CMSC 425: Lecture 8
Geometric Data Structures: Enclosures and Spatial Indices

Reading: This material has been collected from a number of difference sources. A good reference on geometric data structures is "Foundations of Multidimensional and Metric Data Structures," by H. Samet, 2006.

Geometric Objects and Queries: Someone once defined a *computer game* as a "database with a fun interface". Large games involve the storage and maintenance of a huge number of geometric objects, many of which change dynamically over time, and the game software needs to be able to access this information efficiently. Access to these structures takes form of *queries* (asking questions about the objects of the database) and *updates* (making changes to these objects).

What sorts of geometric queries might we be interested in asking? This depends a great deal about the application at hand. Queries typically involve determining what things are "close by." One reason is that nearby objects are more likely to have interesting interactions in a game (collisions or attacks). Of course, there are other sorts of interesting geometric properties. For example, in a shooting game, it may be of interest to know which other players have a line-of-sight to a given entity.

While we will focus on purely geometric data in this lecture, it is worth noting that geometry of an object may not be the only property of interest. For example, the query "locate all law enforcement vehicles within a half-mile radius of the player's car", might be quite reasonable for a car-theft game. Such queries involve both geometric properties (half-mile radius) and nongeometric properties (law enforcement). Such hybrid queries may involve a combination of multiple data structures.

Bounding Enclosures: When storing complex objects in a spatial data structure, it is common to first approximate the object by a simple enclosing structure. Bounding enclosures are often handy as a means of approximating an object as a filter in collision detection. If the bounding enclosures do not collide, then the objects to not collide. If they do, then we strip away the enclosures and apply a more extensive intersection test to the actual objects. Examples of bounding structures include:

Axis-aligned bounding boxes: This is an enclosing rectangle whose sides are parallel to the coordinate axes (see Fig. 1(a)). They are commonly called *AABBs* (axis-aligned bounding boxes). They are very easy to compute. (The corners are based on the minimum and maximum x- and y-coordinates.) An AABB can be represented by two points, for example, the lower-left point p^- and the upper-right point p^+. AABBs are preserved under translation, but not under rotation.

General bounding boxes: The principal shortcoming of axis-parallel bounding boxes is that it is not possible to rotate the object without recomputing the entire bounding box. In contrast, general (arbitrarily-oriented) bounding boxes can be rotated without the need to recompute them (see Fig. 1(b)).

A natural approach to represent such a box is to describe the box as an AABB but relative to a different coordinate frame. For example, we could define a frame whose

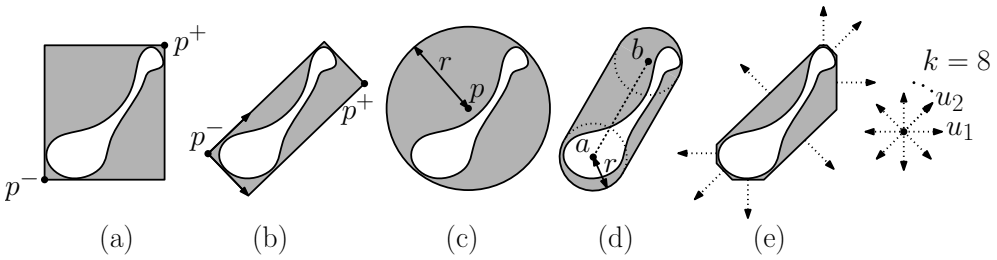


Fig. 1: Examples of common enclosures: (a) AABB, (b) general BB, (c) sphere, (d) capsule, (e) 8-DOP.

origin is one of the corners of the box and whose axes are aligned with the box's sides. By applying an appropriate affine transformation, we can map the general box to an AABB.

Computing the minimum bounding box is not simple. It will be the AABB for an appropriate rotation of the body, but determining the best rotation (especially in 3-space) is quite tricky.

Bounding spheres: These are among the most popular bounding enclosures. A sphere can be represented by a center point p and a radius r (see Fig. 1(b)). Spheres are invariant under rigid transformations, that is under translation and rotation. Unfortunately, they are not well suited to skinny objects.

