89 shows the result of applying spiral hashing to the points of Figure 1 under the same conditions that were used to construct Figure 79 for linear hashing. Once again, we assume a bit interleaving ordering with the y coordinate being the most significant after application of the mapping f to the coordinate values where $f(z) = z \div 12.5$, primary and overflow bucket capacities of size 2, and that a bucket will be split whenever τ, the storage utilization factor, is greater than or equal to 0.66 (i.e., $\alpha = 0.66$). The cities are inserted in the order: `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`. Notice that the spatial regions associated with bucket `11` are not spatially contiguous.

Our presentation of quantile hashing, PLOP hashing, and spiral hashing all assumed a cyclic partitioning order. This is not absolutely necessary. In other words, we can use a variant of MDEH that permits the partitions for the different keys to proceed in arbitrary order. For example, assuming three-dimensional data, one possible key partition order would be $zzxyzxxxyy\ldots$. Such an ordering is achieved by modifying the MDEH function [93, 108] to keep track of the order in which the keys were partitioned. This is not needed when using a cyclic order as given the partition number and the cyclic order of the keys, we know the identity of the corresponding partitioning key. Of course, we still have the requirement that the number of partitions along each key's axis doubles for each subsequent partitioning along that key. Notice that this variation on the order is also possible for spiral hashing although once again we need to keep track of the order in which the keys are partitioned instead of via bit interleaving or bit concatenation as is usually the case.

Exercises

See also the exercises in Section **??** of Chapter **??**.

[47] r has the same meaning as d in Section **??** of Chapter **??**. We did not use d in this discussion as here d is used to denote the dimension of the underlying space.

[48] r has the same meaning as d in Section **??** of Chapter **??**. Again, we did not use d in this discussion as here d is used to denote the dimension of the underlying space.

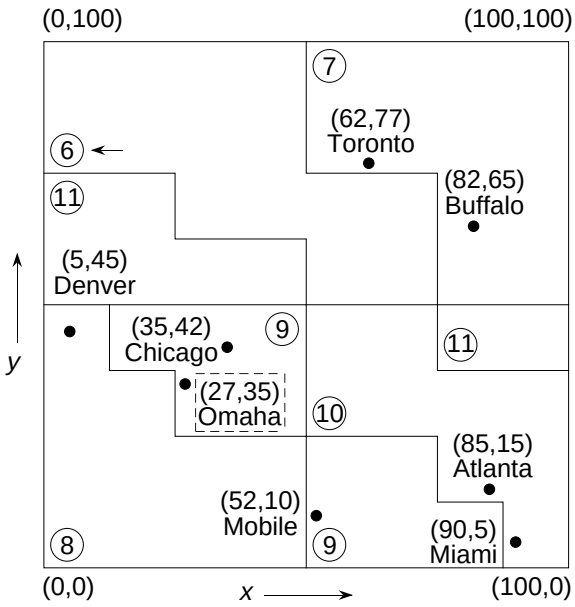


Figure 89: Spiral hashing representation using reversed bit interleaving corresponding to the data of Figure 1. The y coordinate value is assumed to be the most significant. Overflow bucket contents are enclosed by broken lines. A leftward-pointing arrow indicates the next bucket to be split.

1. Why couldn't you implement the linear scales for PLOP hashing as one-dimensional arrays? In other words, why must they be implemented as binary trees?

2. Write a procedure `QUANTILE_POINT_SEARCH` to perform a point search a 2-dimensional database implemented with quantile hashing.

3. Write a procedure `PLOP_POINT_SEARCH` to perform a point search a 2-dimensional database implemented with PLOP hashing.

4. Write a procedure `QUANTILE_RANGE_SEARCH` to perform a range search for a rectangular region in a 2-dimensional database implemented with quantile hashing.

5. Write a procedure `PLOP_RANGE_SEARCH` to perform a range search for a rectangular region in a 2-dimensional database,, implemented with PLOP hashing.

6. Give an algorithm for computing the value of MDEH for an arbitrary partition order rather than a cyclic one. In other words, we do not have to cycle thourgh the keys. For example, for three-dimensional data a cyclic partitioning order is $zyxzyxzyx \ldots zyx$, while $zzxyzxxxxyy \ldots$ is an example of an arbitrary order. Of course, we still have the requirement that the number of partitions along each key's axis doubles for each subsequent partitioning along that key.

7. Give an algorithm for computing the value of MDEH for an arbitrary partition order rather than a cyclic one as in Exercise 6 that permits partial expansions.

8. How would you completely generalize the implementation of PLOP hashing so that there is no need to complete a set of partitions along a key before starting to partition another key. In other words, the partitions on the keys can be sequenced arbitrarily without requiring that the number of grid cells be doubled for a key before a partition can be made on a different key. Moreover, there is no need to cycle through the keys.

9. Give an algorithm for merging grid cells in a 2-dimensional database of points implemented with quantile hashing.

10. One possible rationale for splitting all grid cells from a slice before splitting grid cells from another slice (as is the case when using the MDEH function) is that the grid cells in the slice are split in such an order that if grid cell c_i is split immediately before grid cell c_{i+1}, then c_i and c_{i+1} are spatially contiguous and hence more likely to be full. Does the spatial contiguity property always hold for successive splits of grid cells within a slice?

11. Suppose that spiral hashing is used to store two-dimensional point data, and each point is mapped to a value by use of bit interleaving. Prove that each bucket will span at most two spatially noncontiguous regions.

12. Suppose that spiral hashing is used to store point data of arbitrary dimension, and each point is mapped to a value by use of bit interleaving. What is the maximum number of spatially noncontiguous regions that can be spanned by a bucket?

13. Can you come up with a mapping for multidimensional point data to a value so that when it is used in conjunction with spiral hashing each bucket will span just one spatially contiguous region?

14. Show the result of applying spiral hashing to the database of Figure 1 using reversed bit interleaving. Thus, instead of using the hashing function $h(k) = k/2^s$ where s is the total number of bits that make up the interleaved key value, use the function $h''(k) = reverse(k)/2^s$, in which *reverse* reverses the value of key k prior to the application of the hashing function. Is there a difference in this example between using the value of the x coordinate as the most significant or that of the y coordinate?

15. Is there any advantage to using reversed bit interleaving with spiral hashing?

7.2.5 Comparison

Both the grid file and EXCELL guarantee that a record can be retrieved with two disk accesses: one for the d-dimensional grid directory element that spans the record (i.e., the grid cell in which its corresponding point lies) and one for the bucket in which it is stored. This is not necessarily the case for linear hashing methods (as well as its variants including spiral hashing). The problem is that although linear hashing methods often dispense with the need for a directory[49], the possible presence of an unbounded number of overflow buckets means that each one must be accessed during an unsuccessful search for a record corresponding to a point that is spanned by a particular grid cell.

Although, for pedagogical purposes, the grid directories of Figures 69 and 74 were constructed in such a way that EXCELL required less grid cells than the grid file (8 vs. 9), this is not generally the case. In fact, splitting points can always be chosen so that the number of grid cells and buckets, required by the grid file is less than or equal to the number required by EXCELL. For example, if the space of Figure 69 were split at $x = 55$ and $y = 37$, then only four grid cells and buckets would be necessary.

Interestingly, EXCELL can be characterized as a method that decomposes the underlying space into buckets of grid cells where the space spanned by the buckets corresponds to blocks in a bintree decomposition of space (see Section **??** of Chapter **??**), while the buckets corresponding to the individual grid cells are accessed by a directory in the form of an array. This decomposition of space is also identical to what is obtained by the bucket PR k-d tree. It is this close relationship between these methods that causes EXCELL to be classified as a representation that employs regular decomposition. In particular, the buckets in EXCELL are created by a halving process.

