Fall 2019

CMSC 420
Hanan Samet

Programming Assignment 1: A Data Structure For VLSI Applications[1]

Abstract

In this assignment you are required to implement an information management system for handling data similar to that used in VLSI (very large scale integration) applications. In such an environment the primary entities are small rectangles and the problem in which we are interested is how to manage a large collection of them. In the following we trace the development of a variant of the quadtree data structure that has been found to be useful for such a problem. Your task is to implement a Rectangular Quadtree in such a way that a number of operations can be efficiently handled. An example JAVA applet for the data structure can be found on the home page of the class.

This assignment is divided into four parts. C or C++ are the permitted programming languages. JAVA is not permitted. Also, you are not allowed to make use of any built in data structures from any library such as, but not limited to, STL in C++. For the first two parts, you must read the attached description of the problem and data structure. A detailed explanation of the assignment including the specification of the operations which you are to implement is found at the end of the description. After you have done this, you are to turn in a proposed implementation of the data structure using C++ classes or C structs. This is due at the beginning of the class meeting one week after this assignment has been distributed to you. No late solutions will be accepted for this part.

One week later you must turn in a C or C++ program for the command decoder (i.e., scanner for the commands corresponding to the operations which are to be performed on the data structure). For the third part, you are to write a C or C++ program to implement the data structure and operations (1)-(9). For the fourth part, you are to implement operations (10)-(14). Operations (15)-(17) are optional and you will get extra credit if you turn them in with part four.

[1]Copyright ©2019 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 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 [1] 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 1.14[2] where a point quadtree of 8 nodes 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 (i.e., in Figure 1.14, Chicago at location (35,40) subdivides the two dimensional space into four rectangular regions). Requiring the regions to be of equal size leads to the region quadtree of Klinger [4,5,6]. 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. For a history and review of the quadtree representation, see pp. 423-426 in [5].

As an example of the region quadtree, consider the region shown in Figure 1.28a which is represented by a $2^3 \times 2^3$ binary array in Figure 1.28b. Observe that 1's correspond to picture elements (termed pixels) which are in the region and 0's 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 1's or 0's (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 1's or entirely of 0's. For example, the resulting blocks for the region of Figure 1.28b are shown in Figure 1.28c. 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 1.28c is shown in Figure 1.28d.

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 contains at most one data point. For example, Figure 1.31 is the PR quadtree corresponding to the data of Figure 1.1. 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.

Quadtrees are especially attractive in applications that involve search. A typical query is one that re-

[2] All numbered figures of the form X.YY and page numbers refer to [5] while all numbered figures of the form ZZ are found in this document.

quests 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 1.31. Suppose that 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 1.27. 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. For more details on PR quadtrees, see pp. 42-47 in [5].

3 Rectangle Quadtrees

The *Rectangle quadtree* is a term we use to describe a quadtree-like data structure for representing a large collection of non-overlapping rectangles for application in computer graphics, VLSI, and cartography that are raster-based. The goal is to have an exact representation of the rectangles. The Rectangle 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 rectangle. For example, Figure 2 is the block decomposition of the Rectangle quadtree corresponding to the collection of rectangles of Figure 1 while Figure 3 is its tree representation.

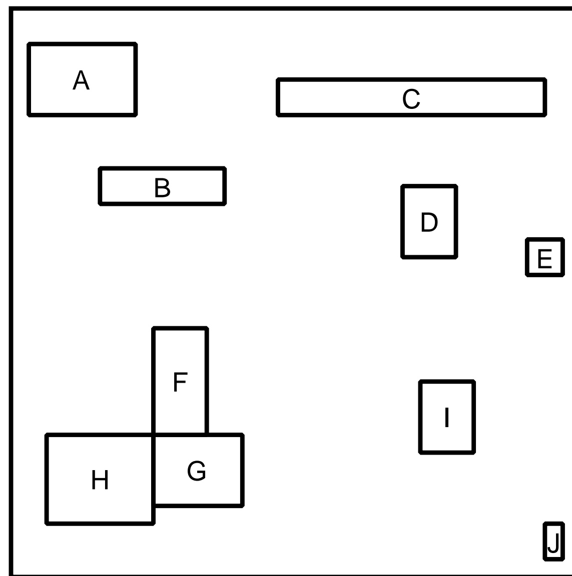

Figure 1: Sample collection of rectangles.

The term *r-piece* is used to refer to a segment of a rectangle that is formed by clipping a piece of a rectangle against the border of the region represented by a quadtree node. It should be clear that every rectangle in the collection is covered by a set of r-pieces that are connected. Note that our definition of

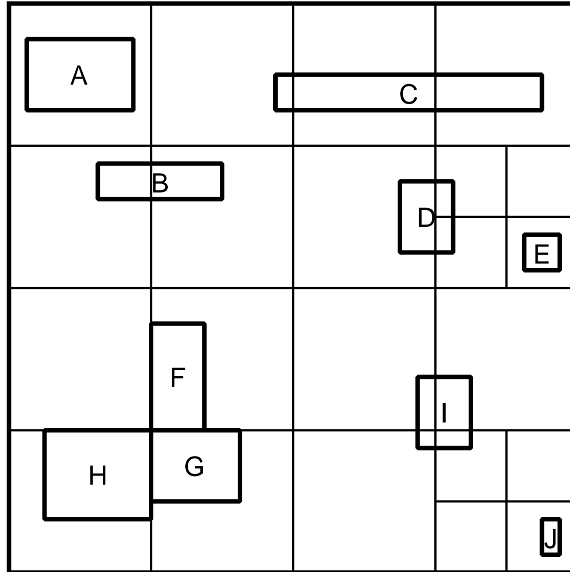

Figure 2: Blocks corresponding to the quadtree decomposition of the Rectangle quadtree for the collection of rectangles in Figure 1.