Minimum bounding spheres are tricky to compute exactly. A commonly used heuristic is to first identify (by some means) a point p that lies near the center of the body, and then set the radius just large enough so that a sphere centered at p encloses the body. Identifying the point p is tricky. One heuristic is set p to be the center of gravity of the body. Another is to compute two points a and b on the body that are farthest apart from each other. This is called the *diametrical pair*. Define p to be the midpoint of the segment $\overline{ab}$.

Bounding ellipsoids: The main problem with spheres (and problem that also exists with axis-parallel bounding boxes) is that skinny objects are not well approximated by a sphere. A ellipse (or generally, an ellipsoid in higher dimensions) is just the image of a sphere under an affine transformations. As with boxes, ellipsoids may either be axis-parallel (meaning that the principal axes of the ellipse are parallel to the coordinate axes) or arbitrary.

As with spheres, minimum bounding ellipsoids are difficult to compute exactly. Various heuristics can be employed. For example, you can use the diametrical pair as defining the principal axis of the ellipse, but this is not generally optimal.

Capsules: This shape can be thought of as a "rounded cylinder." It consists of the set of points that lie within some distance r of a line segment $\overline{ab}$ (see Fig. 1(c)).

k-DOPs: People like objects bounded by flat sides, because the mathematics involved is linear. (There is no need to solve algebraic equations.) Unfortunately, an axis-aligned bounding box may not be a very close approximation to an object. (Imagine a skinny diagonal line segment.) As mentioned above, computing the minimum general bounding box may also be quite complex. A k-DOP is a compromise between these two.

Given an integer parameter $k \geq 3$ (or generally $k \geq d+1$), we generate k directional vectors that are roughly equally spaced and span the entire space. For example, in two-dimensional space, we might consider a set of unit vectors at angles $2\pi i/k$, for $0 \leq i < k$. Let $\{u_1, \ldots, u_k\}$ be the resulting vectors. We then compute extreme point of the object along each of these directions. We then put an orthogonal hyperplane through this point. The intersection of these hyperplanes defines a convex polygon with k sides (generally, a convex polytope with k facets) that encloses the objects. This is called a *k-discrete oriented polytope*, or *k-DOP* (see Fig. 1(d) and (e)).

Detecting Collisions: By enclosing an object within a bounding enclosure, collision detection reduces to determining whether two such enclosures intersect each other. Note that if we support k different types of enclosure, we need to handle all possible pairs of combinations of collisions. Here are a few examples:

AABB-AABB: We can test whether two axis-aligned bounding boxes overlap by testing that all pairs of intervals overlap. For example, suppose that we have two boxes b and b', where the box b extends from the lower-left corner (x_1, y_1) to the upper right corner (x_2, y_2) and b' extends from the lower-left corner (x'_1, y'_1) to the upper right corner (x'_2, y'_2) (see Fig. 2(a)). These boxes overlap if and only if

$$[x_1, x_2] \cap [x'_1, x'_2] \neq \emptyset \qquad \text{and} \qquad [y_1, y_2] \cap [y'_1, y'_2] \neq \emptyset$$

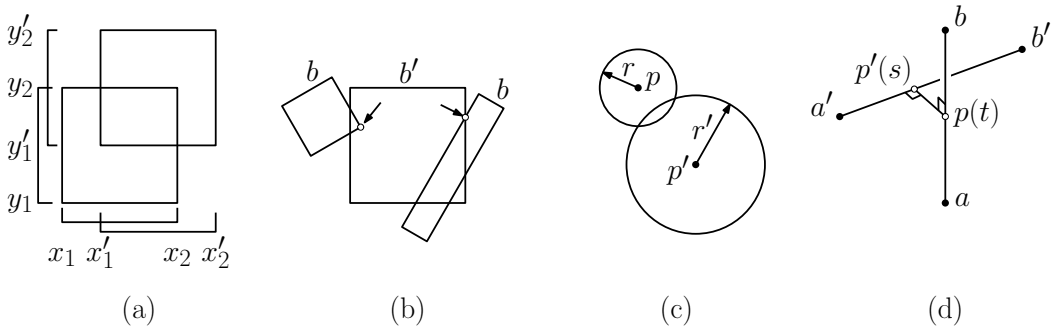


Fig. 2: Detecting collisions.

