CMSC 420 ASSIGNMENT: A DATA STRUCTURE FOR COMPUTER GRAPHICS

Hanan Samet

Computer Science Department
University of Maryland
College Park, Maryland 20742

September 6, 2020

Abstract

In this assignment you are required to implement an information management system for handling data similar to that used in computer graphics applications. In such an environment the data is often in the form of a collection of lines which is used to define objects. Line segments can be combined to form polygons. We are interested in the problem of managing a large collection of polygons which we term a *polygonal map*. Our goal is to store these maps without the information loss that usually results from the digitization process that is inherent to a raster-based graphics system. In this assignment we first trace the development of a variant of the quadtree data structure that has been found to be suitable for such a problem. Next, we outline an assignment which you are to program. In particular, your task is to implement this data structure in such a way that a number of operations used in computer graphics can be efficiently handled. An example JAVA applet for the data structure can be found on the home page of the class.[1]

[1]Copyright©2020 by Hanan Samet. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the express prior permission of the author.

1 REGION-BASED QUADTREES

The quadtree is a member of a large class of hierarchical data structures that are based on the principle of recursive decomposition. As an example, consider the point quadtree of Finkel and Bentley [3] which should be familiar to you as it is simply a multidimensional generalization of a binary search tree. In two dimensions each node has four subtrees corresponding to the directions NW, NE, SW, and SE. Each subtree is commonly referred to as a quadrant or subquadrant. For example, see Figure 2 where the correspondence of a quadtree of 8 nodes to the data of Figure 1 is presented. In our presentation we shall only discuss two-dimensional quadtrees although it should be clear that what we say can be easily generalized to more than two dimensions. For the point quadtree, the points of decomposition are the data points themselves (e.g., in Figure 2, Chicago at location (35,40) subdivides the two dimensional space into four rectangular regions). Requiring the regions to be of equal size leads to a variant of the region quadtree of Klinger [5,6,8,9]. This data structure was developed for representing homogeneous spatial data and is used in computer graphics, image processing, geographical information systems, pattern recognition, and other applications. Its use in computer graphics is rooted in the work of Warnock [11] who applied it to hidden surface elimination. For a good introduction to computer graphics, see the texts by Foley, van Dam, Feiner, and Hughes [4] and Newman and Sproull [7].

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.

As an example of the region quadtree, consider the region shown in Figure 3a which is represented by a $2^3 \times 2^3$ binary array in Figure 3b. Observe that 1s correspond to picture elements (termed pixels) which are in the region and 0s correspond to picture elements that are outside the region. The region quadtree representation is based on the successive subdivision of the array into four equal-size quadrants. If the array does not consist entirely of 1s or 0s (i.e., the region does not cover the entire array), then we subdivide it into quadrants, subquadrants, ... until we obtain blocks (possibly single pixels) that consist entirely of 1s or entirely of 0s. For example, the resulting blocks for the region of Figure 3b are shown in Figure 3c. This process is represented by a quadtree in which the root node corresponds to the entire array, the four sons of the root node represent the quadrants, and the leaf nodes correspond to those blocks for which no further subdivision is necessary. Leaf nodes are said to be BLACK or WHITE depending on whether their corresponding blocks are entirely within or outside of the region respectively. All non-leaf nodes are said to be GRAY. The region quadtree for Figure 3c is shown in Figure 3d.

2 PR QUADTREES

There are a number of ways of adapting the region quadtree to represent point data. If the domain of data points is discrete, then we can treat data points as if they are BLACK pixels in a region quadtree. If this is not the case, then the data points cannot be represented since the minimum separation between the data points is unknown. This leads us to an adaptation of the region quadtree to point data which associates data points (that need not be discrete) with quadrants. In order to avoid confusion with the point and region quadtrees we call the resulting data structure a *PR quadtree* (P for point and R for region). 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 the values of its coordinates. A quadrant

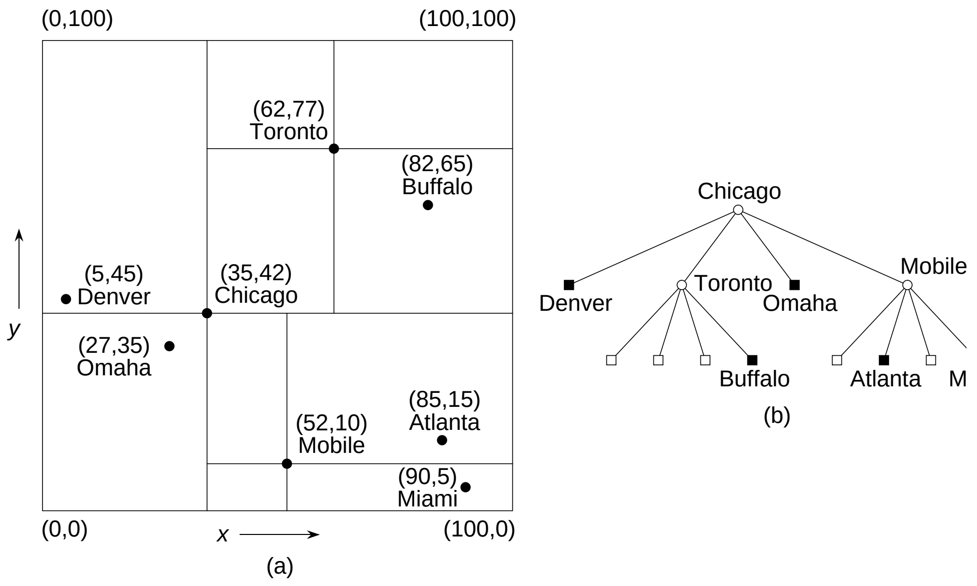


Figure 2: A point quadtree and the records it represents corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation.

contains at most one data point. For example, Figure 4 is the PR quadtree corresponding to the data of Figure 1. Note, that unlike the region quadtree, when a non-terminal node has 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. Recall that each data point is different whereas the empty leaf nodes have the absence of information as their common property and thus they can be safely merged.

2.1 INSERTION

Nodes in the PR quadtree consist of two types of records. The record's type depends on whether it corresponds to a terminal or non-terminal node. This information is stored in the NODETYPE field of the record. We use P to denote a pointer to the record. Non-terminal nodes have four fields in addition to the NODETYPE field. These four fields are termed SON(P,I) where I corresponds to one of the four principal directions. In addition to the NODETYPE field, terminal nodes have the following three fields. NAME contains descriptive information about the node (e.g., city name, etc.). XCOORD and YCOORD contain the x and y coordinate values, respectively, of the data point. We assume that each data point is unique - i.e., it is associated with just one name. If more than one data point were permitted to have the same coordinate values, then an overflow list would be required at each data point node. The empty PR quadtree is represented by a pointer to a WHITE node. .PR In order to cope with data points that lie directly on one of the quadrant lines emanating from a subdivision point, we adopt the convention that quadrants NE and SE are closed with respect to the x coordinate and quadrants NW and NE are closed with respect to the y coordinate. This means that quadrants NW and SW are open with respect to the x coordinate and quadrants SW and SE are open with respect to the y coordinate. For example, in Figure 4, Mobile with coordinate values (50,10) lies in quadrant SE of the tree rooted at (50,50). Using this convention, the function PRCOMPARE, given below, determines the quadrant in which a given data point lies relative to a grid subdivision point.