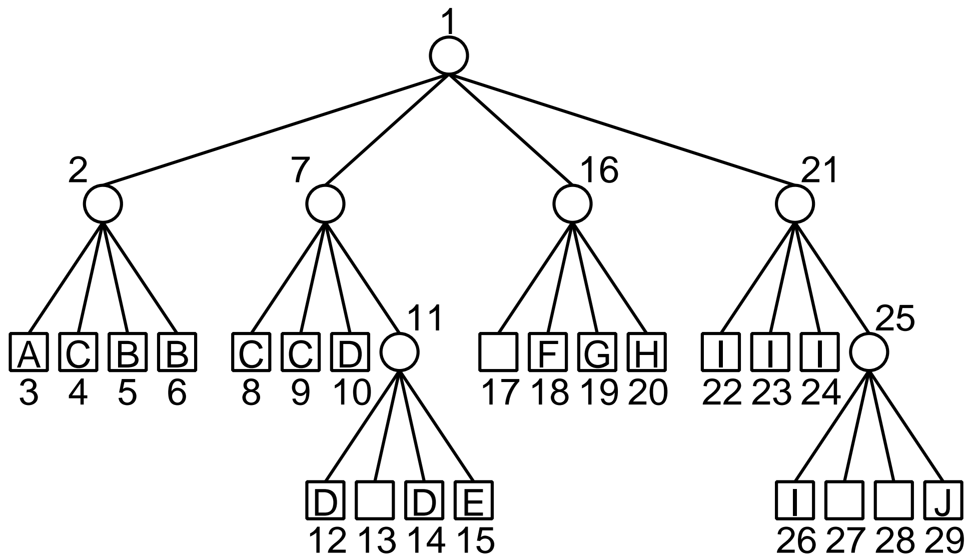

Figure 3: Tree representation corresponding to the quadtree decomposition of the Rectangle quadtree for the collection of rectangles in Figure 1.

a Rectangle quadtree is very similar to that of a PR quadtree with the difference that we are representing rectangles rather than points. This has an effect on the definition of what action to take when the sides of a rectangle are coincident with the border of a quadtree node. We make use of the convention that the left and bottom sides of the region represented by a node are closed and the right and top sides are open (as done for the PR quadtree).

3.1 Insertion

Rectangles are inserted into a Rectangle quadtree by searching for the position which they are to occupy. We assume that the rectangle does not intersect (i.e., overlap) an existing rectangle. In particular, a rectangle is inserted into a Rectangle 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 rectangle cannot be inserted. If the rectangle can be inserted into the node (i.e., the node is empty), say *P*, then we are done. Otherwise, a list, say *L*, is formed containing the rectangle and any r-pieces 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*. For example, Figure 4a–e shows how the Rectangle quadtree for the collection of rectangles in Figure 1 is constructed in incremental fashion for rectangles A, B, C, D, and E. We assume that the empty collection is represented by a one node tree having no rectangles.

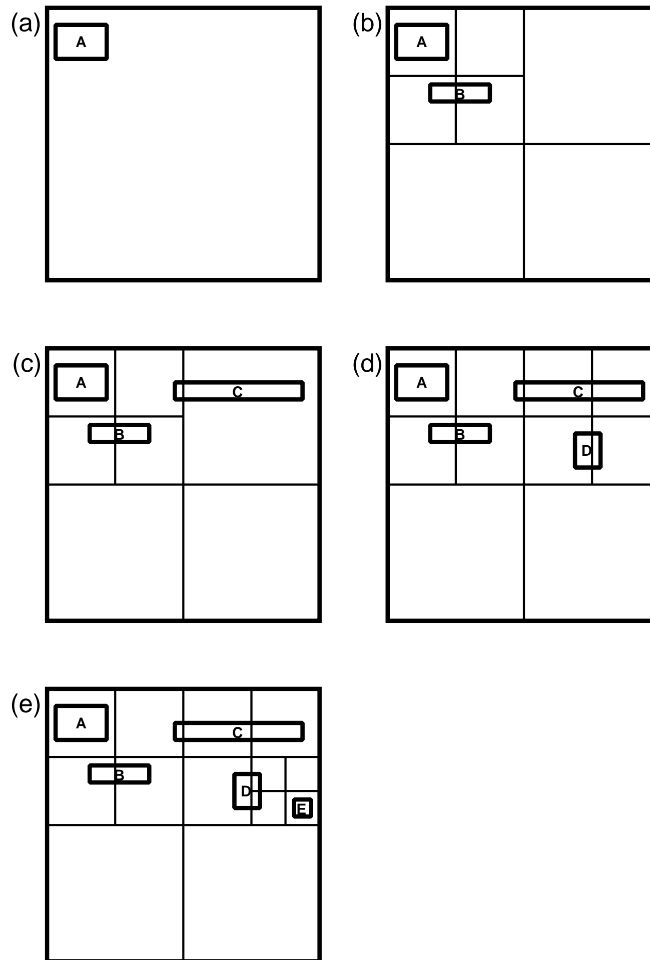

Figure 4: Sequence of partial block decompositions showing how a Rectangle quadtree is built when adding (a) A, (b) B, (c) C, (d) D, and (e) E corresponding to the collection of rectangles in Figure 1.