Box-Box: Determining whether two arbitrarily oriented boxes b and b' intersect is a non-trivial task. If they do intersect, one of the following must happen (see Fig. 2(b)):

- A vertex of b lies within b' or vice versa.
- An edge of b intersects a face of b' or vice versa.

One way to simplify these computations is to first compute a rotation that aligns one of the two boxes with the coordinate axes. The operations involved with determining point membership or edge-face intersection with an AABB is simpler than for general boxes. If both tests fail, we reverse the roles of the two boxes and try again.

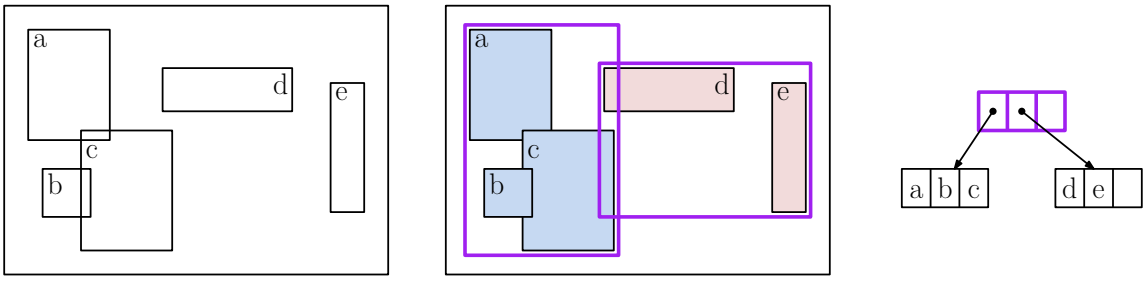
Sphere-Sphere: We can determine whether two spheres intersect by computing the distances between their centers. Given two spheres, one with center p and radius r and the other with center p' and radius r', they intersect if and only if $\text{dist}(p, p') \leq r + r'$ (see Fig. 2(c)).

Capsule-Capsule: Just as the sphere-sphere intersection reduces to computing the distances between the center points, the capsule-capsule intersection reduces to computing the distance between two line segments. In particular, given a capsule defined as the set of points that are within distance r of $\overline{ab}$ and the capsule defined as the set of points that are within distance r' of $\overline{a'b'}$, we would first compute the distance between the line segments $\overline{ab}$ and $\overline{a'b'}$. Assuming we can do this, the capsules intersect if and only if the shortest distance between the line segments is at most $r + r'$.

Here is a short sketch of how to compute the shortest distance between two line segments. We know that any point on the infinite $\overline{ab}$ can be expressed as an affine combination $p(t) = (1-t)a+tb$, where $0 \le t \le 1$. Similarly, any point on the line $\overline{a'b'}$ can be expressed as $p'(s) = (1-s)a' + sb'$. At the closest point between the two (infinite) lines, the line segment $\overline{p(t)p'(s)}$ must be perpendicular to both lines (see Fig. 1(d)). (Proving this is a simple exercise in Euclidean geometry.) Enforcing this perpendicularity condition involves solving a linear system of two equations and two unknowns s and t. We can easily solve the resulting 2×2 linear system of equations. We clamp the values of s and t to the interval $[0, 1]$, since all other points lie off the line segment. Once we know s and t, we can get the coordinates of the points easily, and once we have the coordinates we can get the distance.

Hierarchies of bounding bodies: What if the above bounding volumes are not sufficiently accurate for your purposes? A natural generalization is that of constructing a hierarchy consisting of multiple levels of bounding bodies, where the bodies at a given level enclose a constant number of bodies at the next lower level.

If you consider the simplest case of axis-aligned bounding boxes, the resulting data structure is called an *R-tree*. In Fig. 3 we given an example, where the input boxes are shown in (a), the hierarchy (allowing between 2–3 boxes per group) is shown in (b) and the final tree is shown in (c).