The main difference between EXCELL and these representations is that the directory of EXCELL corresponds to the deepest level of a complete bintree or bucket PR k-d tree. Alternatively, we can say that EXCELL

[49] We use the qualifier *often* because although linear hashing methods dispense with the need for a d-dimensional grid directory, a one-dimensional grid directory O may be needed to access the buckets associated with the grid cells unless there is a one-to-one correspondence between the numbering of the grid cells and the bucket labels in which case there is no need for O.

provides a directory to a space decomposition that is implemented as a bintree or bucket PR k-d tree. In particular, EXCELL results in faster access to the buckets since the k-d trie search step (i.e., following pointers in the tree) is avoided. However, if the longest search path in the tree has length s, then the EXCELL directory has 2^s entries.

From a superficial glance, EXCELL appears to be closely related to linear hashing when using OPLH with reversed bit interleaving as the hashing function. The reason for this observation is the fact that the space spanned by by the buckets in the two methods resembles blocks in a bintree decomposition of space. For example, compare the decomposition induced by EXCELL when the x coordinate value is split before the y coordinate value (shown in Figure 74) with the result of applying linear hashing to the data of Figure 1 when the x coordinate value is the most significant (shown in Figure 90).

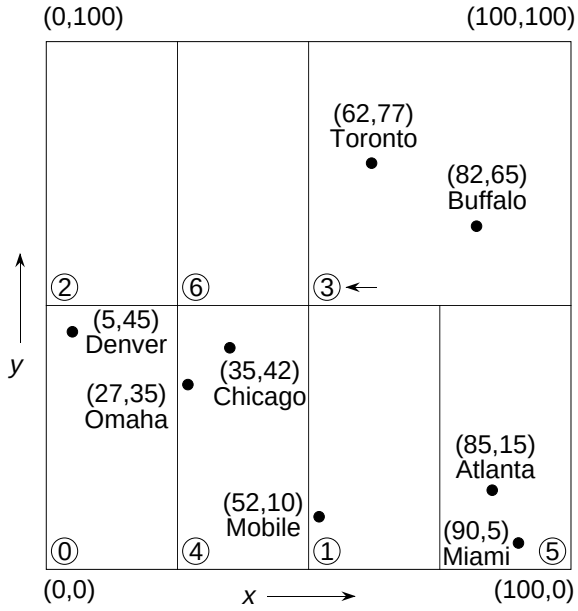


Figure 90: The representation resulting from applying linear hashing using OPLH with reversed bit interleaving corresponding to the data of Figure 1 when the x coordinate value is taken as the most significant.

Unfortunately, this similarity only exists when the buckets in EXCELL span no more than two grid cells. Recall that that there is a one-to-one correspondence between primary buckets and grid cells in linear hashing, and for OPLH all grid cells are either of size v or $v/2$. Greater variance in the volume of grid cells is not permitted for linear hashing with OPLH (but see linear hashing with MDEH as well as quantile hashing and PLOP hashing). For example, compare the decomposition induced by EXCELL when the y axis is split before the x axis (see Figure 91) with the result of applying linear hashing using OPLH with reversed bit interleaving to the data of Figure 1 when the y coordinate is the most significant (see Figure 79). Letting v be the size of the next grid cell to be split in the linear hashing example (i.e., Figure 79), we see that in the space decomposition induced by linear hashing all buckets span regions (i.e., grid cells) of size v or $v/2$, while in the space decomposition induced by EXCELL (i.e., Figure 91) we have buckets that span regions of size $2v$ (e.g., bucket `B`), $v/2$ (e.g., `A` and `F`), and $v/4$ (e.g., `C`, `D`, `E`, and `G`). Thus we see a bintree of four levels in the case of EXCELL while a bintree of just two levels in the case of linear hashing using OPLH with reversed bit interleaving.

Other differences are that in EXCELL a bucket overflow may trigger a bucket split or a doubling of the directory, whereas in linear hashing at most two new buckets are allocated (one for a bucket split and one for an overflow bucket — recall the insertion of `Omaha` in the transition from Figure 80c to 80d in Section 7.2.3). In essence, using linear hashing there is a more gradual growth in the size of the directory at the expense of the loss of the guarantee of record retrieval with two disk accesses. In addition, assuming equal bucket sizes and

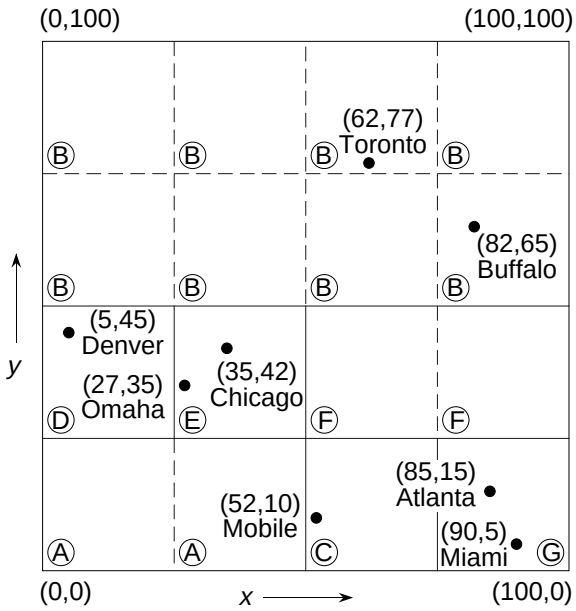


Figure 91: EXCELL representation corresponding to the data of Figure 1 when the y axis is split before the x axis.

ignoring the situation of a directory split, linear hashing usually requires additional buckets, since the bucket that is split is not necessarily the one that has overflowed.

Nevertheless, the relationship between EXCELL and linear hashing (using either OPLH or MDEH) is worthy of further comment. Disregarding our observations about the differing bucket sizes and just dealing with the grid cells that result from using EXCELL, a relationship does indeed exist. In particular, assuming an instance where linear hashing has applied all of the possible partitions for a specific key and no key is partially partitioned, we find that linear hashing (using either OPLH or MDEH) yields a decomposition of the underlying space U into grid cells which is equivalent to EXCELL without requiring a d-dimensional grid directory.

At a first glance, a similar statement can be made about the relationship of PLOP hashing to the grid file. In particular, PLOP hashing appears to yield a partition of the underlying space U into grid cells which is similar to that of the grid file without requiring a d-dimensional grid directory. Unfortunately, this statement is not quite true as PLOP hashing requires that the number of partitions along each key's axis doubles for each subsequent partitioning along that key and that during the time at which we partition along a particular key, we cannot instersperse a partition along a different key. Thus even though we do not have to cycle through the different keys, we must perform a complete partition along the key. In contrast, the grid file permits the partitions along the various keys to proceed in arbitrary order and allows an arbitrary number of partitions along each key's axis. In other words, key i may be partitioned once, followed by two partitions along key j, which is in turn followed, by five partitions along key i, etc.

A general problem that afflicts linear hashing, especially with OPLH (but also the other variants), is that the records may not be well-distributed in the buckets when the data is not uniformly distributed. This means that overflow will be more common than with traditional hashing methods. It also has the effect of possibly creating a large number of sparsely filled buckets. Consequently, random access is slower since, in the case of overflow, several buckets may have to be examined. Similarly, sequential access will also take longer as several buckets will have to be retrieved without finding too many records. The problem is that all of the variants of linear hashing require that all buckets be stored at one or two levels. This has the advantage that at most two hash operations are required to determine the primary bucket to be accessed (see Exercise **??** in Section **??** of Chapter **??**).