3.2 Deletion

Deletion of a rectangle, say *R*, from a Rectangle quadtree is analogous to the process used for PR quadtrees. The control structure is identical to that used in the insertion of a rectangle. Again, the tree is traversed in preorder and the rectangle is successively clipped against the blocks corresponding to the nodes. Once a leaf node is encountered in which rectangle *R* participates, say *P*, the rectangle is removed from *P*. Once the remaining brothers of *P* have been checked for the presence of r-pieces of *R*, we determine if nodes can be merged (termed *collapsing*; for more details, see pp. 43-44 and the solutions to the associated exercises in [5]). Collapsing takes place if the brothers of *P* were either empty or contained the r-pieces of the same rectangle. The difference from deletion in a PR quadtree is that in a PR quadtree collapsing can only take place if two of the brothers of *P* are empty. On the other hand, in the Rectangle quadtree collapsing can take place as long as all of *P*'s brothers contain r-pieces of the same rectangle. For example, when rectangle J is deleted from Figure 2, the result is that nodes 26, 27, 28, and 29 are merged to yield node 25, which is in turn merged with nodes 22, 23, and 24 to yield node 21 (see the resulting block decomposition in Figure 5).

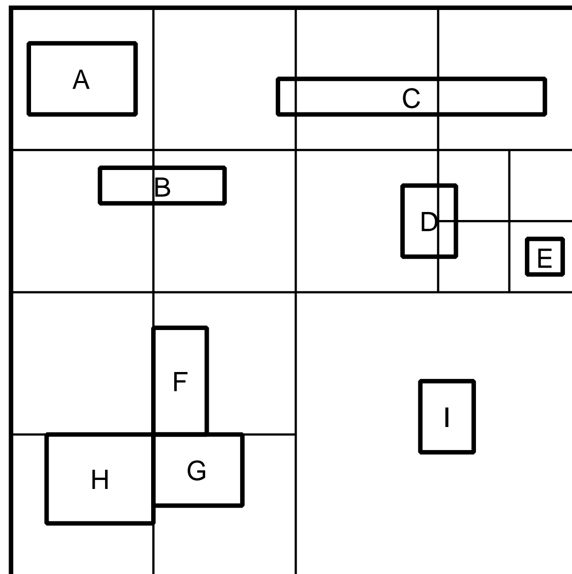

Figure 5: Rectangle quadtree result of deleting rectangle J from the collection of rectangles given in Figure 1.

3.3 Search

The most common search query is one that seeks to determine if a given rectangle overlaps (i.e., intersects) any of the existing rectangles. This operation is a prerequisite to the successful insertion of a rectangle. Range queries can also be performed. However, they are more usefully cast in terms of finding all the rectangles in a given area (i.e., a window query). Another popular query is one that seeks to determine if one collection of rectangles can be overlaid on another collection without any of the component rectangles intersecting one another.

These two operations can be implemented by using variants of algorithms developed for handling set operations (i.e., union and intersection) in region-based quadtrees [3,7]. The range query is answered by intersecting the query rectangle with the Rectangle quadtree. The First, intersect the two Rectangle quadtrees. If the result is empty, then they can be safely overlaid and we merely need to perform a union of the two

Rectangle quadtrees. It should be clear that Boolean queries can be easily handled. An example JAVA applet for the Rectangle quadtree data structure can be found on the home page of the class.

4 Assignment

This assignment has four parts. It is to be programmed in C or C++. JAVA is not permitted. You are not allowed to make use of any built in data structures from any library such as, but not limited to, STL in C++. 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 part is to be turned at the next class meeting after this assignment has been distributed to you. It is worth 10 points. The second part is also worth 10 points. It is to be turned in one week after you turn in the data structure. There will be NO late submissions accepted for these two parts of the assignment. 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 or C++ program to implement the data structure and the specified operations. Together they are worth 60 points. Part three consists of operations (1)-(9) given below. They are worth a total of 30 points, with varying point values for the different operations. Part four consists of operations (10)-(14) given below. They are worth 30 points. Operations (15)-(17) are for extra credit and are to be turned in with part four. They are worth up to 4 points apiece.

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 part of the assignment. For any operation that is not implemented, say `OP`, your command decoder must output a message of the form ``‘‘COMMAND OP IS NOT IMPLEMENTED’’``.

In order to facilitate your program as well as lend some realism to your task you are to implement the Rectangle quadtree in a raster-based graphics environment. This means that you are dealing with a world of pixels. The size of the world can be varied, and is a $2^w \times 2^w$ array of pixels. As a default, you should assume $w = 7$, i.e., a size of 128×128. The pixel at the lower left corner has coordinate values (0,0) and the pixel at the upper right corner has coordinate values $(2^w - 1, 2^w - 1)$. 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. Note that the endpoints and widths of the rectangles will be restricted to integers. All rectangles are of size $(3+i) \times (3+j)$, where $0 \leq i \leq 125$ and $0 \leq j \leq 125$. In other words, the smallest rectangle is of size 3×3 and the largest is 128×128.