Data points are inserted into a PR quadtree by searching for them. First, an appropriate record is formed for the data point. Next, we search for the desired record based on its x and y coordinate values using the function PRCOMPARE to guide the search. If a record with the same x and y coordinate values already exists, then we replace it with the new record. Actually we don't search for the record. Instead, we search for the quadrant in which the record, say A, belongs (i.e., a leaf node). If the quadrant is already occupied

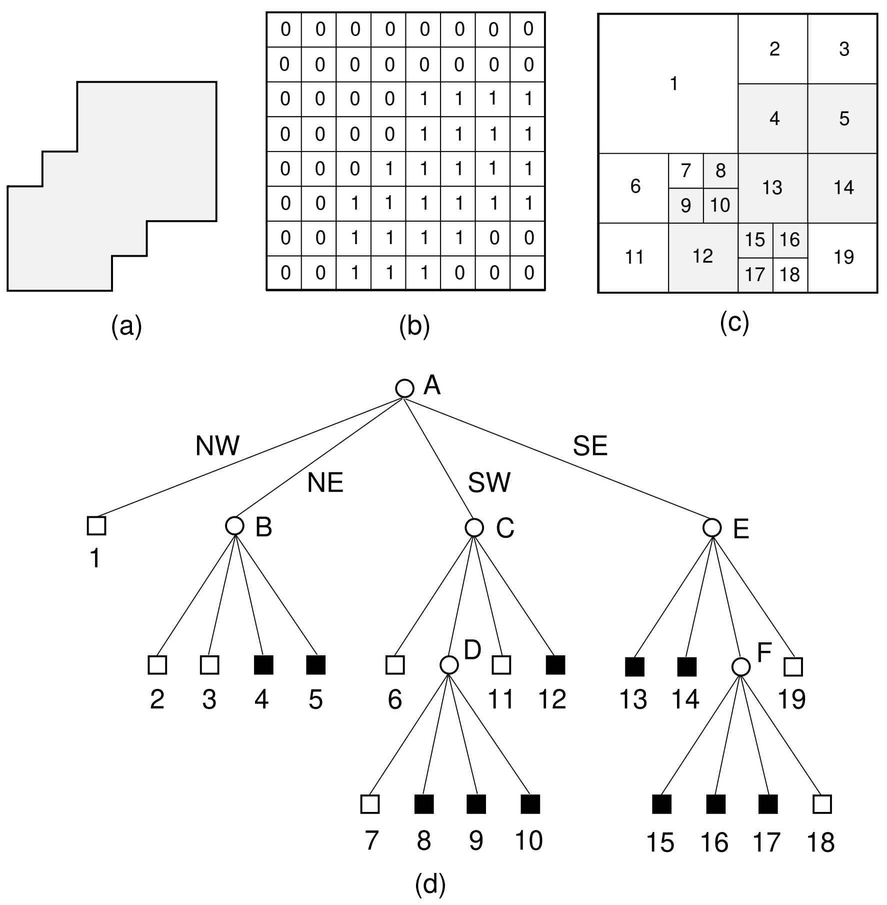


Figure 3: An example of (a) a region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding tree.

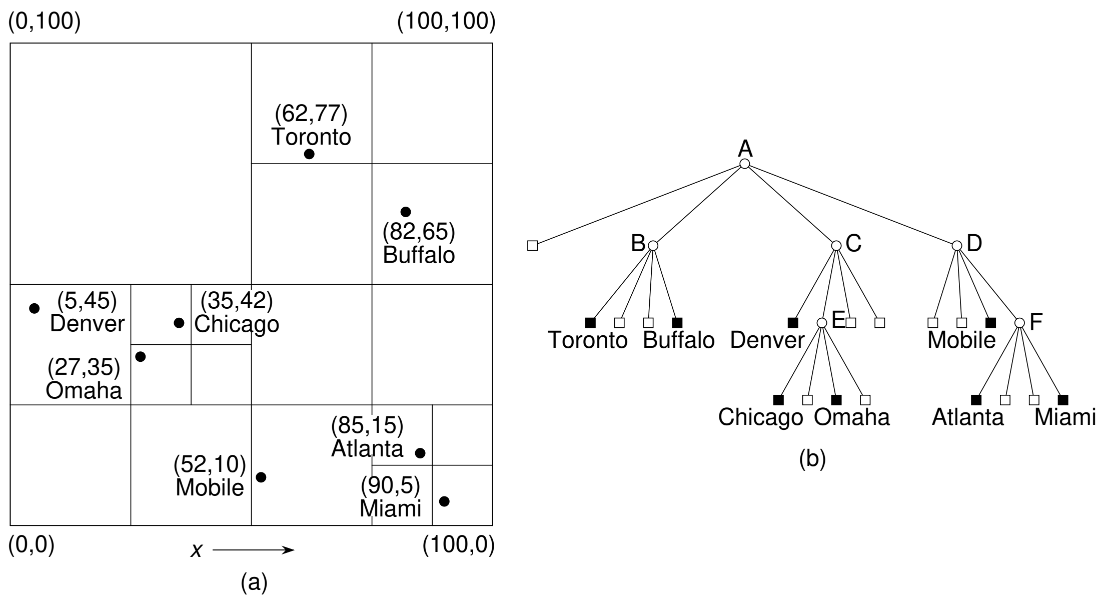


Figure 4: A PR quadtree and the records it represents corresponding to Figure 1: (a) the resulting partition of space, and (b) the tree representation.