Orenstein [104] proposes a technique he terms *multi-level order preserving linear hashing (MLOPLH)* to be used with OPLH to deal with these problems. In particular, MLOPLH alleviates the sparseness problem by storing parts of the file in buckets at lower levels than n. Such buckets result from the combination of a sparse bucket with its brother(s). The major problem with MLOPLH is that now we may require $n + 1$ disk read operations (equal to the number of bucket accesses) to locate a bucket, whereas previously at most two hash operations were necessary.

The reason so many buckets may have to be accessed in MLOPLH is that linear hashing methods do not make use of a directory. Thus, encountering an empty bucket is the only way to detect that records of one bucket have been merged with those of another bucket at a lower level (see Exercise 3). Even more buckets will have to be accessed when deletion of a record causes a sparse bucket to be combined with its brother. In contrast, methods such as the grid file and EXCELL which use a directory do not suffer from such a problem to the same extent. The reason why the sparseness issue can be avoided is that the directory indicates the buckets associated with each grid cell, and several grid cells can share a bucket.

Exercises

1. Why does linear hashing usually require at least as many buckets as EXCELL?
2. Given a large set of multidimensional point data, which of the bucket methods would you use? Take the volume of the data and its distribution into account. Can you support your choice using analytic methods?
3. Explain how MLOPLH [104] looks up a value v.
4. Explain how MLOPLH splits a bucket.
5. Explain how MLOPLH merges two buckets.

8 Conclusion

A principal motivation for our discussion of the representation of collections of multidimensional point data is the desire to be able to distinguish between the individual elements of the collections so that they can be retrieved efficiently on the basis of the values of their attributes. The key to the retrieval process is the fact that the values of the attributes can be ordered (i.e., a sort sequence exists for them). We are not concerned with retrieval on the basis of values that cannot be ordered. The case of one-dimensional data was treated separately in Chapter **??**. Much of our presentation (for both one-dimensional and multidimensional point data) has differentiated the representations on the basis of whether they organize the data to be stored according to its value (termed *tree-based* earlier) or whether they organize the values of the embedding space from which the data is drawn (termed *trie-based* earlier). In either case, the effect is to decompose the space (i.e., domain) from which the data is drawn into groups. The representations are further differentiated on the basis of how the ranges of the boundaries of the spatial units corresponding to the groups are determined, the nature of the grouping process, and how the groups are accessed.

In this section we present a taxonomy that includes many of the representations that were described in this chapter. Some of those that are not explicitly discussed here are a subject of Exercises 1–5). To understand better their interrelationship, we make use of Figure 92, a tree whose nodes are different representations. Each node in the tree is labeled with the name of the corresponding representation and contains a drawing which is an approximation of how it represents the two-dimensional data of Figure 1. The nodes of the tree are also labeled with a classification indicating whether they organize the data based on its values (denoted by `D`), organize the embedding space from which the data is drawn (denoted by `E`), or neither organize the data nor the embedding space (denoted by `N`). We also makes use of an additional classification denoted by `H` indicating that the representation exhibits behavior that we associate with at least two of `D`, `E`, and `N`.

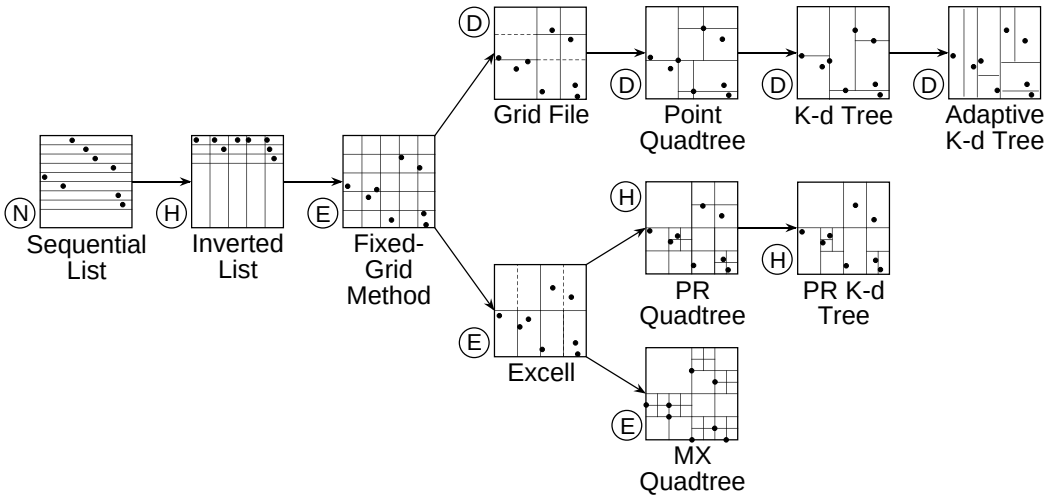


Figure 92: Tree representation of the interrelationship among the different data structures discussed in this Chapter using the data in Figure 1.

The depths of the nodes in the tree also convey important information on the relationship between the various data structures. We assume that the root of the tree is at depth 0. Our taxonomy reflects the fact that all of the representations are based on a decomposition of the underlying space into spatial elements termed *cells*. The tree reflects a progression from representations based on a fixed grid to ones based on an adaptive grid. The latter means that we have flexibility with respect to the size and shape of the cells. This progression also mirrors closely the transition from the use of access structures (i.e., directories) in the form of fixed grids (i.e., arrays) to trees in the sense that an adaptive grid usually requires some variant of a tree access structure. We make no special provision for the bucket methods that make use of a tree directory. In particular, we view them as natural methods of grouping more than one item in a cell. This is not the case for some of the bucket methods that make use of a grid directory which are shown in the tree in which case we use a bucket capacity of 2. When the access methods are trees, there are many choices for the type of the tree. We do not assume a particular choice for the tree.

Note that our characterizations are not necessarily unique. Other characterizations are undoubtedly equally valid (see Exercise 8), and we may have omitted some characterizations or data structures (see Exercise 5). Nevertheless, we have tried to place each of the data structures discussed in the previous sections in an appropriate position relative to the other representations.

Depth 0 represents no identifiable organization of the data. The data is maintained in a sequential list that usually corresponds to the order in which it was inserted into the database. In the example in Figure 92, we have drawn the sequential list so that the elements are in the same order from top to bottom as they appear in Figure 1. The position of the point in each row corresponds to the value of its x coordinate.

Depth 1 corresponds to a partition of space that only takes one key into consideration. The inverted list appears at this depth and is characterized as being hybrid. The key governing the partition exhibits behavior of type `D`. In contrast, the data values of the remaining keys have no influence on the way the space is organized and hence exhibit behavior of type `N`. In the example in Figure 92, we have assumed an inverted list for the key corresponding to the x coordinate. The position of the point in each entry corresponds to the value of its x coordinate. At this depth, we find the simplest types of cells.

At depth 2, all keys are taken into account resulting in a partition that yields a decomposition into a fixed number of cells. Although we do not do so here, we could distinguish between cells of equal size and of varying size. The fixed-grid method, type `E`, is an example of the former. There are a number of ways of realizing cells of varying size although they were not discussed in this chapter. For example, we could have a grid of cells of unequal size which would require an additional data structure such as linear scales to locate

the partition lines. Such a data structure would be of type D. Both this and the fixed-grid methods make use of a directory in the form of an array. An alternative is to use a recursive decomposition process to generate the partition into a fixed number of cells. An example of such a representation is the Dynamically Quantized Pyramid (DQP) [128, 107] which is a recursive decomposition into 2^d hyper-rectangular cells which need not be of equal size. The DQP requires a directory in the form of a tree and would also be of type D.