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. In addition, you are also to test your code with some randomly generated data, which in this case is a randomly generated rectangle. You should come up with a reasonable way of generating random rectangles. You should think about what it means to generate a random rectangle and about their expected sizes so that they are well-distributed rather than all being of approximately the same size, and whether they have a high likelihood of intersecting. This will require that you examine the types of rectangles that you are generating.

4.1 Data Structure Selection

You are to select a data structure to implement the Rectangle quadtree. Turn in a definition in the form of a set of C++ classes or C structs. Again, you are not allowed to make use of any built in data structures from any library such as, but not limited to, STL in C++. In doing this part of the assignment you should bear in mind the type of data that is being represented and the type of of operations that will be performed on it. In order to ease your task, remember that the primitive entity is the rectangle. We specify a rectangle by giving the x and y coordinate values of its lower left corner, and the horizontal and vertical distances to its borders (i.e., the lengths of its sides). 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 rectangle. Each rectangle is assigned a unique name. 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 data structure that keeps track of the geometry of the rectangles.

4.2 Command Decoder

You are to turn in a working command decoder written in C or 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.

The output for the command decoder consists of the number of the operation (e.g., “1” for command `INIT_QUADTREE`) and the actual values of the parameters if the command has any parameters (e.g., the value of `WIDTH` for the `INIT_QUADTREE` command).

4.3 Part Three: Basic Operations

In order to facilitate grading of these operations as well as the advanced and optional operations in Sections 4.4 and 4.5, respectively, please provide a trace output of the execution of the operations which lists the nodes (both leaf and nonleaf) that have been visited while executing the operation. This trace is initiated by the command `TRACE ON` and is terminated by the command `TRACE OFF`. In order for the trace output to be concise, you are to represent each node of the rectangle quadtree that has been visited by a unique number which is formed as follows. The root of the quadtree is assigned the number 0. Given a node with number N, its `NW`, `NE`, `SW`, and `SE` children are labeled $4 \cdot N+1$, $4 \cdot N+2$, $4 \cdot N+3$, and $4 \cdot N+4$, respectively. For example, starting at the root, the `NE` child is numbered 2, while the `SE` child of the `NW` child of the root is numbered 4*(4*0+1)+4=8.

(1) (1 point) 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 $2^{\texttt{WIDTH}}$. It also has the effect of starting with a fresh data set.

(2) (1 point) Generate a display of a $2^{\texttt{WIDTH}} \times 2^{\texttt{WIDTH}}$ square from the Rectangle quadtree. It is invoked by the command `DISPLAY()`. To draw the Rectangle 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 rectangles should be solid (i.e., not dashed). Rectangle names should

be output somewhere near the rectangle or within the rectangle. When this convention causes the output of a quadrant line to coincide with the output of the boundary of a rectangle, then the output of the rectangle takes precedence and the coincident part of the quadrant line is not output.

(3) (3 points) List all the rectangles 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_RECTANGLES()` and yields for each rectangle its name, the *x* and *y* coordinate values of its lower left corner, and the horizontal and vertical distances to its borders from the lower left corner (i.e., the lengths of its sides). This is of use in interpreting the display since sometimes it is not possible to distinguish the boundaries of the rectangles from the display. You should list all of the rectangles in the database whether or not they have been deleted.

(4) (1 point) Create a rectangle by specifying the coordinate values of its lower left corner and the distances to its borders, and assign it a name for subsequent use. It is invoked by the command `CREATE_RECTANGLE(N,LLX,LLY,LX,LY)` where `N` is the name to be associated with the rectangle, `LLX` and `LLY` are the *x* and *y* coordinate values, respectively, of its lower left corner, and `LX` and `LY` are the horizontal and vertical distances, respectively, to its borders from the lower left corner. `LLX`, `LLY`, `LX`, and `LY` must be integer numbers. Output an appropriate message indicating that the rectangle has been created as well as its name and endpoints. Note that any rectangle can be created — even if it is outside the space spanned by the Rectangle quadtree.

There is also a variant of this query called `CREATE_RECTANGLE_RANDOM(N)` that generates a rectangle at random which means that `LLX`, `LLY`, `LX`, and `LY` are generated at random subject to the above conditions that these values are integers in the appropriate range.

(5) (5 points) Determine whether a query rectangle intersects (i.e., overlaps) any of the existing rectangles. This operation is a prerequisite to the successful insertion of a rectangle in the Rectangle quadtree. It is invoked by the command `RECTANGLE_SEARCH(N)` where `N` is a name of a rectangle. If the rectangle does not intersect an existing rectangle, then `RECTANGLE_SEARCH` returns a value of false and outputs an appropriate message such as `‘‘N DOES NOT INTERSECT AN EXISTING RECTANGLE’’`. Otherwise, it returns the value true and uses the name associated with one of the intersecting rectangles (i.e., if it intersects more than one rectangle) to output the following two messages: `‘‘N INTERSECTS RECTANGLE [NAME OF RECTANGLE]’’`. Note that if an endpoint of the query rectangle touches the endpoint of an existing rectangle, then `RECTANGLE_SEARCH` returns false. You are only to check against the rectangles that are in the Rectangle quadtree of existing rectangles, and not the rectangles that existed at some time in the past and have been deleted by the time this command is executed.