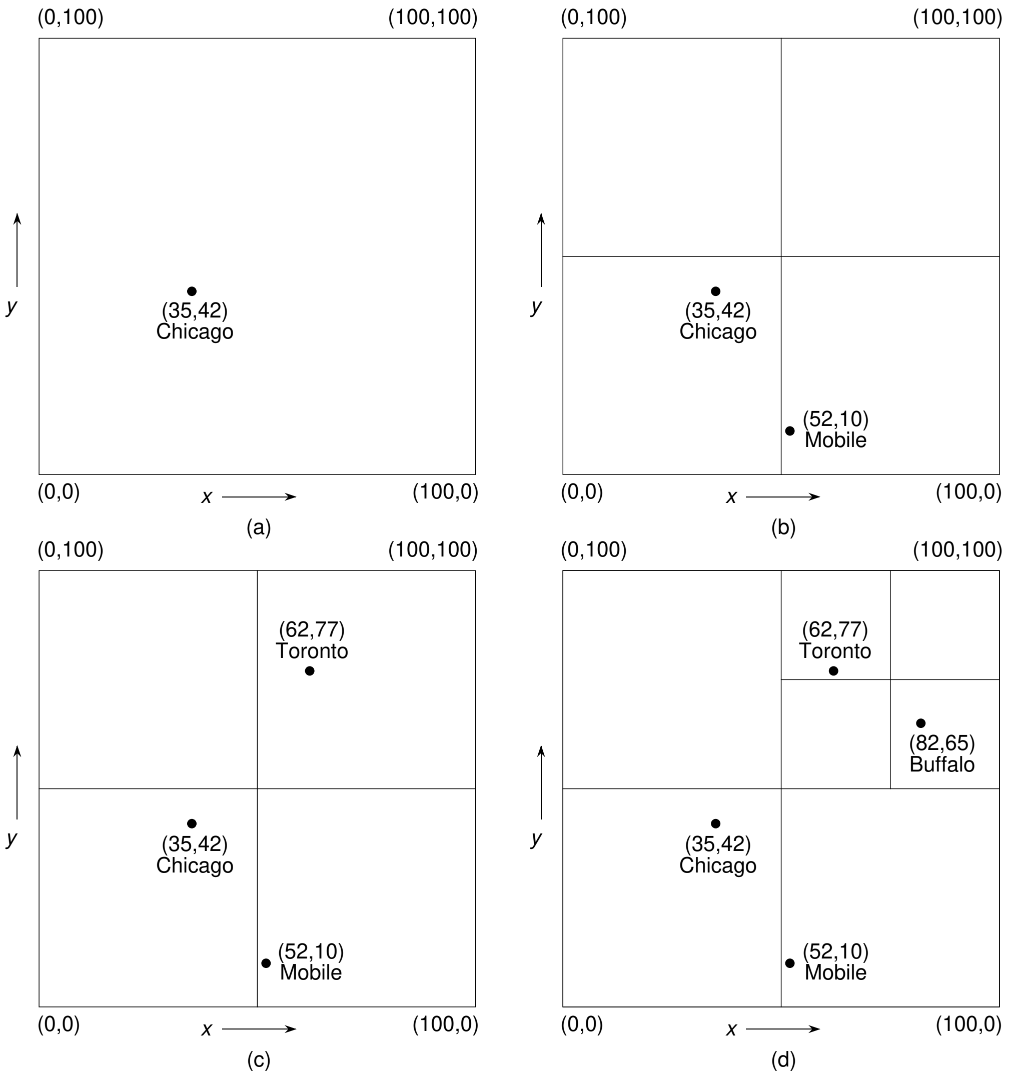
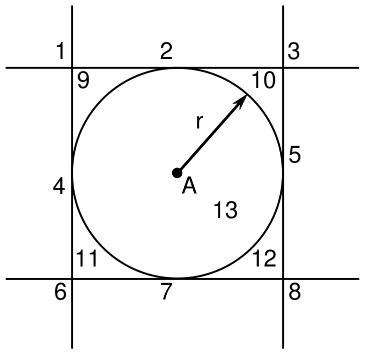


Figure 5: Sequence of partial block decompositions showing how a PR quadtree is built when adding (a) Chicago, (b) Mobile, (c) Toronto, and (d) Buffalo corresponding to Figure 1.



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

Figure 6: Relationship between a circular search space and the regions in which a root of a point quadtree may reside.

by another record with different x and y coordinate values, say B, then we must subdivide the quadrant repeatedly (termed *splitting*) until nodes A and B no longer occupy the same quadrant.

Algorithm 1: PRCOMPARE(P,X,Y)

```
if XCOORD(P) < X then
    if YCOORD(P) < Y then return SW ;
    else return NW ;
else
    if YCOORD(P) < Y then return SE ;
    else return NE ;
end
```

This may result in many subdivisions especially if the two points 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 and B is very small. As a result, we observe that every non-terminal 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 4 was built for the sequence Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, and Miami. Figures 5a, 5b, 5c, and 5d show how the tree was constructed in an incremental fashion for Chicago, Mobile, Toronto, and Buffalo.

2.2 DELETION

Deletion of nodes in PR quadtrees is considerably simpler than deletion in point quadtrees since in the PR quadtree 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 non-terminal nodes are being deleted from point quadtrees. For example, to delete Mobile from the PR quadtree in Figure 4 we simply set the SW son of its

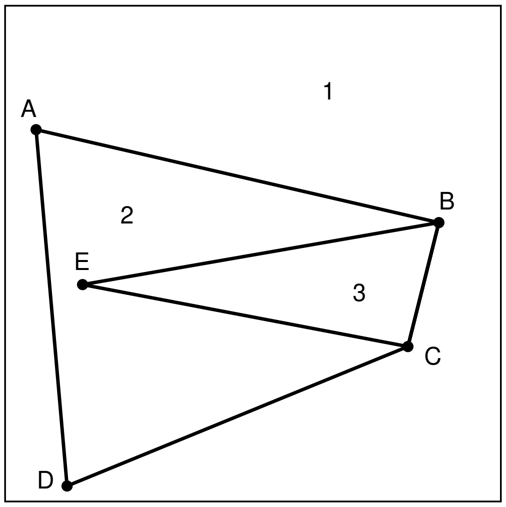


Figure 7: Sample polygonal map.

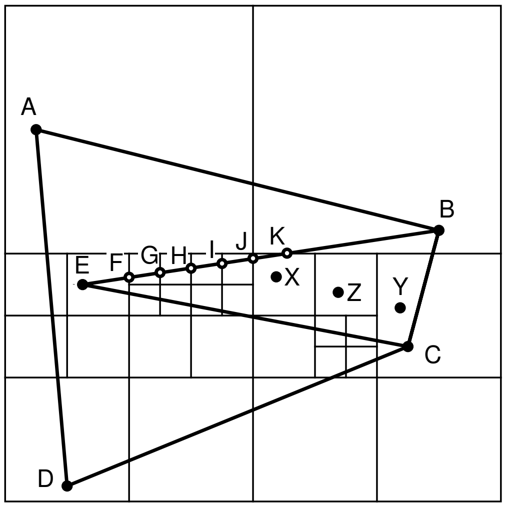


Figure 8: PM_1 quadtree corresponding to the polygonal map of Figure 7.

father node, D, to point to an empty leaf node (i.e., WHITE). Deleting Toronto is slightly more complicated. Setting the NW son of its father node, B, to to point to an empty leaf node (i.e., WHITE) is not enough since now we violate the property of the PR quadtree that each non-terminal 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 non-terminal 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 a record in a PR quadtree. Note that collapsing may take place over several levels and is only applicable when the deleted node has exactly one brother that is a terminal node and no brothers that are non-terminal 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 non-terminal nodes are returned to the free storage list. As an example, suppose we delete Mobile and Atlanta in sequence from Figure 4. 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 and non-terminal nodes D and F become superfluous and are returned to the free storage list.