Depth 3 is a generalization of depth 2. It permits the cells of the representations at depth 2 to vary in number. The amount of variation depends on the volume of the data. The goal is not to fill a cell beyond its capacity. Thus the decomposition is adaptive. The fixed-grid method is refined to yield the grid file, type D, with cell boundaries that are allowed to vary, and EXCELL, type E, with cell boundaries at pre-determined positions. Notice that the representations at this level (as well as most of the ones at depth 2) use directories in the form of arrays to access the data. However, in the case of the grid file an additional access structure in the form of a tree is needed to determine the cell of the array that corresponds to a given point since the boundaries of the cells are free to vary. In addition, the representations at depth 3 enable the retrieval of a record in constant time.

The representations at depth 4 differ from those at depth 3 in that when a cell becomes too full (i.e., it overflows), then only that particular cell is partitioned. In contrast, assuming d-dimensional data, at depth 3 a partition is made over a $(d-1)$-dimensional cross section. This means that we can no longer make use of a directory in the form of an array. Instead, we make use of directories in the form of a tree. The partition in the representations at depth 4 results in a cell being decomposed into 2^d cells. At this depth, we find that the grid file is refined to yield the point quadtree of type D. EXCELL is refined to yield an MX quadtree of type E and a PR quadtree of type H. The PR quadtree is of type H because the partition points are fixed; however, the number of partition points depends on the relative position of the data points.

When a cell in one of the representations at depth 5 overflows, it is partitioned but it is only decomposed into two cells in contrast with 2^d for depth 4. The partitions that are made at the different levels in the relevant data structures cycle through the keys. At this depth, we find that the point quadtree is refined to yield a k-d tree of type D, and the PR quadtree is refined to yield a PR k-d tree of type H.

When a cell in one of the representations at depth 6 overflows, a partition is made using the key that has the greatest spread in values across the cell. Thus, we no longer need to cycle through the various keys as was done for the representations at depth 5. The adaptive k-d tree of type D is an example of this class of data structure.

The representations at depth 4–6 are usually implemented using access structures in the form of a tree with fanout 2^d for depth 4 and fanout 2 for depth 5 and 6. However, they can also be implemented by representing each of the cells by a unique number n which corresponds to the location of some easily identifiable point in the block (i.e., its lowe-right corner) and its size. n is stored in one of the one-dimensional access structures such as a binary search tree, B-tree, etc. along with the actual point that is associated with the cell. Such representations are discussed in greater detail in Sections **??** and **??** of Chapter **??**. Observe that use of the tree access structure results in an implicit aggregation of the space spanned by the cells that are being aggregated. This is in contrast to representations that aggregate the objects (i.e., points) that are stored in the cells in which case the space occupied by the points must be aggregated explicitly (e.g., by use of minimum bounding boxes as is done in the R-tree).

At this point, let us examine how some of the other representaions that we disccused fit into our taxonomy. Representations such as the range tree and the priority search tree (as well as the range priority tree), although not shown explicitly in Figure 92, belong at depth 1 as they correspond to the result of the imposition of an access structure on the inverted list. Recall that the inverted list is nothing more than a decomposition of the underlying space into one-dimensional cells. The one-dimensional range tree provides an access structure on just one attribute while the multidimensional range tree corresponds to the recursive application of range trees of one fewer attribute to range trees. The drawback of the multidimensional range tree is the high storage requirments since the data must be replicated. The priority search tree is like a range tree on one attribute (i.e., the one for which we hae the inverted list) with the addition of a heap on the remaining attribute.

Notice that, with the exception of the grid file and EXCELL, we did not include the bucket methods that

make use of a tree directory in our taxonomy. The reason is that the basic idea behind these bucket methods is the grouping of several items in a cell (e.g., the bucket PR quadtree) as well as applying the grouping to the cells (e.g., the k-d-B-tree, k-d-B-trie, LSD tree, hB-tree, BV-tree, VAMSplit trees, etc.). However, the basic idea is still the same in the sense that the principle behind the underlying organization is one of the representations at depths 4–6. Of course, the difference is that the internal nodes of the tree have a higher fanout.

It is important to observe that our classification (i.e., Figure 92) has not included any of the data structures that are used to support the one-dimensional orderings. This is to be expected as once the data has been ordered in this way (regardless of whether the attributes are locational or nonlocational), we are representing the ordering rather than the actual data. The elements of the ordering are not multidimensional points in a true sense. Nevertheless, they can be viewed as multidimensional points of dimension 1. In this case, their representation fits into some of the categories that we have defined. For example, binary search trees and B-trees would fit in the same classification as the point quadtree.

Notice also that our classification system did not contain any of the grid directory representations such as linear hashing, quantile hashing, PLOP hashing, and spiral hashing as they do not generally result in a new method of decomposing the underlying space. They are primarily implementations of the grid directory methods that provide an alternative in the form of a one-dimensional directory to the use of a d-dimensional directory as in the grid file and EXCELL. In addition, they have the important property that since the directory is one-dimensional, the directory can grow in a more gradual manner. In particular, the directory grows by one directory element (referred to as a grid cell) at a time instead of by doubling (as in EXCELL) or by the volume (in terms of the number of grid cells) of the $(d-1)$-dimensional cross-section perpendicular to the key whose axis is being partitioned (as in the grid file).

Actually, the above statement that the one-dimensional grid directory methods do not result in a different decomposition of the underlying space is only true for linear hashing (with reversed bit interleaving as the hashing function) where the underlying space is decomposed as in EXCELL, and for quantile hashing and PLOP hashing which decompose the underlying space as in the grid file. On the other hand, this statement is not true for spiral hashing which is probably the most radically different of these one-dimensional directory methods as it is the only method whose use does not result in decomposing the underlying space into hyper-rectangular d-dimensional cells. Moreover, not all the regions resulting from the decomposition are spatially contiguous.

Thus our failure to include linear hashing, quantile hashing, and PLOP hashing is justified, while spiral hashing does not seem to have any place in our classification as our entire discussion has been predicated on the fact that the underlying space is decomposed into hyper-rectangular d-dimensional cells. In retrospect, perhaps the right place to insert a family of methods that include spiral hashing is at a depth between 1 and 2 which would distinguish methods on the basis of the shape of the underlying cell (see also Exercise 6) as well as on whether or not the cells need to be spatially contiguous. It should be clear that at this point we have really only scratched the surface in our attempt at a classification. Many others are possible and desirable.

At this point, let us briefly comment on the generality of the representations that we have described. Our presentation and examples have been in terms of records all of whose attribute values were locational or numeric. As stated in the opening remarks of the Chapter, most of these representations can be generalized so that they are also applicable to records that contain additional nonlocational attributes, as well as only nonlocational attributes. Below, we briefly discuss how to achieve this generalization.

The tree-based methods are easily extended to to deal with nonlocational attributes. In this case, the partitions are made on the basis of the values of the attributes and make use of the ordering that is used for the attribute‘s value. Thus representations such as the point quadtree, k-d tree, adaptive k-d tree, as well as range trees and priority search trees can be used. This ordering can also serve as the basis of an inverted list as well as a grid file.