(a) (b) (c)

Fig. 3: A hierarchy of bounding boxes: (a) the input boxes, (b) the hierarchy of boxes, (c) the associated R-tree structure.

There are a number of interesting (and often quite complicated) technical issues in the design of R-trees. For example: What is the best way to cluster smaller boxes together to form larger boxes? How do you minimize the wasted space within each box? How do you minimize the overlap between boxes at a given level of the tree? As optimization problems, these are all

quite challenging. Usually, designers apply simple heuristic strategies to obtain good, albeit suboptimal, solutions.

This structure is widely used in the field of spatial databases, since the number of boxes contained within another box can be adjusted so that each node corresponds to a single disk block. (In this respect, it is analogous to the famous *B-tree* data structure for storing 1-dimensional data sets.)

Grids: One virtue of simple data structures is that are usually the easiest to implement, and (if you are fortunate in your choice of geometric model) they may work well. An example of a very simple data structure that often performs quite well is a simple rectangular grid. For simplicity, suppose that want to store a collection of objects. We will assume that the distribution of objects is fairly uniform. In particular, we will assume that there exists a positive real Δ such that almost all the objects are of diameter at most $c\Delta$ for some small constant c, and the number of objects that intersect any cube of side length Δ is also fairly small.

If these two conditions are met, then a square grid of side length Δ may be a good way to store your data. Here is how this works. First, for each of your objects, compute its axis-aligned bounding box. We can efficiently determine which cells of the grid are overlapped by this bounding box as follows. Let p and q denote the bounding box's lower-left and upper-right corners (see Fig. 4(a)). Compute the cells of the grid that contain these points (see Fig. 4(b)). Then store a pointer to the object in all the grid cells that lie in the rectangle defined by these two cells (see Fig. 4(c)). Note that this is not perfect, since the object may be associated with grid cells that it does not actually intersect. This increases the space of the data structure, but it does not compromise the data structure's correctness.

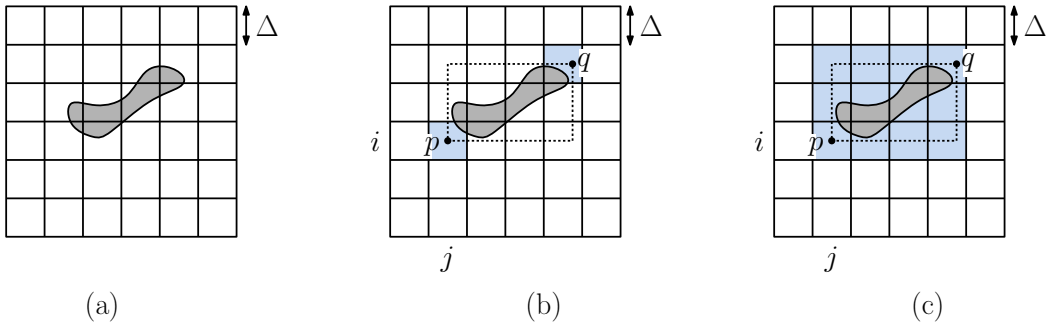


Fig. 4: Storing an object in a grid.

Computing the indices of the grid cell that contain a given point is a simple exercise in integer arithmetic. For example, if $p = (p_x, p_y)$, then let

$$j = \left\lfloor \frac{p_x}{\Delta} \right\rfloor \qquad \text{and} \qquad i = \left\lfloor \frac{p_y}{\Delta} \right\rfloor .$$

Then, the point p lies within the grid cell $G[i, j]$.

If the diameter of most of the objects is not significantly larger than Δ, then each object will only be associated with a constant number of grid cells. If the density of objects is not too

high, then each grid square will only need to store a constant number of pointers. Thus, if the above assumptions are satisfied then the data structure will be relatively space efficient.