(6) (5 points) Insert a rectangle in the Rectangle quadtree. If the rectangle intersects an existing rectangle, then do not make the insertion and report this fact by returning the name of the intersecting rectangle. Also, if any part of the rectangle is outside the space spanned by the Rectangle quadtree, then do not make the insertion and report this fact by a suitable message such as `INSERTION OF RECTANGLE N FAILED AS N LIES PARTIALLY OUTSIDE SPACE SPANNED BY RECTANGLE QUADTREE`. Otherwise, return the name of the rectangle that is being inserted as well as output a message indicating that this has been done. It is invoked by the command `INSERT(N)` where `N` is the name of a rectangle. It should be clear that the Rectangle quadtree is built by a sequence of `CREATE_RECTANGLE` and `INSERT` operations.

(7) (4 points) Given a point, return the name of the rectangle that contains it. It is invoked by the command `SEARCH_POINT(PX,PY)` where `PX` and `PY` are the *x* and *y* coordinate values, respectively, of the point. If no such rectangle exists, then output a message indicating that the point is not contained in any of the rectangles.

(8) (6 points) Delete a rectangle or a set of rectangles from the Rectangle quadtree. This operation has two variants, `DELETE_RECTANGLE` and `DELETE_POINT`. The command `DELETE_RECTANGLE(N)` deletes the rectangle named `N`. It returns `N` if it was successful in deleting the specified rectangle and outputs a message indicating it. Otherwise, it outputs an appropriate message. The command `DELETE_POINT(PX,PY)` has as

its argument a point within the rectangle to be deleted whose *x* and *y* coordinate values are given by PX and PY, respectively. DELETE_POINT returns as its value the name of the rectangle that has been deleted and prints an appropriate message indicating its name. If the point is not in any rectangle, then an appropriate message indicating this is output. The code for DELETE_POINT should make use of SEARCH_POINT. Note that rectangle N is only deleted from the Rectangle quadtree and not from the database of rectangles.

(9) (4 points) Move a rectangle in the Rectangle quadtree. The command is invoked by MOVE(N,CX,CY) where N is the name of the rectangle, CX, CY are the translation of the centroid of N across the *x* and *y* coordinate axes, and they must be integers The command returns N if it was successful in moving the specified rectangle and outputs a message indicating it. Otherwise, output appropriate error messages if N was not found in the Rectangle quadtree, or if after the operation N lies outside the space spanned by the Rectangle quadtree. Note that the successful execution of the operation requires that the moved rectangle does not overlap any of the existing rectangles in which case an appropriate error message is emitted.

4.4 Part Four: Advanced Operations

(10) (6 points) Determine all the rectangles in the Rectangle quadtree that touch (i.e., are adjacent along a side or a corner) a given rectangle. This operation is invoked by the command TOUCH(N) where N is the name of a rectangle. Since rectangle N is referenced by name, N thus must be in the database for the operation to work but it need not necessarily be in the Rectangle quadtree. The command returns the names of all the touched rectangles in conjunction with the following message ‘‘N SHARES ENDPOINT [X AND Y COORDINATE VALUES OF ENDPOINT] WITH THE RECTANGLES [NAME OF RECTANGLES]’’. Otherwise, the command returns NIL. For each rectangle *r* that touches N, display (i.e., highlight) the point in *r* for which the *x* and *y* coordinate values are minimum (i.e., the lower-leftmost corner). It should be clear that the intersection of *r* with N is empty.

(11) (6 points) Determine all of the rectangles in the Rectangle quadtree that lie within a given distance of a given rectangle. This is the so-called ‘lambda’ problem. Given a distance D (an integer here although it could also be a real number in the more general case), it is invoked by the command WITHIN(N,D) where N is the name of the query rectangle. In essence, this operation constructs a query rectangle Q with the same centroid as N and distances LX+D and LY+D to the border. Now, the query returns the identity of all rectangles whose intersection with the region formed by the difference of Q and N is not empty (i.e,, any rectangle *r* that has at least one point in common with Q-N). In other words, we have a shell of width D around N and we want all the rectangles that have a point in common with this shell. Rectangle N need not necessarily be in the Rectangle quadtree. Note that for this operation you must recursively traverse the tree to find the rectangles that overlap the query region. You will NOT be given credit for a solution that uses neighbor finding, such as one (but not limited to) that starts at the centroid of N and finds its neighbors in increasing order of distance. This is the basis of another operation.

(12) (6 points) Find the nearest neighboring rectangle in the horizontal and vertical directions, respectively, to a given rectangle. To locate a horizontal neighbor, use the command HORIZ_NEIGHBOR(N) where N is the name of the query rectangle. VERT_NEIGHBOR(N) locates a vertical neighbor. By “nearest” horizontal (vertical) neighboring rectangle, it is meant the rectangle whose vertical (horizontal) side, or extension, is closest to a vertical (horizontal) side of the query rectangle. If the vertical (horizontal) extension of a side of rectangle *r* causes the extended side of *r* to intersect the query rectangle, then *r* is deemed to be at distance 0 and is thus not a candidate neighbor. In other words, the distance has to be greater than zero. The commands return as their value the name of the neighboring rectangle if one exists and NIL otherwise as well as an appropriate message. Rectangle N need not necessarily be in the Rectangle quadtree. If more than one rectangle is at the same distance, then return the name of just one of them. Note that rectangles that are inside N are not considered by this query.