Extension of the trie-based methods to deal with nonlocational attributes is a bit more complex. The main issue is that many of the representations that we discussed are based on a binary subdivision of the underlying domain of the attributes, thereby resulting in trees with 2-way branches. In the case of the locational attributes, which are always numbers, this is quite simple as we use the representation of the values of the numeric at-

tributes as binary digits. When the nonlocational attribute values can be treated as sequences of characters with M possible values, then we have M-way branches instead of 2-way branches. Thus representations such as the PR quadtree and PR k-d tree are easily adapted to nonlocational attributes. Similarly, we can also adapt the fixed-grid method as well as EXCELL. In fact, the M possible values of each of the characters can also be viewed as being composed of $\lceil \log_2 M \rceil$ binary digits, and thus we can still use a 2-way branch on the ordering of the individual characters making up the attribute's value.

Note that the MX quadtree cannot be adapted to the general situation where we have locational and nonlocational attributes. The problem is that the MX quadtree is only appropriate for the case that all of the attribute values have the same type and range. Moreover, the domain of the attribute values must be discrete. Of course, if these criteria are satisfied, then an appropriate variant of the MX quadtree can be used for nonlocational attributes.

Exercises

1. How does the pseudo quadtree fit into the taxonomy of Figure 92?

2. How does the optimized point quadtree fit into the taxonomy of Figure 92?

3. How does the BD tree fit into the taxonomy of Figure 92?

4. How do the range tree, priority search tree, inverse priority search tree, and multidimensional range tree fit into the taxonomy of Figure 92?

5. The taxonomy shown in Figure 92 could be formulated differently. First, we qualify the `E` and `D` categories as being of type 1 or d depending on whether they take 1 or d dimensions into account. Also, let depth 3 be characterized as a refinement of depth 2 with overflow handling. Add another data structure, termed a *pyramid*, which yields a 2^d growth in the directory upon a bucket overflow. Now, depth 4 can be recast as replacing a directory in the form of an array by a directory in the form of a tree. How would you rebuild depths 3-6 of Figure 92 using this new taxonomy?

6. How would you revise the taxonomy of Figure 92 to account for the fact that the shape of the cells into which the underlying space is decomposed need not be hyper-rectangular (e.g., triangles, etc.)? In other words, include some decompositions that use these shapes. How does this generalize to dimensions higher than 2?

7. How would you revise the taxonomy of Figure 92 to account for the fact that the shape of the cells into which the underlying space is decomposed need not be hyper-rectangular as well as not be spatially contiguous (e.g., spiral hashing)?

8. (Markku Tamminen) In the opening section of this chapter, a number of issues was raised with respect to choosing an appropriate representation for point data. Can you devise a taxonomy in the same spirit as Figure 92 using them? For example, let each depth of the tree correspond to a different property. At depth 0, distinguish between an organization of the data and an organization of the embedding space. At depth 1, distinguish between dynamic and static methods. At depth 2, distinguish between the number of keys on which the partition is being made. At depth 3, distinguish between the presence and absence of a directory. At depth 4, distinguish between the types of directories (e.g., arrays, trees, etc.). At depth 5, distinguish between bucket and nonbucket methods.

9. The Euclidean matching problem consists of taking a set of $2 \cdot N$ points in the plane and decomposing it into the set of disjoint pairs of points so that the sum of the Euclidean distances between the components of each pair is a minimum. Would any of the representations discussed in this chapter facilitate the solution of this problem? Can you solve this problem by building a set of pairs, finding the closest pair of points, discarding it, and then applying the same procedure to the remaining set of points? Can you give an example that demonstrates that this solution is not optimal?

10. An alternative solution to the one proposed in Exercise 9 to the Euclidean matching problem proceeds in a manner analogous to that used in constructing the adaptive k-d tree in Section 5.1.4. In essence, we sort the points by increasing x and y coordinate values and then split them into two sets by splitting across the coordinate with the largest range. This process is applied recursively. Implement this method and run some experiments to show how close it comes to achieving optimality. Can you construct a simple example where the method of Exercise 9 is superior?

11. Suppose that you are given N points in a two-dimensional space, and a rectangular window, say R, whose sides are parallel to the coordinate axes. Find a point in the plane at which the window can be centered so that it contains a maximum number of points in its interior. The window is not permitted to be rotated — it can only be translated to the point. Would any of the representations discussed in this chapter facilitate the solution of this problem?

References

Each reference is followed by a key word(s). It is also followed by a list of the sections in which it is referenced. The format is D or A followed by the section number. D corresponds to this book while A correspond to [123]. D.P and A.P denote the appropriate preface and L denotes the appendix describing the pseudo-code language. All references that are cited in the solutions to the exercises are associated with the section in which the exercise is found.

[1] S. K. Abdali and D. S. Wise. Experiments with quadtree representation of matrices. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, page ?, Rome, July 1988. (Also University of Indiana Computer Science Technical Report No. 241). [matrices] D.4.2, D.8

[2] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, October 1983. [rectangles] D.6, A.2.1.1

[3] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. (English translation in *Soviet Math. Doklady* 3:1259–1263, 1962). [general] D.5.1

[4] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman. The *s*-tree: an efficient index for multidimensional objects. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases — Fifth International Symposium, SSD'97*, pages 350–373, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262). [spatial data structures; object hierarchies; R-trees; rectangles] D.7.1

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, Reading, MA, 1974. [general] D.5.1, A.P, A.4.5

[6] D. P. Anderson. Techniques for reducing pen plotting time. *ACM Transactions on Graphics*, 2(3):197–212, July 1983. [points] D.4.2, A.6.2

[7] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, Winter 1983. [points] D.8

[8] J. J. Bartholdi III and L. K. Platzman. A fast heuristic based on spacefilling curves for minimum-weight matching in the plane. *Information Processing Letters*, 17(4):177–180, November 1983. [points; minimal weight matching; space filling curves] D.8

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. [points] D.1, D.5, D.5.1, D.6, D.8, A.1.3

[10] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, June 1979. (see 1980 article with the same name). [points] D.1, D.2

[11] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, April 1980. [points] D.8

[12] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979. [points] D.1, A.1.3

[13] J. L. Bentley and H. A. Mauer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13:155–168, 1980. [rectangles] D.1, D.2

[14] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, December 1980. [points] D.8

[15] J. L. Bentley and D. F. Stanat. Analysis of range searches in quad trees. *Information Processing Letters*, 3(6):170–173, July 1975. [range queries; quad trees; search trees; points] D.4.1

[16] J. L. Bentley, D. F. Stanat, and E. Hollins Williams Jr. The complexity of finding fixed-radius near neighbors. *Information Processing Letters*, 6(6):209–212, December 1977. associative searching; nearest neighbors; points D.1, D.4.1

[17] T. Bestul. *Parallel paradigms and practices for spatial data*. PhD thesis, University of Maryland, College Park, MD, April 1992. [regions; points; lines; parallelism] D.5

[18] W. A. Burkhard. Interpolation-based index maintenance. *BIT*, 23(3):274–294, 1983. [points] D.7.2

[19] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986. (Also Systems Research Center Technical Report 12, June 1986). [general; binary search; B-tree; iterative search; multiple look-up; range query; dynamization of data structures] D.7.1

[20] B. Chazelle and L. J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1(2):163–191, 1986. (Also Systems Research Center Technical Report 12, June 1986). [general; fractional cascading; iterative search; multiple look-up; binary search; B-tree; range query; dynamization of data structures] D.7.1

[21] Y. Cohen, M. S. Landy, and M. Pavel. Hierarchical coding of binary images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(3):284–298, May 1985. [regions; adaptive algorithms for dynamic image sequences] D.5.2, A.1.3, A.2.2, A.8