Storing a Grid: As we have seen, a grid consists of a collection of cells where each cell stores a set of pointers to the objects that overlap this cell (or at least might overlap this cell). How do we store these cells? Here are a few ideas.

d-dimensional array: The simplest implementation is to allocate a d-dimensional array that is sufficiently large to handle all the cells of your grid. If the distribution of objects is relatively uniform, then it is reasonable to believe that a sizable fraction of the cells will be nonempty. On the other hand, if the density is highly variable, there may be many empty cells. This approach will waste space.

Hash map: Each cell is identified by its indices, (i, j) for a 2-dimensional grid or (i, j, k) for the 3-dimensional grid. Treat these indices like a key into a hash map. Whenever a new object o is entered into some cell (i, j), we access the hash map to see whether this cell exists. If not, we generate a new cell object and add it to the hash map under the key (i, j). If so, we add a pointer to o into this hash map entry.

Linear allocation: Suppose that we decide to adopt an array allocation. A straightforward implementation of the d-dimensional array will result in a memory layout according to how your compiler chooses to allocate arrays, typically in what is called *row-major order* (see Fig. 5(a)). For example, if there are N columns, then the element (i, j) is mapped to index $i \cdot N + j$ in row-major order.

Why should you care? Well, the most common operation to perform on a grid is to access the cells that surround a given grid square. Memory access tends to be most efficient when accessed elements are close to one another in memory (since they are likely to reside within the same cache lines). The problem with row-major order is that entries in successive rows are likely to be far apart in physical memory.

A cute trick for avoiding this problem is to adopt a method of mapping cells to physical memory that is based on a *space filling curve*. There are many different space-filling curves. We show two examples in Figs. 5(b) and (c). The first is called the *Hilbert curve* and the second is known as the *Morton order* (also called the *Z-order*).

There is experimental evidence that shows that altering the physical allocation of cells can improve running times moderately. Unfortunately, the code that maps an index (i, j) to the corresponding address in physical memory becomes something of a brain teaser.

Computing the Morton Order: Between the Hilbert order and the Morton order, the Morton order is by far the more commonly use. One reason for this is that there are some nifty tricks for computing the this order. To make this easier to see, let us assume that we are working in two-dimensional space and that the grid of size $2^m \times 2^m$. The trick we will show applies to any dimension. If your grid is not of this size, you can embed it within the smallest grid that has this property.

Next, since the grid has 2^m rows and 2^m columns, we can view each row and column index as an m-element bit vector, call them $i = \langle i_1, \ldots, i_m \rangle_2$ and $j = \langle j_1, \ldots, j_m \rangle_2$, in order from most significant bit (i_1) to the least significant bit (i_m). Next, we take these

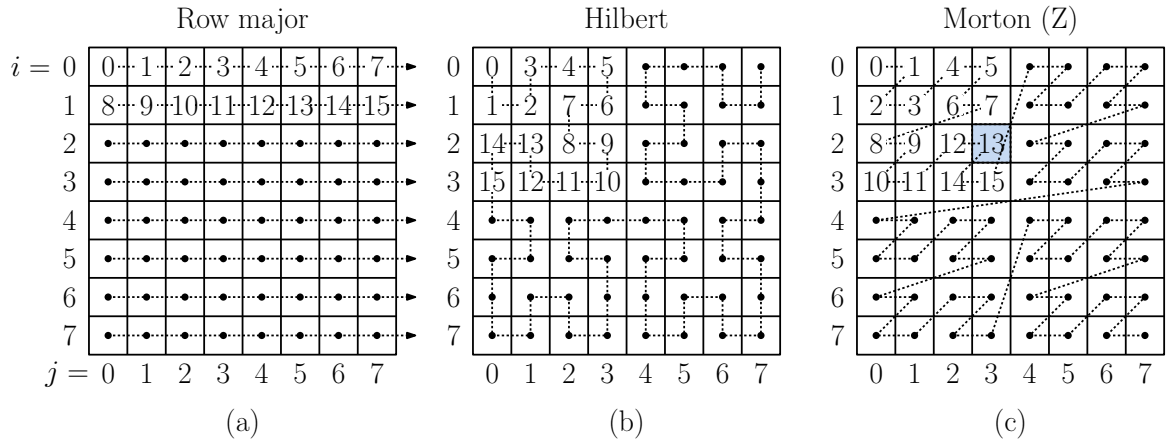


Fig. 5: Linear allocations to improve reference locality of neighboring cells: (a) row-major order, (b) the Hilbert curve, (c) the Morton order (or Z-order).