2.3 SEARCH

Quadtrees are especially attractive in applications that involve search. A typical query is one that requests the determination of all nodes within a specified distance of a given data point - e.g., all cities within 50 miles of Washington, D.C. 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. As an example, we use the PR quadtree of Figure 4. Suppose that in the hypothetical data base of Figure 1 we wish to find all cities within 8 units of a data point with coordinate values (82,10). In such a case, there is no need to search the NW, NE, and SW quadrants of the root (i.e., node A with coordinate values (50,50)). Thus, we can restrict our search to the SE quadrant of the tree rooted at node A. Similarly, there is no need to search the NW and NE quadrants of the tree rooted at node D (i.e., coordinate values (75,25)).

As a further clarification of the amount of pruning of the search space that is achievable by use of quadtrees we make use of Figure 6. 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. Similarly, if R is in region 7, then the subsequent search can be restricted to the NW and NE quadrants of R.

3 PM QUADTREES

The *PM quadtree* is a term we use to collectively describe a number of related quadtree-like data structures devised by Samet and Webber [10] for representing a large collection of lines for application in computer graphics and cartography that are raster-based. The goal is to have an exact representation of the lines - not an approximation as is usually the case due to the digitization process which is inherent to a raster-based graphics system. Some common applications include the determination of the boundary of the region in which a point lies, the determination of the boundaries of all regions lying within a given distance of a point, and overlaying two maps. Samet and Webber propose a number of variants of the PM quadtree. In this assignment we focus on the PM_1 quadtree. The PM_1 quadtree is organized in a similar way to the region and PR quadtrees. A region is repeatedly subdivided into four equal-size quadrants until we obtain blocks which do not contain more than one line. In order to be able to deal with lines that intersect other lines we say that if a block contains a point, say P, then we permit it to have more than one line provided that P is an endpoint of each of the lines it contains. A block can never contain more than one endpoint. For example, Figure 8 is the block decomposition of the PM_1 quadtree corresponding to the polygonal map of Figure 7 while Figure 9 is its tree representation.

The above definition of a PM_1 quadtree can be made more rigorous by viewing the polygonal map as a straight-line planar graph consisting of vertices and edges. We use the term *q-edge* (denoting a quadtree-decomposition edge) to refer to a segment of an edge that is formed by clipping an edge of the polygonal map against the border of the region represented by a quadtree node (e.g., FG and GH in Figure 8). It should be clear that every edge (i.e., line segment) of the map is covered by a set of q-edges that only touch at their endpoints. For example, edge EB in Figure 8 consists of the q-edges EF, FG, GH, HI, IJ, JK, and KB. At this point, we can restate the definition of a PM_1 quadtree as satisfying the following conditions: .sp .np At most one vertex can lie in a region represented by a quadtree leaf. .np If a region contains a vertex, then it can contain no q-edge that does not include that vertex. .np If a region contains no vertices, then it can contain at most one q-edge. .np Each region's quadtree leaf node is maximal.

It should be clear that our definition of a PM_1 quadtree is very similar to that of a PR quadtree with the difference that we are representing edges rather than points. This has an effect on the definition of what action to take when a vertex lies on the border of a quadtree node. We could always move the vertices so that this doesn't happen, but generally this requires global knowledge about the maximum depth of the quadtree prior to its construction. Alternatively, we could also establish the convention that some sides of the region represented by a node are closed and other sides are open (as done for the PR quadtree), but this can lead to implementation difficulties when floating point numbers are involved. An additional problem arises when two pairs of colinear edges (i.e., four in total) meet at a vertex on the border of a quadtree node. The two edges that are in the open quadrant (i.e., the one not containing the vertex) may have to be decomposed to a very deep level. Therefore, the convention that is usually adopted is that all quadrants are closed. This means that a vertex that lies on the border between 2 (or 3 but never more than 4) nodes is inserted in each of the nodes on whose border it exists. Such vertices can be treated as edges of zero length. Nevertheless, in order to facilitate your task in this assignment, you will assume that the world is raster-based and thus each vertex will serve as the center of a 1×1 square (see Section 4.3).

3.1 INSERTION

Lines (or edges) are inserted into a PM_1 quadtree by searching for the position which they are to occupy. We assume that the edge does not intersect an existing edge. However, it may intersect an existing vertex. In particular, an edge is inserted into a PM_1 quadtree by traversing the tree in preorder and successively clipping it against the blocks corresponding to the nodes. Clipping is important because it enables us to avoid looking at areas where the edge cannot be inserted. If the edge can be inserted into the node (i.e., the conditions of a PM_1 quadtree node are satisfied), say P, then it is done. Otherwise, a list, say L, is formed containing the edge and any q-edges already present in the node, P is split, and the insertion process is recursively invoked to attempt to insert the elements of L in the four sons of P. We assume that the empty polygonal map is represented by a one node tree having no edges.

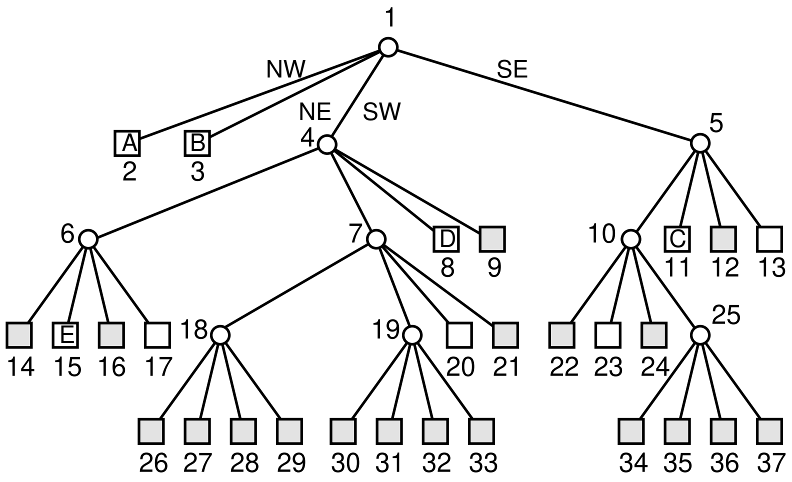


Figure 9: Tree representation of the PM_1 quadtree corresponding to the polygonal map of Figure 7. All leaf nodes through which a q-edge passes (with the exception of those that contain a vertex) are shown shaded.

3.2 DELETION

Deletion of an edge from a PM_1 quadtree is analogous to the process used for PR quadtrees. The control structure is identical to that used in the insertion of an edge. Again, the tree is traversed in preorder and the edge is successively clipped against the blocks corresponding to the nodes. Once a leaf node is encountered in which the edge participates, the edge is removed from the list of edges associated with the node. The difference from deletion in a PR quadtree is that a check for collapsing is made at each level of the PM_1 quadtree where processing one of the sons of a non-terminal node has resulted in the deletion of a q-edge. In particular, collapsing can occur if at least one of the sons of a non-terminal node is a terminal node. There are two situations where collapsing must occur. We shall use as an example the deletion of edge BE from Figure 7 to illustrate them. In the discussion we refer to nodes in Figures 8 and 9.