[22] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979. [general; B-tree;B* -tree; B+ -tree; file organization; index] D.1, D.6, A.2.1, A.2.1.1, A.2.1.4, A.4.2.3

[23] S. P. Dandamudi and P. G. Sorenson. An empirical performance comparison of some variations of the k-d–tree and BD–tree. *International Journal of Computer and Information Sciences*, 14(3):135–159, June 1985. [points] D.5.2

[24] S. P. Dandamudi and P. G. Sorenson. Algorithms for BD–trees. *Software – Practice and Experience*, 16(12):1077–1096, December 1986. [points; multidimensional data structures; file structures; databases; multikey searching; partial match query; range query] D.5.2, D.8

[25] F. De Coulon and O. Johnsen. Adaptive block scheme for source coding of black-and-white facsimile. *Electronics Letters*, 12(3):61–62, February 1976. (Also erratum *Electronics Letters* 12(6):152, March 18, 1976). [regions] D.4.2, A.2.2

[26] L. DeFloriani, P. Marzano, and E. Puppo. Multiresolution models for topographic surface description. *The Visual Computer*, 12(7):317–345, August 1996. D.7.1

[27] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley Interscience, New York, 1973. [general] D.5.1

[28] H. Edelsbrunner. A note on dynamic range searching. *Bulletin of the EATCS*, (15):34–40, October 1981. [rectangles] D.1, D.3

[29] G. Evangelidis, D. Lomet, and B. Salzberg. Node deletion in the hB$^{\Pi}$-tree. Technical Report NU-CCS-94-04, Northeastern University, Boston, MA, 1994. [indexing; hB-tree; deletion; B-trees; multi-attribute access methods; spatial access methods; concurrency, recovery] D.7.1

[30] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995. [indexing; B-trees; multi-attribute access methods; spatial access methods; concurrency, recovery] D.7.1

[31] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6(1):1–25, 1997. [indexing; B-trees; multi-attribute access methods; spatial access methods; concurrency, recovery] D.7.1

[32] V. N. Faddeeva. *Computational Methods of Linear Algebra*. Dover, New York, 1959. [general] D.8

[33] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979. [points; hashing; extendible hashing; searching; index; file organization; radix search; access method; B-tree; trie; directory; external hashing] D.4.2, D.7.2

[34] C. Faloutsos. Multiattribute hashing using gray codes. In *Proceedings of the ACM SIGMOD Conference*, pages 227–238, Washington, DC, May 1986. [points] D.6

[35] C. Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, October 1988. [points] D.6, D.8

[36] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proceedings of the ACM SIGMOD Conference*, pages 426–439, San Francisco, May 1987. [rectangles] D.7, D.7.1

[37] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974. [points] D.1, D.4, D.4.1, D.8, A.1.3, A.1.4

[38] P. Flajolet and C. Puech. Tree structures for partial match retrieval. In *Proceedings of the Twenty-fourth Annual IEEE Symposium on the Foundations of Computer Science*, pages 282–288, Tucson, AZ, November 1983. [points] D.4.2, D.8

[39] W. R. Franklin. Adaptive grids for geometric operations. *Cartographica*, 21(2 & 3):160–167, Summer & Autumn 1984. [regions] D.1

[40] W. R. Franklin, N. Chandrasekhar, M. Kankanhalli, M. Seshan, and V. Akman. Efficiency of uniform grids for intersection detection on serial and parallel machines. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics: Proceedings of the CG International '88*, pages 288–297, Tokyo, Japan, 1988. Springer-Verlag. [regions] D.1

[41] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960. [general] D.0, A.1.3

[42] H. Freeman and J. Ahn. On the problem of placing names in a geographic map. *International Journal of Pattern Recognition and Artificial Intelligence*, 1(1):121–140, 1987. [name placement] D.5.2, D.7.1

[43] M. Freeston. The comparative performance of bang indexing for spatial objects. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, volume 1, pages 190–199, Charleston, SC, August 1992. [points; BANG file] D.7.1

[44] M. Freeston. A general solution of the n-dimensionla b-tree problem. Computer Science Department ECRC-94-40, ECRC, Munich, Germany, 1994. [points; BV-Tree] D.8

[45] M. Freeston. A general solution of the n-dimensional b-tree problem. In *Proceedings of the ACM SIGMOD Conference*, pages 80–91, San Jose, CA, May 1995. [points; k-d-b-trees; BV-tree] D.7.1, D.8

[46] M. W. Freeston. Advances in the design of the bang file. In W. Litwin and H. J. Schek, editors, *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms*, pages 322–338, Paris, June 1989. (Lecture Notes in Computer Science 367, Springer-Verlag, Berlin, 1989). [points] D.7.1

[47] M. W. Freeston. A well-behaved file structure for the storage of spatial objects. In A. Buchmann, O. Günther, T. R. Smith, and Y. F. Wang, editors, *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89*, pages 287–300, Santa Barbara, CA, July 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409). [region data; BANG file] D.7.1

[48] J. H. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24(10):1000–1006, October 1975. [points] D.1, D.8

[49] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977. [points] D.5.1

[50] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980. (Also *Proceedings of the SIGGRAPH'80 Conference*, Seattle, WA, July 1980). [volumes] D.5.1, A.1.3, A.7.1, A.7.1.5, A.7.3

[51] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982. [regions] D.6

[52] L. M. Goldschlager. Short algorithms for space-filling curves. *Software — Practice and Experience*, 11(1):99, January 1981. [regions; space-filling curves; Hilbert curve; Sierpinski curve] D.8, A.1.4

[53] M. F. Goodchild and A. W. Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proceedings of Auto-Carto 6*, volume 1, pages 400–407, Ottawa, October 1983. [regions] D.8, A.1.4

[54] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, Ann Arbor, MI, October 1978. [general] D.3

[55] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, MA, June 1984. [rectangles] D.7

[56] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 45–53, Amsterdam, The Netherlands, August 1989. [points; rectangles] D.7.1, D.8

[57] K. Hinrichs. *The grid file system: implementation and case studies of applications*. PhD thesis, Institut für Informatik, Zurich, Switzerland, 1985. [rectangles] D.7.2

[58] K. Hinrichs. Implementation of the grid file: design concepts and experience. *BIT*, 25(4):569–592, 1985. [rectangles] D.7.2

[59] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases — Fourth International Symposium, SSD'95*, pages 83–95, Portland, ME, August 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951). [points; lines; incremental nearest neighbor searching; ranking] D.0, D.4.1

[60] E. G. Hoel. *Spatial data structures and query performance in the sequential and data-parallel domains*. PhD thesis, University of Maryland, December 1995. (Also available as Technical Report TR-3584, University of Maryland, College Park, MD). [spatial databases] D.5

[61] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Princeton University, Princeton, NJ, 1978. [regions] A.1.3 A.1.4 A.3.2.2 A.4.3.1 A.4.4 A.5.1.2 A.5.2 A.5.3 A.6.3 A.6.3.2 A.6.5.1 A.7.1.1 A.7.1.2 A.9.2 D.4.2, A.1.2

[62] G. M. Hunter. Geometrees for interactive visualization of geology: an evaluation. System science department, Schlumberger-Doll Research, Ridgefield, CT, 1981. [volumes] D.4.2, A.1.3

[63] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, April 1979. [regions] A.4.4 A.5.1.2 A.5.2 A.5.3 A.6.3 A.6.3.2 A.6.5.1 A.7.1.1 A.7.1.2 A.9.2 D.4.2, A.3.2.2, A.4.3.1