bit vectors and interleave them as if we were shuffling a deck of cards:

$$k \;=\; \langle i_1, j_1, i_2, j_2, \ldots, i_m, j_m \rangle_2.$$

If you have not seen this trick before, it is rather remarkable that it works. As an example, consider the cell at index $(i, j) = (2, 3)$, which is labeled as 13 in Fig. 5(c). Expressing i and j as 3-element bit vectors we have $i = \langle 0, 1, 0 \rangle_2$ and $j = \langle 0, 1, 1 \rangle_2$. Next, we interleave these bits to obtain

$$k \;=\; \langle 0, 0, 1, 1, 0, 1 \rangle_2 \;=\; 13,$$

just as we expected.

This may seem like a lot of bit manipulation, particularly if m is large. It is possible, however, to speed this up. For example, rather than processing one bit at a time, we could break i and j up into 8-bit bytes, and then for each byte, we could access a 256-element look-up table to convert its bit representation to one where the bits have been "spread out." (For example, suppose that you have the 8-element bit vector $\langle b_0, b_1, \ldots, b_7 \rangle_2$. The table look-up would return the 16-element bit vector $\langle b_0, 0, b_1, 0, \ldots, b_7, 0 \rangle_2$.) You repeat this for each byte, applying a 16-bit shift in each case. Finally, you apply an addition right shift of the j bit vector by a single position and bitwise "or" the two spread-out bit vectors for i and j together to obtain the final shuffled bit vector. By interpreting this bit vector as an integer we obtain the desired *Morton code* for the pair (i, j).

Quadtrees: Grids are fine if the density of objects is fairly regular. If there is considerable variation in the density, a quadtree is a practical alternative. You have probably seen quadtrees in your data structures course, so I'll just summarize the main points, and point to a few useful tips.

First off, the term "quadtree" is officially reserved for 2-dimensional space and "octree" for three dimensional space. However, it is too hard to figure out what the name should be when you get to 13-dimensional space, so I will just use the term "d-dimensional quadtree" for all dimensions.

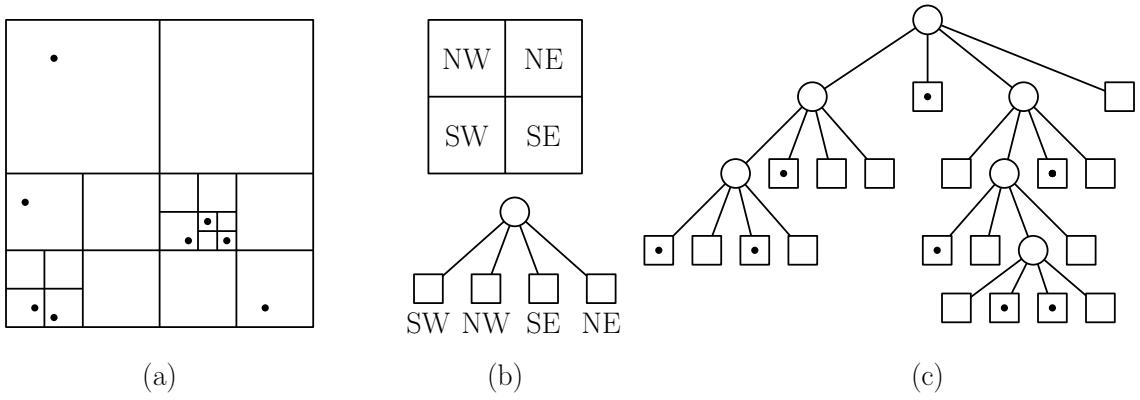


Fig. 6: A quadtree decomposition and the associated tree.