The first situation arises when the deletion of an edge has resulted in four brother terminal nodes having zero or one edge pass through them. In our example, removal of q-edges FG and GH from nodes 26 and 27 leaves non-terminal node 18 with only one edge (i.e., CE) passing through it. This means that collapsing can be applied after processing the remaining sons of non-terminal node 18. Similarly, removal of q-edges HI and IJ from nodes 30 and 31 leaves node 19 with only one edge (i.e., CE) passing through it. Once again collapsing is applied after processing the remaining sons of non-terminal node 19. In fact, further collapsing is possible at this point since non-terminal node 7 now has only one edge (i.e., CE) passing through it.

The second situation is somewhat tricky. It arises when deletion causes all of the remaining edges in the descendants of the non-terminal node, say N, to have a common vertex which lies in one of the sons of N (not a descendant of N who is not a son!). In this case collapsing can occur thereby making N a terminal node. In our example, removal of q-edge JK from node 22 leaves non-terminal node 10 with only two edges (i.e., CE and CD) passing through node 25 and having C as a common vertex. Moreover, removal of q-edge JK also leaves non-terminal node 5 with the three edges BC, CD, and CE passing through it and, again, all having C as a common vertex. Thus collapsing is applied to the sons of nodes 25, 10, and 5 in succession. It is very important to note that collapsing can only occur after processing the sons of non-terminal node 5. The reason it cannot be performed sooner (e.g., in part, after processing the sons of non-terminal node 23 or 10) is that the common vertex (i.e., C) is not in the node whose sons are being collapsed.

3.3 SEARCH

The most common search query is one that seeks to determine the boundary of the polygon in which a given point lies. A variation of this query is to locate all the boundaries of all of the polygons within a specified distance of a given data point - e.g., all polygons within 20 units of a point at location (X,Y). Range queries can also be performed. However, they are more usefully cast in terms of finding the boundaries of all of the polygons of a polygonal map that lie within a given area. The test area can be an arbitrary polygon and usually we wish to know the polygons that are enclosed as well as partially overlapped by the test area. This is a subproblem of the general map overlay problem. Note that we have not said anything about identifying the polygons in the search queries aside from their boundaries. The maintenance of polygon labels in a dynamically changing environment (i.e., when edges are inserted and deleted) is a non-trivial problem. This issue is addressed further in Section 5. An example JAVA applet for the PM_1 quadtree data structure can be found on the home page of the class.

4 ASSIGNMENT

This assignment is divided into four parts. You should program in the C language. The first part is concerned with data structure selection. The second part requires the construction of a command decoder. The third and fourth parts require that you implement a given set of operations.

The first two parts are to be turned in first two weeks (one week for each part). They are worth 25 points in total. There will be NO late submissions accepted for these two parts of the assignment. The data structure specification is worth 10 points and the command decoder is worth 15 points. While doing parts one and two you are also to start thinking and coding the program necessary to implement the operations. This should be done in such a way that the data structure is a BLACK BOX. Thus you need to specify your primitives in such a way that they are independent of the data structure finally chosen. You are strongly advised to begin implementing some of the operations. For example, you should implement an output routine so that you can see whether your program is working properly.

For the third and fourth parts of the assignment, you are to write a C program to implement the data structure and the specified operations. Together they are worth 175 points. Part three consists of operations (1)-(7) given below. They are worth 95 points. Part four consists of operations (8)-(11) given below. They are worth 80 points. Operations (12) and (13) are for extra credit and are to be turned in with part four. They are worth 30 points.

In order to facilitate grading and your task, you are to use the data structure implementation that will be given to you in class on the first meeting date after you turn in the first two parts of the assignment. For any operation that is not implemented, say OP, your command decoder must output a message of the form `THE COMMAND IS NOT IMPLEMENTED` in one line.

In order to facilitate your program as well as lend some realism to your task you are to implement the PM_1 quadtree in a raster-based graphics environment. This means that you are dealing with a world of pixels. For the sake of simplicity, assume that the world is a $2^7 \times 2^7$ (i.e.,128×128) array of pixels. The pixel at the lower left corner has coordinate values (0,0) and the pixel at the upper right corner has coordinate values (127,127). Each pixel serves as the center of a square of size 1×1. This is the smallest unit into which our quadtrees will decompose the world. At times, the situation may arise that insertion of a line may force the decomposition to be deeper than 7 levels by virtue of condition (3) of the PM_1 uadtree definition. In this case you should stop subdivision but still insert the line into the node. Note that a line will be specified in integers (i.e., the coordinate values of its endpoints). In addition, we restrict our lines to have a minimum horizontal or vertical displacement of one pixel widths. Thus isolated vertices (i.e., pixels) are not permitted. All lines must have their endpoints in our world of pixels.

One class meeting date before the due date of each part of the project you will be informed of the availability of and name of the test data file which you are to use in exercising your program for grading purposes. You should also prepare your own test data. A sample file for this purpose will also be provided.

4.1 DATA STRUCTURE SELECTION

You are to select a data structure to implement the PM_1 quadtree. Turn in a definition in the form of a set of C structures. In doing this part of the assignment you should bear in mind the type of data that is being represented and the type of operations that will be performed on it. In order to ease your task, remember that the primitive entity is the line. We specify a line by giving the x and y coordinate values of its endpoints. The rest of your task is to build on this entity adding any other information that is necessary. The nature of the operations is described in Sections 4.3-4.5.

From the description of the operations you will see that a name is associated with each line. At times, the operations are specified in terms of these names. Thus you will also need a mechanism to efficiently keep track of these names. It should be integrated with the rest of your data structures.

4.2 COMMAND DECODER

You are to turn in a working command decoder written in C for all the commands (including the optional ones) given in Sections 4.3-4.5. You are not expected to do error recovery and can assume that the commands are syntactically correct. All commands will fit on one line. Lengths of names are restricted to 6 characters or less and can be any combination of letters or digits (e.g., A, 1, 2A, B33, etc.). However, for your own safety you may wish to incorporate some primitive error handling. Test data for this part of the assignment will be found in a file specified by the Teaching Assistant.

4.3 PART THREE: BASIC OPERATIONS

(1) [20 points] Initialize the quadtree. The command `INIT_QUADTREE(WIDTH)` is always the first command in the input stream. `WIDTH` determines the length of each side of the square are covered by the quadtree. Each side has the length 2^{WIDTH}. It also has the effect of starting with a fresh data set. This also enables you to start working with a new polygonal map. The initialization can also be invoked by `BUILD_QUADTREE(WIDTH)`, which is followed by a tree representation. The format of the representation is introduced in the course slides about alternative quadtree representations (ar6) as well as an appendix on the course web page. Besides the initialization, you should also be able to output the tree representation of the current quadtree, which is invoked by `ARCHIVE_QUADTREE()`. It also gives you a quadtree that is correct in case you are not able to build the quadtree correctly.

(2) [10 points] Generate a display of a $2^{\mathrm{WIDTH}} \times 2^{\mathrm{WIDTH}}$ square from the PM_1 quadtree. It is invoked by the command `DISPLAY()`. To draw the PM_1 quadtree, you are to use the drawing routines provided. An appendix to the project description covers their use, and the utilities *showquad* and *printquad*, that can be used to render the output of your programs on a screen or a printer. A dashed (broken) line should be used to draw quadrant lines, but the lines should be solid (i.e., not dashed). Line names should be output somewhere near the line.