[64] A. Hutflesz, H. W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *Proceedings of the Fourth IEEE International Conference on Data Engineering*, pages 572–579, Los Angeles, February 1988. [points; linear hashing] D.7.2, D.8

[65] A. Hutflesz, H. W. Six, and P. Widmayer. The twin grid file: a nearly space optimal index structure. In *Proceedings of the International Conference Extending Database Technology*, pages 352–363, Venice, Italy, March 1988. [points] D.7.2

[66] A. Hutflesz, H. W. Six, and P. Widmayer. Twin grid files: space optimizing access schemes. In *Proceedings of the ACM SIGMOD Conference*, pages 183–190, Chicago, IL, June 1988. [points] D.7.2

[67] M. Iri, K. Murota, and S. Matsui. Linear-time approximation algorithms for finding the minimum-weight perfect matching on a plane. *Information Processing Letters*, 12(4):206–209, August 1981. [points; approximation algorithms; minimum-weight matching; computational complexity; graphics] D.8

[68] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, Washington, DC, November 1993. [R-trees] D.7.2

[69] M. L. Kersten and P. van Emde Boas. Local optimizations of QUAD trees. Technical Report IR-51, Free University of Amsterdam, Amsterdam, The Netherlands, June 1979. [points] D.8

[70] G. D. Knott. Expandable open addressing hash table storage and retrieval. In *Proceedings of SIGFIDET Workshop on Data Description, Acess and Control*, pages 187–206, San Diego, November 1971. [points] D.7.1

[71] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973. D.3

[72] D. E. Knuth. *The Art of Computer Programming vol. 1, Fundamental algorithms*. Addison-Wesley, Reading, MA, second edition, 1973. [general] A.3.2.1.2 D.4.1, D.7.2, A.P, A.1.4, A.2.1.4

[73] D. E. Knuth. *The Art of Computer Programming vol. 3, Sorting and Searching*. Addison-Wesley, Reading, MA, 1973. [general] D.2.8.2.1 D.2.8.2.2 A.P A.1.3 D.0, D.1, D.4.1, D.5.1, D.5.2, D.8

[74] H. P. Kriegel and B. Seeger. Multidimensional order preserving linear hashing with partial expansions. In *Proceedings of the International Conference on Database Theory*, pages 203–220, Rome, September 1986. (Lecture Notes in Computer Science 243, Springer-Verlag, New York, 1986). [points] D.7.2, D.8

[75] H. P. Kriegel and B. Seeger. Multidimensional dynamic quantile hashing is very efficient for nonuniform record distributions. In *Proceedings of the Third IEEE International Conference on Data Engineering*, pages 10–17, Los Angeles, February 1987. [points] D.7.2

[76] H. P. Kriegel and B. Seeger. PLOP–hashing: a grid file without directory. In *Proceedings of the Fourth IEEE International Conference on Data Engineering*, pages 369–376, Los Angeles, February 1988. [points] D.7.2

[77] H. P. Kriegel and B. Seeger. Multidimensional quantile hashing is very efficient for nonuniform distributions. *Information Sciences*, 48(2):99–117, July 1989. [points] D.7.2

[78] R. Krishnamurthy and K. Y. Whang. Multilevel grid files. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1985. [points] D.7.1

[79] P.Å. Larson. Linear hashing with partial expansions. In F. H. Lochovsky and R. W. Taylor, editors, *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 224–232, Montreal, October 1980. [points] D.7.2

[80] P.Å. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988. [points] D.7.2

[81] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976. [general] D.8

[82] D. T. Lee. Maximum clique problem of rectangle graphs. In F. P. Preparata, editor, *Advances in Computing Research*, volume 1, pages 91–107. JAI Press, Greenwich, CT, 1983. [rectangles] D.8

[83] D. T. Lee and B. J. Shachter. Two algorithms for constructing a delaunay triangulation. *International Jounal of Computer and Information Sciences*, 9(3):219–242, June 1980. [regions; Delaunay triangulation; triangulation; divide-and conquer; Voroni tessellation; computational geometry; analysis of algorithms] D.5.1

[84] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977. [points] D.4.1, D.5.1, D.8

[85] P. Letellier. *Transmission d'images à bas débit pour un système de communication téléphonique adateé aux sourds*. PhD thesis, Université de Paris-Sud, Paris, September 1983. [regions] D.4.2

[86] J. Linn. General methods for parallel searching. Digital Systems Laboratory 81, Stanford University, Stanford, CA, May 1973. [points] D.5

[87] W. Litwin. Linear hashing: A new tool for file and table addressing. In F. H. Lochovsky and R. W. Taylor, editors, *Proceedings of the Sixth International Conference on Very Large Data Bases (VLDB)*, pages 212–223, Montreal, October 1980. [points] D.7.2

[88] D. Lomet and B. Salzberg. A robust multi-attribute search structure. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 296–304, Los Angeles, February 1989. [points; hB-tree] D.5.2

[89] D. Lomet and B. Salzberg. The hB–tree: a multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990. (Also Northeastern University Technical Report NU-CCS-87-24). [points] D.5.2, D.7.1, D.8

[90] G. N. N. Martin. Spiral storage: incrementally augmentable hash addressed storage. Department of Computer Science, Theory of Computation 27, University of Warwick, Coventry, Great Britain, March 1979. [points] D.7.2

[91] T. Matsuyama, L. V. Hao, and M. Hagao. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing*, 26(3):303–318, June 1984. [points] D.4.2, D.5.1

[92] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985. [rectangles] D.1, D.3

[93] T. H. Merrett and E. J. Otoo. Dynamic multipaging: a storage structure for large shared data banks. In P. Scheuermann, editor, *Improving Database Usability and Responsiveness*, pages 237–254. Academic Press, New York, 1982. [points; access methods; dynamic files; multiattribute storage structures; multipaging; order-preserving; key-to-key address transformations; tidy functions] D.7.2, D.8

[94] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. IBM Ltd., Ottawa, Canada, 1966. [regions] D.6, A.1.4, A.2.1.1, A.4.1

[95] J. K. Mullin. Spiral storage: efficient dynamic hashing with constant performance. *Computer Journal*, 28(3):330–334, August 1985. [points] D.7.2

[96] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. (Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, August 1986). [lines] D.5.2, A.1.3

[97] R. C. Nelson and H. Samet. A population analysis of quadtrees with variable node size. Computer Science TR–1740, University of Maryland, College Park, MD, December 1986. [points] D.4.2, D.5.2

[98] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, pages 270–277, San Francisco, May 1987. [points] D.4.2, D.5.2

[99] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984. [points; file structures; database; dynamic storage allocation; multikey searching; multidimensional data] D.1, D.7.2

[100] Y. Ohsawa and M. Sakauchi. The BD-Tree – a new n-dimensional data structure with highly efficient dynamic characteristics. *Information Processing*, pages 539–544, 1983. [points; new data structure; BD-Tree; suitable for geometrical database retrievals] D.5.2, D.7.1

[101] Y. Ohsawa and M. Sakauchi. Multidimensional data management structure with efficient dynamic characteristics. *Systems, Computers, Controls*, 14(5):77–87, 1983. (translated from, *Denshi Tsushin Gakkai Ronbunshi 66–D*, 10(October 1983), 1193–1200). [points] D.5.2, D.7.1

[102] B. C. Ooi, K. J. McDonell, and R. Sacks-Davis. Spatial *k–d*–tree: an indexing mechanism for spatial database. In *Proceedings of the Eleventh International Computer Software and Applications Conference COMPSAC*, pages 433–438, Tokyo, October 1987. [rectangles] D.5.1