We begin by assuming that the domain of interest has been enclosed within a large bounding square (or generally a hypercube in d-dimensional space). Let's call this Q_0. Let us suppose that we have applied a uniform scaling factor so that Q_0 is mapped to the d-dimensional unit hypercube $[0,1]^d$. A *quadtree box* is defined recursively as follows:

- Q_0 is a quadtree box
- If Q is any quadtree box, then the 2^d boxes that result by subdividing Q through its midpoint by axis aligned hyperplanes is also a quadtree box.

This definition naturally defines a hierarchical subdivision process, which subdivides Q_0 into a collection of quadtree boxes. This defines a tree, in which each node is associated with a quadtree box, and each box that is split is associated with the 2^d sub-boxes as its children (see Fig. 6). The root of the tree is associated with Q_0. Because Q_0 has a side length of 1, it follows directly that the quadtree boxes at level k of the tree have side length $1/2^k$.

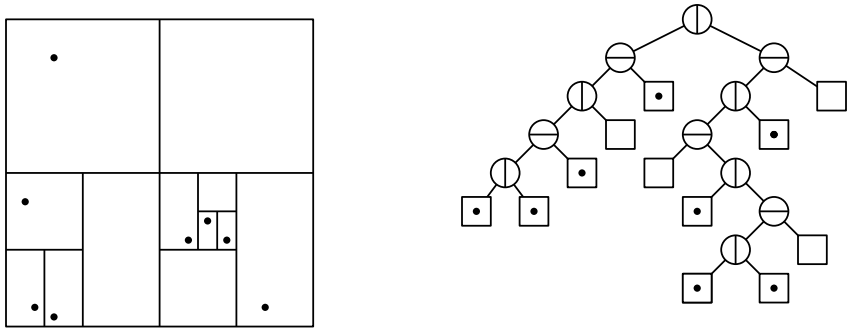
Quadtree variants: (Optional material)

There are a couple of practical variants of quadtrees that are worth knowing about. Here are a few. (See Samet's book for much more information.)

Binary Quadtrees: In dimension 3 and higher, having to allocate 2^d children for every internal node can be quite wasteful. Unless the points are uniformly distributed, it is often the case that only a couple of these nodes contain points. An alternative is rely only on binary splits. First, split along the midpoint x-coordinate, then the midpoint y-coordinate, and so forth, cycling through the axes (see Fig. 7).

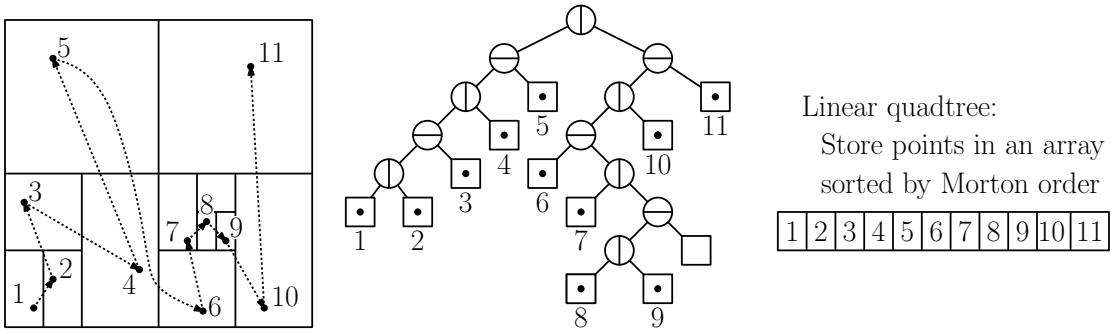
Linear Quadtree: A very clever and succinct method for storing quadtrees for point sets involves no tree at all! Recall the Morton order, described earlier in this lecture. A point (x, y) is mapped to a point in a 1-dimensional space by shuffling the bits of x and y together. This maps all the points of your set onto a space filling curve.

What does this curve have to do with quadtrees? It turns out that the curve visits the cells of the quadtree (either the standard, binary, or compressed versions) according to an in-order traversal of the tree (see Fig. 8).



(a) (b)

Fig. 7: A binary quadtree.



(a) (b) (c)

Fig. 8: A linear quadtree.