(3) [10 points] List all the lines in the data base in alphanumerical order. This means that letters come before digits in the collating sequence. Similarly, shorter identifiers precede longer ones. For example, a sorted list is A, AB, A3D, 3DA, 5. It is invoked by the command `LIST_LINES()` and yields for each line its name, and the x and y coordinate values of its endpoints. This will be used to interpret the display since sometimes it will not be possible to distinguish the boundaries of the lines from the display. You should list all of the lines in the database whether or not they have been deleted. The output should consist of several lines and each line only contains one name.

(4) [10 points] Create a line by specifying its two endpoints and assign it a name for subsequent use. It is invoked by `CREATE_LINE(N,AX,AY,BX,BY)` where N is the name to be associated with the line, `AX` and `AY` are the x and y coordinate values, respectively, of one endpoint while `BX` and `BY` are the x and y coordinate values, respectively, of its other endpoint. Output an appropriate message indicating that the line has been created as well as its name and endpoints: `LINE N IS CREATED`. In the following, you may need to output messages for many operations. Please note that each message should occupy one line and do not output unnecessary symbols (including new lines and spaces).

(5) [10 points] Determine whether a query line overlaps or intersects any of the existing lines. By overlap we mean that the line is a subline of an existing line. For example, a line from (3,5) to (8,5) is a subline of the line from (1,5) to (10,5). This operation is a prerequisite to the successful insertion of a line in the PM_1 quadtree. It is invoked by the command `LINE_SEARCH(N)` where `N` is the name of a line. If the line does not overlap or intersect an existing line, then `LINE_SEARCH` returns a value of false and outputs a message `N DOES NOT INTERSECT ANY EXISTING LINE`. Otherwise, it returns the value true and uses the number of all overlapping or intersecting lines to output the following message: `N INTERSECTS X LINE(S)`. Note that it is allowed in the PM_1 quadtree that an endpoint of the query line touches the endpoint of an existing line. As a result, if the line only touches endpoints of the other lines in quadtree, you should return false and output `N DOES NOT INTERSECT ANY EXISTING LINE`. You are only to check against the lines that are in the

quadtree of existing lines and not the lines that existed at some time in the past and have been deleted by the time this command is executed.

(6) [15 points] Insert a line in the PM_1 quadtree. If the line overlaps or intersects an existing line, then do not make the insertion and report this fact by outputing the number of the overlapping or intersecting lines: `N INTERSECTS X LINE(S)`. Otherwise, return the name of the line that is being inserted as well as output a message indicating that this has been done: `N IS INSERTED`. It is invoked by the command `INSERT(N)` where `N` is the name of a line. It should be clear that the PM_1 quadtree is built by a sequence of `INSERT` operations. You may assume that the line has been created before insertion.

(7) [20 points] Delete a line or a set of lines from the PM_1 quadtree. This operation has two variants, `DELETE_LINE` and `DELETE_POINT`. The command `DELETE_LINE(N)` deletes the line named `N`. It returns `N` if it was successful in deleting the specified line and outputs a message: `N IS DELETED`. Otherwise, it outputs the message `N DOES NOT EXIST`. On the other hand, the command `DELETE_POINT(PX,PY)` deletes all the lines which have an endpoint whose x and y coordinate values are given by `PX` and `PY`, respectively. `DELETE_POINT` returns as its value the names of the lines that have been deleted and prints an appropriate message indicating the number of edges deleted: `X LINE(S) DELETED`. If the point is not on any line, then an appropriate message indicating this is output: `LINES DO NOT EXIST`.

4.4 PART FOUR: ADVANCED OPERATIONS

(8) [10 points] Find the nearest line from a given point. To locate a nearest neighbor we use the command `NEIGHBOR(PX,PY)` where `PX` and `PY` are the x and y coordinate values, respectively, of the point whose neighboring line is sought. The command returns as its value the name of the neighboring line if one exists and outputs it: `THE NEAREST NEIGHBOR IS N`; otherwise (i.e., the tree is empty) an appropriate message is output: `THE LINE DOES NOT EXIST`. Note that if there are multiple nearest lines, you should output the one with the smallest name in alphanumerical order.

(9) [15 points] Find the k-th nearest line from a given point. To locate the k-th nearest neighbor we use the command `KTH_NEIGHBOR(PX,PY)` where `PX` and `PY` are the x and y coordinate values, respectively, of the point whose neighboring line is sought. The command returns as its value the name of the k-th neighboring line if one exists and outputs it: `THE KTH NEAREST NEIGHBOR IS N`; otherwise (i.e., the tree is empty) an appropriate message is output: `THE LINE DOES NOT EXIST`. Note that if there are multiple k-th nearest lines, you should sort them according to the names in alphanumerical order.

(10) [30 points] Find all lines in a rectangular window. It is invoked by the command `WINDOW(AX,AY,BX,BY)` where `AX` and `AY` are the x and y coordinate values, respectively, of the left and lower corner of the window and `BX` and `BY` are the x and y coordinate values, respectively, of the right and upper corner. You may assume `AX<BX` and `AY<BY`. Your output is the number of the lines completely inside the window: `X LINE(S) IN THE WINDOW`. This is similar to a clipping operation. If the window query is invoked by `WINDOW_DISPLAY(AX,AY,BX,BY)`, you should also output a display of the PM_1 quadtree that only shows segments of the lines that are in the window. Draw the boundary of the window using the percent symbol. Do not show quadrant lines within the window. The correct display is worth 10 points.

(11) [25 points] Find the polygon that encloses a given point. It is invoked by `FIND_POLYGON(PX,PY)` where `PX` and `PY` are the x and y coordinate values, respectively, of the point whose surrounding polygon is sought. To perform this operation you must first find the nearest line and then do a walk of edges so that the point is always to the same side. `FIND_POLYGON` returns a list of the vertices of the polygons if such a polygon exists. In this case, you should output the area of the polygon: `POLYGON FOUND: S/2 PIXELS`, where `S` is two times the area of the polygon, which is an integer. For example, if the result polygon is the rectangle whose two corners are (0,0) and (10,10) you should output `POLYGON FOUND: 200/2 PIXELS`. Otherwise, an appropriate error message is output indicating the problem: `NO ENCLOSING POLYGON`. Some examples of the case where a polygon fails to exist include touching the border of the space in which the PM_1 quadtree is embedded. Another problematic example arises when there are several polygons in the world which are not connected (i.e., they do not have any edge in common) and the point is in the area that is not in any of the polygons. How do you detect such a situation? Holes may also pose a problem (e.g., Figure 10). In addition, you must be able to handle the situation that a vertex has only one line emanating from it in which case you may stop the search and output an appropriate message, or ignore the line (or lines) and continue the walk. Your grade for the task will depend, in part, on the efficiency and completeness of your solution.