(13) (6 points) Given a point, return the name of the nearest rectangle. By "nearest," it is meant the rectangle whose side or corner is closest to the point. Note that this rectangle could also be a rectangle that contains the point. In this case, the distance is zero. It is invoked by the command `NEAREST_RECTANGLE(PX,PY)` where `PX` and `PY` are the *x* and *y* coordinate values, respectively, of the point. If no such rectangle exists (e.g., when the tree is empty), then output an appropriate message (i.e., that the tree is empty). If more than one rectangle is at the same distance, then return the name of just one of them.

(14) (6 points) Find all rectangles in a rectangular window anchored at a given point. It is invoked by the command `WINDOW(LLX,LLY,LX,LY)` where `LLX` and `LLY` are the *x* and *y* coordinate values, respectively, of the lower left corner of the window and `LX` and `LY` are the horizontal and vertical distances, respectively, to its borders from the corner. Your output is a list of the names of the rectangles that are completely inside the window, and a display of the Rectangle quadtree that only shows the rectangles that are in the window. This is similar to a clipping operation. Draw the boundary of the window using a dashed rectangle. Do not show quadrant lines within the window. All arguments to `WINDOW` are integers (i.e., `LX`, `LY` `LLX`, and `LLY`). Note that for this operation you must recursively traverse the tree to find the rectangles that overlap the query region. You will NOT be given credit for a solution that uses neighbor finding, such as one (but not limited to) that starts at the centroid of the window and finds its neighbors in increasing order of distance. This is the basis of another operation.

4.5 Optional Operations

(15) (4 points) Find the nearest neighbor in all directions to the boundary of a given rectangle. It is invoked by the command `NEAREST_NEIGHBOR(N)` where `N` is the name of a rectangle. By "nearest," it is meant the rectangle *C* with a point on its side or corner, say *P*, such that the distance from *P* to a side or corner of the query rectangle is a minimum. `NEAREST_NEIGHBOR` returns as its value the name of the neighboring rectangle if one exists and `NIL` otherwise as well as an appropriate message. Rectangle `N` need not necessarily be in the Rectangle quadtree. If more than one rectangle is at the same distance, then return the name of just one of them. Note that rectangles that are inside `N` are not considered by this query. Note that rectangles that are inside `N` are not considered by this query.

(16) (4 points) Given a rectangle, find its nearest neighbor with a name that is lexicographically greater. It is invoked by the command `LEXICALLY_GREATER_NEAREST_NEIGHBOR(N)` where `N` is the name of a rectangle. By "lexically greater nearest" it is meant the rectangle *C* whose name is lexicographically greater than that of `N` with a point on `C`'s side, say *P*, such that the distance from *P* to a side of the query rectangle is a minimum. `LEXICALLY_GREATER_NEAREST_NEIGHBOR` returns as its value the name of the neighboring rectangle if one exists and `NIL` otherwise as well as an appropriate message. Rectangle `N` need not necessarily be in the Rectangle quadtree. If more than one rectangle is at the same distance, then return the name of just one of them. Note that rectangles that are inside `N` are not considered by this query. This operation should not examine more than the minimum number of rectangles that are necessary to determine the lexicographically greater nearest neighbor. Thus you should use an incremental nearest neighbor algorithm (e.g., [2]).

(17) (4 points) Perform connected component labeling on the Rectangle quadtree. This means that all touching rectangles are assigned the same label. By "touching," it is meant that the rectangles are adjacent along a side or a corner. This is accomplished by the command `LABEL()`. The result of the operation is a display of the Rectangle quadtree where all touching rectangles are shown with the same label. Use integer labels.

5 Hints

In the following, we represent every point by its x and y coordinate values and every rectangle by a pair of points. For example, $R = (p_1, p_2)$, is a rectangle with lower left corner at $p_1 = (x_1, y_1)$ and upper right corner at $p_2 = (x_2, y_2)$.

Because we represent points by two dimensional vectors, we can define algebraic operations on points. For example given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, we can define $p_1 + p_2$ to be the point $(x_1 + x_2, y_1 + y_2)$. Similarly, we can define scalar multiplication (e.g. $5p_1 = (5x_1, 5y_1)$), subtraction, etc. Given a vector $v = (x, y)$, we denote the length of the vector v by $||v||$ which is given by $||v|| = \sqrt{(x^2 + y^2)}$. Notice that for a point $p = (x, y)$, $||p||$ is the length of the vector from $(0, 0)$ to p.

- **Comparing Points:**
  Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. We say $p_1 < p_2$ if and only if both $x_1 < x_2$ and $y_1 < y_2$. Similarly, we say $p_1 \leq p_2$ if and only if both $x_1 \leq x_2$ and $y_1 \leq y_2$. Note that based on this definition it is possible that neither $p_1 \leq p_2$ nor $p_2 \leq p_1$ (e.g. consider the two points $p_1 = (0, 1)$ and $p_2 = (1, 0)$).

- **Inside:**
  Given a point p and a rectangle $R = (p_1, p_2)$, the point p is *inside* R if and only if $p_1 \leq p < p_2$.

- **Min/Max:**
  Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, we define the min and max operations as follows:

  $$\min(p_1, p_2) = (\min(x_1, x_2), \min(y_1, y_2))$$
  $$\max(p_1, p_2) = (\max(x_1, x_2), \max(y_1, y_2))$$

  Similarly, we can define the min and max for more that two points (e.g. $\max(p_1, p_2, \cdots, p_n)$). In other words, the minimum/mamximum of several points is a point that has the minimum/maximum of their coordinate values.