How can you exploit this fact? It seems almost unbelievable that this would work, but you sort all the points of your set by the Morton order and store them in an array (or any 1-dimensional data structure). While this would seem to provide very little useful structure, it is remarkable that many of the things that can be computed efficiently using a quadtree can (with some additional modifications) be computed directly from this sorted list. Indeed, the sorted list can be viewed as a highly compressed encoding of the quadtree.

The advantage of this representation is that it requires zero additional storage, just the points themselves. Even though the access algorithms are a bit more complicated and run a bit more slowly, this is a very good representation to use when dealing with very large data sets.

Kd-trees: While quadtrees are widely used, there are some applications where more flexibility is desired in how the object space is partitioned. There are a number of alternative index structures that are based on the hierarchically subdividing space into simple regions. Data structures based on such hierarchical subdivisions are often called *partition trees*. One of the most widely-used partition-tree structures is the kd-tree.

A *kd-tree*[1] is a partition tree based on orthogonal slicing. We start by assuming that all the points of our space are stored within some large bounding (axis-aligned) rectangle, which is associated with the root node of the tree. Every node of the tree is associated with a (hyper)-rectangular region, called its *cell*. Each internal node of the tree is associated with an axis-aligned *splitting plane*, which is used to split the cell in two. The points falling on one side are stored in one child and points on the other side are stored in the other. Each internal node t of the kd-tree is associated with the following quantities:

$t.cut\text{-}dim$	the cutting dimension (e.g., 0, 1, or 2 representing x, y, or z, respectively)
$t.cut\text{-}val$	the cutting value (a real number)

Of course, there generally may be additional information associated with each node (for example, the number of objects lying within the node's cell), depending on the exact application. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to $t.cut\text{-}val$ are stored in the left subtree, and the remaining points are stored in the right subtree (see Fig. 9). (If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side.) When a single point remains, we store it in a leaf node, whose only field $t.point$ is this point.

There are two key decisions in the implementation of the kd-tree.

How is the cutting dimension chosen? The simplest method is to cycle through the dimensions one by one. (This method is shown in Fig. 9.) Since the cutting dimension depends only on the level of a node in the tree, one advantage of this rule is that the

[1]The terminology surrounding kd-trees has some history. The data structure was proposed originally by Jon Bentley. In his notation, these were called "k-d trees," short for "k-dimensional trees" since the generalize classical binary trees for 1-dimensional data. Thus, there are 2-d trees, 3-d trees, and so on. However, over time, the specific value of k was lost, and they are simply called kd-trees.

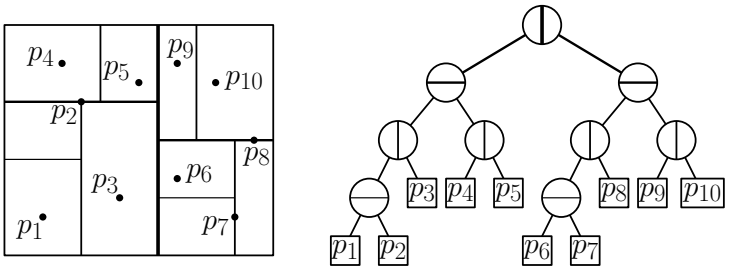


Fig. 9: A kd-tree and the associated spatial subdivision.

cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

One disadvantage of this splitting rule is that, depending on the data distribution, this simple cyclic rule may produce very skinny (elongated) cells, and such cells may adversely affect query times. Another method is to select the cutting dimension to be the one along which the points have the greatest *spread*, defined to be the difference between the largest and smallest coordinates. Bentley call the resulting tree an *optimized kd-tree*.

How is the cutting value chosen? To guarantee that the tree is balanced, that is, it has height $O(\log n)$, the best method is to let the cutting value be the *median coordinate value* along the cutting dimension. In our example, when there are an odd number of points, the median is associated with the left (or lower) subtree.

Note that a kd-tree is a special case of a more general class of hierarchical spatial subdivisions, called *binary space partition trees* (or *BSP trees*) in which the splitting lines (or hyperplanes in general) may be oriented in any direction, not just axis-aligned.