4.5 OPTIONAL OPERATIONS

(12) [20 points] Generalize the `WINDOW` command to work for a window in the form of an arbitrary polygon. For the simplicity of the implementation, the lines do not need to be completely inside the window. Instead, you should count all lines intersecting the window (including those that touch the boundary and are completely inside). It is invoked by the command `POLYGONAL_WINDOW(PXS,PYS)`, followed by an arbitrary number of commands `VERTEX_WINDOW(PX,PY)`, and terminated by the command `END_WINDOW()`. `PXS` and `PYS` are the x and y coordinate values, respectively, of the first vertex in the window polygon. The `VERTEX_WINDOW` command is used to specify the remaining vertices in the polygonal window in a clockwise order with `PX` and `PY` serving as their x and y coordinate values, respectively. `END_WINDOW` indicates the end of the specification process of the window and that the last vertex is to be connected to the first vertex. Your output is the number of the lines: `X LINE(S) IN THE POLYGONAL WINDOW`. If the operation is invoked by `POLYGONAL_WINDOW_DISPLAY(PXS,PYS)` (the remaining commands indicating the vertices of the window are the same), you should also output a display of the PM_1 quadtree that only shows segments of the lines that intersects the window. Draw the boundary of the window using the percent symbol. Do not show display quadrant lines within the window. The correct display is worth 10 points.

(13) [10 points] Perform connected component labeling on the PM_1 quadtree. This means that all WHITE (i.e., empty) quadtree blocks within the same polygon are assigned the same label. This is accomplished by the command `LABEL()`. The result of the operation is a display of the PM_1 quadtree where all blocks in each polygon are shown with the same label. If a block is of size 1×1 or $q \times 2$, then do not display its boundaries (i.e., omit the quadrant lines) and just output the label. Blocks that fall in more than one polygon are displayed without a label. Use one character labels formed from letters, digits, and other symbols as necessary.

5 DISCUSSION

Absent from the set of operations that you have been requested to implement (with the exception of operation 11) is any notion of maintaining region (or polygon) identification with the q-edges. This is not an oversight for the question of how to efficiently maintain the labels of the q-edges as lines are inserted and deleted is a non-trivial matter. These actions cause a split of a region into two parts or the merger of two regions. Updating the region information is complex. In the rest of this section we detail an approach described by Samet and Webber [10] to deal with this problem. It is obviously more complex than you wish to implement. However, it does provide a good case study in the judicious selection and use of data structures to overcome algorithmic complexity issues. Of course, the ensuing discussion is predicated on the premise that the PM_1 quadtree is a reasonable approach to storing collections of line segments which is arguable - an issue that is best left to another forum.

For example, Figure 10 contains a pentagonal region ABCDE with many holes. All the q-edges forming the ABCDE border and those forming the outer border of each of the holes must be labeled with the name of the pentagonal region. Now consider what happens when edge BE is inserted. The outer border of the pentagonal region has been split in two and the new edge has been inserted forming two new regions - a triangular region and a quadrilateral region. Also the holes of the original pentagonal region must have their labels changed to indicate whether they are now in the triangular region or in the quadrilateral region. If the region labels were stored directly on each q-edge, then the updating of region labels would require that every q-edge be visited. If no additional data structure were used to organize the labels, then visiting each q-edge would be analogous to doing a connected component analysis of the portion of the map bounded by the pentagonal region. This analysis can be greatly simplified if all the q-edges in a particular region are kept in a linked list, but it will still involve an amount of work proportional to the number of q-edges in the outer border of the pentagonal region and in the outer border of all the holes inside the pentagonal region. In the following, we propose to organize the q-edges within a region in a more efficient manner than a linked list. Note that the above comments about the insertion of the edge BE apply equally to the problem of deleting the edge BE.

The q-edges that border a region can be partitioned into a collection of disconnected chains of q-edges. Elements of this collection correspond to walks along the outer border of the region, to walks along the outer borders of any holes in the region, and miscellaneous q-edges that have the same region on both their left

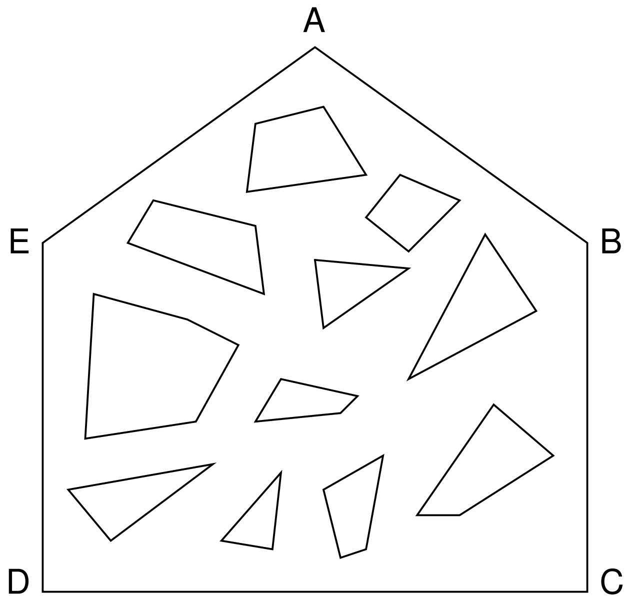


Figure 10: Example polygonal map with many holes.

and right sides (and hence are not properly borders of any region). The q-edges of each chain are grouped into a 2-3 tree [1] (termed a *chain 2-3 tree*) and we only store the label information (i.e., an identifier for the chain) in the root of the tree. In fact, each q-edge is linked to the leaves of at most two of these 2-3 trees (one link for the region on each side of the q-edge) and the region information is found by moving up the tree using father links. Moreover, the chains of each region are also grouped into a 2-3 tree (termed a *region 2-3 tree*) and again we only store the label information in the root of the tree (this time the label information is the name of the region). The 2-3 tree is chosen because of its algorithmic simplicity (it is a degenerate case of the much studied B-tree [2]). In particular, the 2-3 tree has the property that insertion, deletion, and splitting and concatenation of two 2-3 trees can be performed in $O(\log_2 n)$ time for trees of size n. In addition, two 2-3 trees of size t_1 and t_2 respectively can be merged into a new 2-3 tree in $O(t_1 + t_2)$ time. However, other balanced tree techniques could have also been used.

In order to be able to make use of the 2-3 trees we must be able to define an ordering for q-edges and chains. The q-edges in a given chain can be ordered according to their appearance during a walk along the chain starting at the leftmost of the uppermost vertices incident to the chain. In case of a tie (e.g., a closed chain in which case there are two q-edges meeting this criterion), we choose the q-edge whose other endpoint is uppermost leftmost. We also have to order the collection of chains corresponding to each region which we do by using their “extreme” q-edge. By extreme we mean that for each chain we choose the q-edge having the uppermost leftmost endpoint. Again, in case of a tie (e.g., extreme q-edges that share the same endpoint), we apply the same rule to the second endpoint. This extreme q-edge will provide a unique label for the entire region (with respect to a particular map).