- **Valid Rectangle:**
  Given a rectangle $R = (p_1, p_2)$, we say a rectangle is *valid* if and only if $p_1 \leq p_2$, otherwise the rectangle R is *invalid*.

- **Empty Rectangle:**
  A rectangle $R = (p_1, p_2)$ is *empty* if and only if R is a valid rectangle and $p_1 < p_2$ is false. In other words, R is a *valid* but *empty* rectangle if and only if $p_1 \leq p_2$ is true but $p_1 < p_2$ is false.

- **Intersection:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, let $p''_1 = \max(p_1, p'_1)$ and $p''_2 = \min(p_2, p'_2)$. Then R and R' *intersect* if and only if the rectangle $R'' = (p''_1, p''_2)$ is a *valid* and not *empty* rectangle. In other words, they intersect if and only if $p''_1 < p''_2$. If they do intersect then their intersection is given by the rectangle R'' defined above.

- **Touch:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, we say that R and R' *touch* if their intersection is a valid but empty rectangle. In other words: Let $p''_1 = \max(p_1, p'_1)$ and $p''_2 = \min(p_2, p'_2)$. Then R and R' touch if and only if $p''_1 \leq p''_2$ is true but $p''_1 < p''_2$ is false.

- **Rectangle Containment:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, we say R' is contained in R if and only if $p_1 \leq p'_1$ and $p'_2 \leq p_2$. This is also equivalent to saying R' is contained in R if and only if the intersection of R' and R is R' itself.

- **Point-Point Distance:**
  Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ let $d = p_1 - p_2$ be their difference vector (i.e. the vector connecting p_1 to p_2). The distance between p_1 and p_2 is given by $||d||$. In other words the distance between p_1 and p_2 is $||p_1 - p_2||$ which is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

- **Point-Rectangle Distance:**
  Given a point p and a rectangle $R = (p_1, p_2)$, let $d = \max(p_1 - p, p - p_2, (0,0))$ be their difference vector. Then, the distance between p and R is given by $||d||$.

- **Rectangle-Rectangle Distance:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, let $d = \max(p_1 - p'_2, p'_1 - p_2, (0,0))$ be their difference vector. Then, the distance between R and R' is given by $||d||$.

- **Horizontal Distance**
  The horizontal distance between two objects is defined as the distance between the projection of the two objects on the X-axis. The projection of a rectangle $R = ((x_1, y_1), (x_2, y_2))$ on the X-axis is the half-open interval $[x_1, x_2)$. So, the horizontal distance between two rectangles is just the distance between their projections on the X-axis. So, given the rectangle $R' = ((x'_1, y'_1), (x'_2, y'_2))$, we can compute the horizontal distance of R and R' by $\max(x_1 - x'_2, x'_1 - x_2)$. Notice that if the projections of R and R' overlap then this distance could be negative.

- **Vertical Distance**
  The vertical distance between two objects is defined as the distance between the projections of the two objects on the Y-axis. This is defined similarly to the horizontal distance.

You will need to use a priority queue to implement some of the operation (i.e. `HORIZ_NEIGHBOR`, `VERT_NEIGHBOR`, `NEAREST_RECTANGLE`, `NEAREST_NEIGHBOR`, and `LEXICALLY_GREATER_NEAREST_NEIGHBOR`). The following is a pseudo-code which should give you an idea of how you should implement these operations. Note that the following may miss the details specific to each operation so you may need to modify it or add to it to implement each operations correctly.

Notice that Q is a priority queue of quad-tree nodes in which nodes with shorter distance to the *query* come first. If two nodes have the same distance to the *query* then the one with a lower quad-tree number comes first. Also, notice that *distance* is defined based on the operation you are implementing. For each specific operation you may need to check for other conditions. For example, if you are implementing the `VERT_NEIGHBOR` then at line 1 you should also check that the image of p on the vertical axis is not entirely *contained* in the image of the *query* on the vertical axis (because if it is so then p cannot possible contain the solution). Similarly, at line 1 you should check that the image of r on the vertical axis does not overlap the image of the *query* on the vertical axis (remember that this is for `VERT_NEIGHBOR`, for other operations you need to check for other conditions).

```
Find_Nearest (Object query);
Let Q be a priority queue of quad-tree nodes ;
min_dist ← ∞ ;
closest_rect ← null ;
Push the root of the quad-tree into Q ;
while Q is not empty do
    p ← pop the next quad-tree node from the head of Q ;
    d ← distance of p from the query;
    if d < min_dist then            // You may need to check for other conditions too
        if tracing is enabled then
            print the quad-tree node number of p ;
        if p is a gray node then
            Push all of the child nodes of p into Q ;
        else if p is a black node then
            r ← the rectangle in p ;
            d′ ← distance of r from the query;
            else if d′ < min_dist then            // Check other conditions too
                min_dist ← d′ ;
                closest_rect ← p ;
end
return closest_rect ;
```