[103] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982. [points; associative searching; partial matching query; range query; relation; tree] D.1, D.5.2, D.7.2, D.8, A.1.3

[104] J. A. Orenstein. A dynamic hash file for random and sequential accessing. In M. Schkolnick and C. Thanos, editors, *Proceedings of the Ninth International Conference on Very Large Data Bases (VLDB)*, pages 132–141, Florence, October 1983. [points] D.7.2

[105] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT–SIGMOD Symposium on Principles of Database Systems*, pages 181–190, Waterloo, Canada, April 1984. [points] D.6, D.7.2, A.1.4

[106] J. O'Rourke. Dynamically quantized spaces for focusing the hough transform. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 737–739, Vancouver, August 1981. [points; Hough transform; dynamically quantized space; hierarchical organization] D.5.1

[107] J. O'Rourke and K. R. Sloan Jr. Dynamic quantization: two adaptive data structures for multidimensional spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(3):266–280, May 1984. [points; Accumulator arrays; dynamic data structures; dynamics quantization; hierarchical data structures; Hough transform; k-d trees; multidimensional data structures; multidimensional histograms; pyramids] D.5.1, D.8

[108] E. J. Otoo. A mapping function for the directory of multidimensional extendible hashing. In U. Dayal, G. Schlageter, and L. H. Seng, editors, *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 493–506, Singapore, August 1984. [linear hashing; grid file; PLOP hashing] D.7.2, D.8

[109] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 90–105, Atlanta, March 1983. [points] D.7.2

[110] M. H. Overmars. Geometric data structures for computer graphics: an overview. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 21–49. Springer–Verlag, Berlin, Germany, 1988. [rectangles] D.8

[111] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and *k–d* trees. *Acta Informatica*, 17(3):267–285, 1982. [points] D.4.1, D.5.1, D.8

[112] G. Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890. [regions] D.6, A.1.4

[113] M. Regnier. Analysis of grid file algorithms. *BIT*, 25(2):335–357, 1985. [points; grid file analysis] D.4.2, D.8

[114] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice–Hall, Englewood Cliffs, NJ, 1977. [general] D.6

[115] E. M. Reingold and R. E. Tarjan. On the greedy heuristic for complete matching. *SIAM Journal on Computing*, 10(4):676–681, November 1981. [points; graph algorithms; matching; greedy heuritic; analysis of algorithms] D.8

[116] J. T. Robinson. The *k–d–b*–tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981. [points] D.6, D.7, D.7.1

[117] A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber. Application of hierarchical data structures' to geographical information systems: phase ii. Computer Science Department TR–1327, University of Maryland, College Park, MD, September 1983. [general] D.4.2, A.2.1, A.2.1.1

[118] B. Salzberg. *File structures: An analytic approach*. Prentice–Hall, Englewood Cliffs, NJ, 1988. [general] D.5.2

[119] B. Salzberg. On indexing spatial and temporal data. *Information Systems*, 19(6):447–465, September 1994. [points; temporal data; overview] D.7.1

[120] H. Samet. Deletion in *k*–dimensional quadtrees. (unpublished), 1977. [points] D.8

[121] H. Samet. Deletion in two–dimensional quad trees. *Communications of the ACM*, 23(12):703–710, December 1980. [points] D.4.1, D.8, A.7.1.5

[122] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984. [general] D.1, A.P

[123] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990. [general] D.4.2

[124] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990. [general] D.1, A.P, A.1

[125] B. Seeger. *Design and implementation of multidimensional access methods*. PhD thesis, University of Bremen, Bremen, Germany, 1989. (in German). [points] D.7.1, D.8

[126] B. Seeger and H. P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, pages 590–601, Brisbane, Australia, August 1990. [points] D.7.1, D.8

[127] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+–tree: a dynamic index for multi–dimensional objects. In P. M. Stocker and W. Kent, editors, *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 71–79, Brighton, United Kingdom, September 1987. (Also University of Maryland Computer Science TR–1795). D.7, D.7.1

[128] K. R. Sloan Jr. Dynamically quantized pyramids. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 734–736, Vancouver, August 1981. [points; pyramid; dynamically quantized pyramid; Hough transform] D.5.1, D.8

[129] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986. [rectangles] D.7, D.7.1

[130] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969. [matrices] D.8

[131] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, 1981. (Mathematics and Computer Science Series No. 34). [regions] D.7.2, A.1.3

[132] M. Tamminen. Order preserving extendible hashing and bucket tries. *BIT*, 21(4):419–435, 1981. [points] D.7.2

[133] M. Tamminen. Performance analysis of cell based geometric file organizations. *Computer Vision, Graphics, and Image Processing*, 24(2):168–181, November 1983. [regions; file organization schemes; efficient spatial access; geometric data; polygon network; fixed cell methods; address computation; extendible cell methods] D.4.2

[134] M. Tamminen. Comment on quad– and octtrees. *Communications of the ACM*, 27(3):248–249, March 1984. [regions; digital images; image encoding] D.5, A.1.3, A.2.1.2

[135] M. Tamminen. Efficient geometric access to a multirepresentation geo–database. *Geo–Processing*, 2:177–196, 1984. [lines] D.7.2

[136] M. Tamminen. On search by address computation. *BIT*, 25(1):135–147, 1985. [points] D.5.2

[137] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975. [regions] D.5.1, A.1.4, A.8.3

[138] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1984. [general] D.8

[139] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 23(2):71–77, February 1981. [points] D.6

[140] P. M. Vaidya. Geometry helps in matching. In *Proceedings of the Twentieth Annual ACM Symposium on the Theory of Computing*, pages 422–425, Chicago, IL, May 1988. [points] D.8

[141] V. K. Vaishnavi. Multidimensional height–balanced trees. *IEEE Transactions on Computers*, 33(4):334–343, April 1984. [points] D.5.1, D.8

[142] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978. [data structures; priority queues] D.3

[143] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, 1996. http://vision.ucsd.edu/papers/simret. [points; feature points; nearest neighbor search; similarity retrieval; medium and high dimensional indexing; approximate queries; image databases] D.5.1, D.7.1, D.8

[144] M. White. N-trees: large ordered indexes for multi-dimensional space. Statistical research division, US Bureau of the Census, Washington, DC, 1982. [points] D.6, D.8, A.1.4

[145] D. E. Willard. Balanced forests of *k–d* trees as a dynamic data structure. Aiken Computation Lab technical report TR–23–78, Harvard University, Cambridge, 1978. [points] D.5.1

[146] D. E. Willard. Polygon retrieval. *SIAM Journal on Computing*, 11(1):149–165, February 1982. [points] D.4.1, D.8

[147] D. S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20(4):195–199, May 1985. [matrices] D.8

[148] D. S. Wise. Matrix algebra and applicative programming. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture Theoretical Aspects of Computer Science*, pages 134–153, Portland, OR, 1987. (Also Lecture Notes in Computer Science 274, Springer–Verlag, Berlin, 1987). [matrices; algorithms] D.4.2

[149] D. S. Wise and J. Franco. Costs of quadtree representation of non–dense matrices. *Journal of Parallel and Distributed Computing*, 9(3):282–296, July 1990. (Also Indiana University Computer Science Technical Report No. 229, Bloomington, Indiana, October 1987). [matrices] D.4.2

[150] I. H. Witten and B. Wyvill. On the generation and use of space-filling curves. *Software — Practice and Experience*, 13(6):519–525, June 1983. [regions] D.8, A.1.4