If deletion or insertion of a q-edge does not change the number of regions in the map, then the operation can be performed in $O(\log_2 q)$ time where q is the number of q-edges (i.e., leaves) in the largest chain 2-3 tree associated with a given region. If deletion of a q-edge merges two regions, then the region 2-3 trees that order the collection of chains for each region will need to be merged. If the number of leaves (i.e., chains) in the region 2-3 trees representing these two regions is t_1 and t_2, respectively, then the cost of this operation will be $O(t_1 + t_2 + \log_2 q)$ time. This is accomplished by a procedure that merges the two collections of disconnected chains in $O(t_1 + t_2)$ time and creates the border of the new region from the borders of the two merged regions in $O(\log_2 q)$ time. Note that for cartographic data the individual chains are typically long (i.e., q can become large), but the number of chains in each region's collection is usually small for a given region (i.e., $t_1 + t_2$ is usually small). Indeed, if there are no chains that have the same region on both sides, then the average value of t_1 (and also t_2) could not be larger than 1 (corresponding to the region's outer border) plus the average number of holes. But since each hole is itself a region, the average number of holes cannot exceed 1. Therefore the average value of t_1 (and of t_2) cannot exceed 2.

Similarly, when q-edge insertion causes splitting of a region with a region 2-3 tree of t chains, the cost will be $O(t + \log_2 q)$ time. This is accomplished by a procedure that separates the collection of disconnected chains of q-edges that bounded the old region (i.e., its region 2-3 tree) into a collection for each of the new regions (i.e., new region 2-3 trees) in $O(t)$ time and splits the outer border of the old region into the two outer borders of the new regions in $O(\log_2 q)$ time. Note that the new region might be created by the inserted q-edge completing the border of a hole instead of connecting the outer border in 2 places. The cost of determining the region to which a q-edge belongs is $O(\log_2 q)$ time.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1974.
2. D. Comer, The Ubiquitous B-tree, *ACM Computing Surveys 11*, 2(June 1979), 121-137.
3. R. A. Finkel and J. L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1(1974), 1-9.
4. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley, Reading, MA, 1990.
5. G. M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.
6. A. Klinger, Patterns and Search Statistics, in *Optimizing Methods in Statistics*, J. S. Rustagi, Ed., Academic Press, New York, 1971, 303-337.
7. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, Second Edition, McGraw Hill-Hill, New York, 1979.
8. H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Morgan-Kaufmann, San Francisco, 2006.
9. H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS* Addison-Wesley, Reading, MA, 1990.
10. H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222 (also *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127-132; and University of Maryland Computer Science TR-1372).
11. J. L. Warnock, A hidden surface algorithm for computer generated half tone pictures, Computer Science Department TR 4-15, University of Utah, Salt Lake City, June 1969.

SAMPLE INPUT

```
INIT_QUADTREE(4)
CREATE_LINE(AB,8,1,10,2)
CREATE_LINE(BC,10,2,5,5)
CREATE_LINE(CA,5,5,8,1)
CREATE_LINE(CD,5,5,9,7)
CREATE_LINE(DE,9,7,12,10)
CREATE_LINE(EC,12,10,5,5)
CREATE_LINE(BD,10,2,9,7)
CREATE_LINE(BF,10,2,11,5)
CREATE_LINE(FD,11,5,9,7)
CREATE_LINE(GE,12,6,12,10)
CREATE_LINE(DG,9,7,12,6)
CREATE_LINE(GF,12,6,11,5)
CREATE_LINE(BG,10,2,12,6)
CREATE_LINE(HE,13,9,12,10)
CREATE_LINE(GH,12,6,13,9)
CREATE_LINE(1,13,5,15,13)
LINE_SEARCH(1)
INSERT(GE)
INSERT(BG)
INSERT(HE)
DELETE(BG)
INSERT(GF)
DELETE(HE)
DELETE(GF)
CREATE_LINE(2,1,4,10,7)
LINE_SEARCH(2)
CREATE_LINE(3,7,2,15,6)
LINE_SEARCH(3)
INSERT(HE)
CREATE_LINE(4,11,5,5,14)
LINE_SEARCH(4)
DELETE(GE)
INSERT(CA)
CREATE_LINE(5,14,14,6,5)
LINE_SEARCH(5)
DELETE_POINT(9,7)
CREATE_LINE(6,4,5,5,0)
LINE_SEARCH(6)
DELETE_POINT(12,6)
CREATE_LINE(7,9,10,5,0)
LINE_SEARCH(7)
INSERT(DE)
INSERT(GF)
CREATE_LINE(8,12,15,6,2)
LINE_SEARCH(8)
INSERT(GE)
DELETE(GF)
DELETE(DE)
DELETE_POINT(8,1)
CREATE_LINE(9,6,7,2,14)
LINE_SEARCH(9)
```

```
INSERT(BG)
CREATE_LINE(10,8,8,13,1)
LINE_SEARCH(10)
CREATE_LINE(11,15,11,10,7)
LINE_SEARCH(11)
INSERT(DE)
INSERT(AB)
CREATE_LINE(12,11,4,13,3)
LINE_SEARCH(12)
DELETE(GE)
```

SAMPLE OUTPUT

```
LINE AB IS CREATED
LINE BC IS CREATED
LINE CA IS CREATED
LINE CD IS CREATED
LINE DE IS CREATED
LINE EC IS CREATED
LINE BD IS CREATED
LINE BF IS CREATED
LINE FD IS CREATED
LINE GE IS CREATED
LINE DG IS CREATED
LINE GF IS CREATED
LINE BG IS CREATED
LINE HE IS CREATED
LINE GH IS CREATED
1 IS CREATED
1 DOES NOT INTERSECT ANY EXISTING LINE
GE IS INSERTED
BG IS INSERTED
HE IS INSERTED
BG IS DELETED
GF IS INSERTED
HE IS DELETED
GF IS DELETED
2 IS CREATED
2 DOES NOT INTERSECT ANY EXISTING LINE
3 IS CREATED
3 DOES NOT INTERSECT ANY EXISTING LINE
HE IS INSERTED
4 IS CREATED
4 DOES NOT INTERSECT ANY EXISTING LINE
GE IS DELETED
CA IS INSERTED
5 IS CREATED
5 DOES NOT INTERSECT ANY EXISTING LINE
0 LINE(S) DELETED
6 IS CREATED
6 DOES NOT INTERSECT ANY EXISTING LINE
0 LINE(S) DELETED
7 IS CREATED
```

```
7 INTERSECTS 1 LINE(S)
DE IS INSERTED
GF IS INSERTED
8 IS CREATED
8 INTERSECTS 1 LINE(S)
GE IS INSERTED
GF IS DELETED
DE IS DELETED
1 LINE(S) DELETED
9 IS CREATED
9 DOES NOT INTERSECT ANY EXISTING LINE
BG IS INSERTED
10 IS CREATED
10 INTERSECTS 1 LINE(S)
11 IS CREATED
11 INTERSECTS 2 LINE(S)
DE IS INSERTED
AB IS INSERTED
12 IS CREATED
12 INTERSECTS 1 LINE(S)
GE IS DELETED
```