6 Sample Input/Output

Here is a sample input:

```
INIT_QUADTREE(8)
LIST_RECTANGLES()
CREATE_RECTANGLE(R1,5,5,25,25)
CREATE_RECTANGLE(R2,20,20,31,31)
CREATE_RECTANGLE(R3,30,30,40,40)
CREATE_RECTANGLE(R4,200,200,210,210)
TRACE ON
INSERT(R4)
SEARCH_POINT(1,1)
INSERT(R1)
INSERT(R4)
INSERT(R2)
SEARCH_POINT(1,1)
INSERT(R3)
SEARCH_POINT(5,5)
TRACE OFF
LIST_RECTANGLES()
RECTANGLE_SEARCH(R2)
SEARCH_POINT(1,1)
INSERT(R2)
SEARCH_POINT(7,7)
```

```
INSERT(R2)
DELETE_POINT(10,10)
INSERT(R2)
INIT_QUADTREE(8)
INSERT(R4)
INSERT(R4)
CREATE_RECTANGLE(S1,5,5,10,10)
CREATE_RECTANGLE(S2,10,10,13,13)
CREATE_RECTANGLE(S3,14,14,20,20)
CREATE_RECTANGLE(S4,13,2,14,24)
INSERT(S1)
INSERT(S3)
TOUCH(S2)
TRACE ON
HORIZ_NEIGHBOR(R2)
TRACE OFF
VERT_NEIGHBOR(R4)
INSERT(R3)
VERT_NEIGHBOR(R4)
INSERT(R4)
INSERT(R4)
INSERT(R4)
NEAREST_RECTANGLE(27,26)
NEAREST_RECTANGLE(50,50)
NEAREST_RECTANGLE(130,130)
NEAREST_RECTANGLE(160,160)
INSERT(S2)
LABEL()
INSERT(S4)
LABEL()
```

Here is the corresponding output:

```
INIT_QUADTREE(8): initialized a quadtree of width 256
LIST_RECTANGLES(): listing 0 rectangles:
CREATE_RECTANGLE(R1,5,5,25,25): created rectangle R1
CREATE_RECTANGLE(R2,20,20,31,31): created rectangle R2
CREATE_RECTANGLE(R3,30,30,40,40): created rectangle R3
CREATE_RECTANGLE(R4,200,200,210,210): created rectangle R4
TRACE ON
INSERT(R4): inserted rectangle R4
SEARCH_POINT(1,1)[ 0]: no rectangle found
INSERT(R1): inserted rectangle R1
INSERT(R4): failed: intersects with R4
INSERT(R2): failed: intersects with R1
SEARCH_POINT(1,1)[ 0 3]: no rectangle found
INSERT(R3): inserted rectangle R3
SEARCH_POINT(5,5)[ 0 3 15 63 255]: found rectangle R1
TRACE OFF
LIST_RECTANGLES(): listing 4 rectangles:
R1 5 5 25 25
R2 20 20 31 31
R3 30 30 40 40
R4 200 200 210 210
RECTANGLE_SEARCH(R2): found 2 rectangles: R1 R3
SEARCH_POINT(1,1): no rectangle found
INSERT(R2): failed: intersects with R1
SEARCH_POINT(7,7): found rectangle R1
INSERT(R2): failed: intersects with R1
DELETE_POINT(10,10): deleted rectangle R1
INSERT(R2): failed: intersects with R3
INIT_QUADTREE(8): initialized a quadtree of width 256
INSERT(R4): inserted rectangle R4
INSERT(R4): failed: intersects with R4
CREATE_RECTANGLE(S1,5,5,10,10): created rectangle S1
CREATE_RECTANGLE(S2,10,10,13,13): created rectangle S2
CREATE_RECTANGLE(S3,14,14,20,20): created rectangle S3
CREATE_RECTANGLE(S4,13,2,14,24): created rectangle S4
INSERT(S1): inserted rectangle S1
INSERT(S3): inserted rectangle S3
TOUCH(S2): found 1 rectangles: S1
TRACE ON
HORIZ_NEIGHBOR(R2)[ 0 1 3 13 15 61 63 254]: found rectangle S3
TRACE OFF
VERT_NEIGHBOR(R4): found rectangle S3
INSERT(R3): inserted rectangle R3
VERT_NEIGHBOR(R4): found rectangle R3
INSERT(R4): failed: intersects with R4
INSERT(R4): failed: intersects with R4
```

```
INSERT(R4): failed: intersects with R4
NEAREST_RECTANGLE(27,26): found rectangle R3
NEAREST_RECTANGLE(50,50): found rectangle R3
NEAREST_RECTANGLE(130,130): found rectangle R4
NEAREST_RECTANGLE(160,160): found rectangle R4
INSERT(S2): inserted rectangle S2
LABEL(): found 4 connected components:
R3 R3
R4 R4
S1 S1
S2 S1
S3 S3
INSERT(S4): inserted rectangle S4
LABEL(): found 3 connected components:
R3 R3
R4 R4
S1 S1
S2 S1
S3 S1
S4 S1
```

References

[1] R. A. Finkel and J. L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1(1974), 1–9.

[2] G. R. Hjaltason and H. Samet, Distance browsing in spatial databases, *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also Computer Science TR-3919, University of Maryland, College Park, MD, and *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 83–95, Portland, ME, August 1995, and Springer-Verlag Lecture Notes in Computer Science 951.

[3] 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.

[4] A. Klinger, Patterns and Search Statistics, in *Optimizing Methods in Statistics*, J. S. Rustagi, Ed., Academic Press, New York, 1971, 303–337.

[5] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[6] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[7] M. Shneier, Calculations of geometric properties using quadtrees, *Computer Graphics and Image Processing 16*, 3(July 1981), 296–302.