Fall 2021

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) as well as game programming 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 the MX-CIF quadtree, a variant of the quadtree data structure that has been found to be useful for such a problem. Your task is to implement a MX-CIF 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. definitions. 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)–(18) are optional and you will get extra credit if you turn them in with the completed part four on time. If you are a graduate student, part four is not optional.

[1]Copyright ©2021 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 [2] 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 A[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 A, 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 [6, 8, 7]. 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. 28–48 and 423–426 in [9].

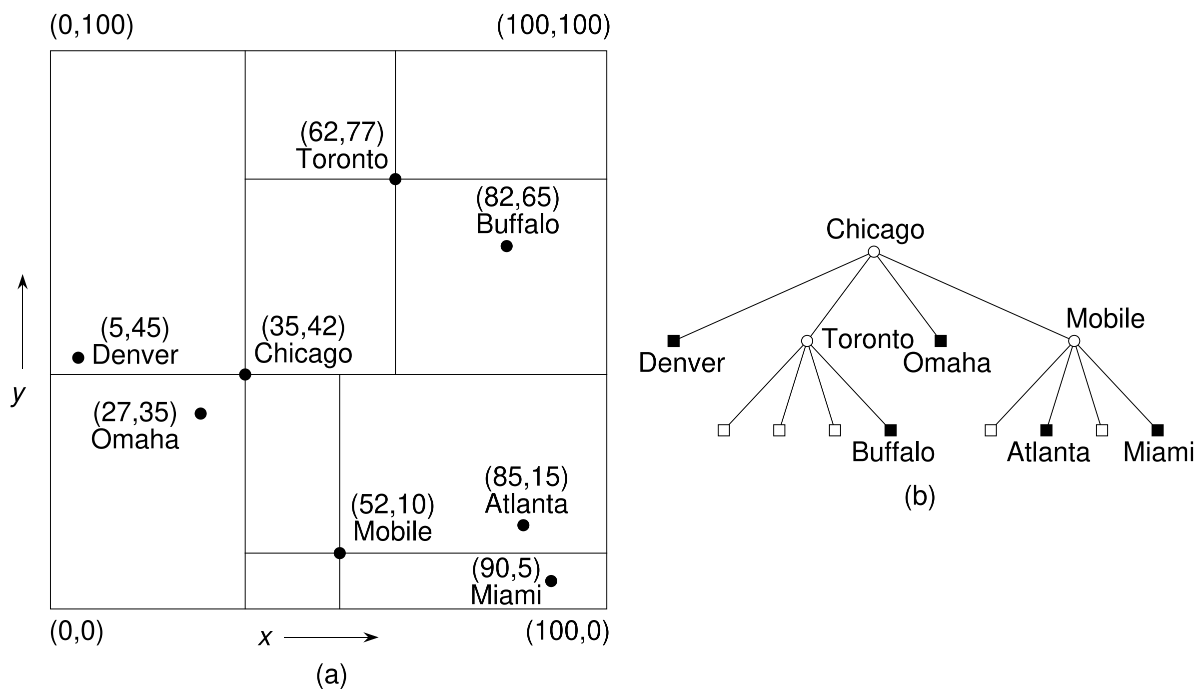


Figure A: A point quadtree and the records it represents corresponding to the data of Figure D: (a) the resulting partition of space and (b) the tree representation.

As an example of the region quadtree, consider the region shown in Figure Ba which is represented by a $2^3 \times 2^3$ binary array in Figure Bb. 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 Bb are shown in Figure Bc This process is represented by a quadtree in which the root node corresponds to the entire array, the four

[2] All page numbers refer to [9] while figure labels are upper case letters which are numbers in [9].

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 Bc is shown in Figure Bd.

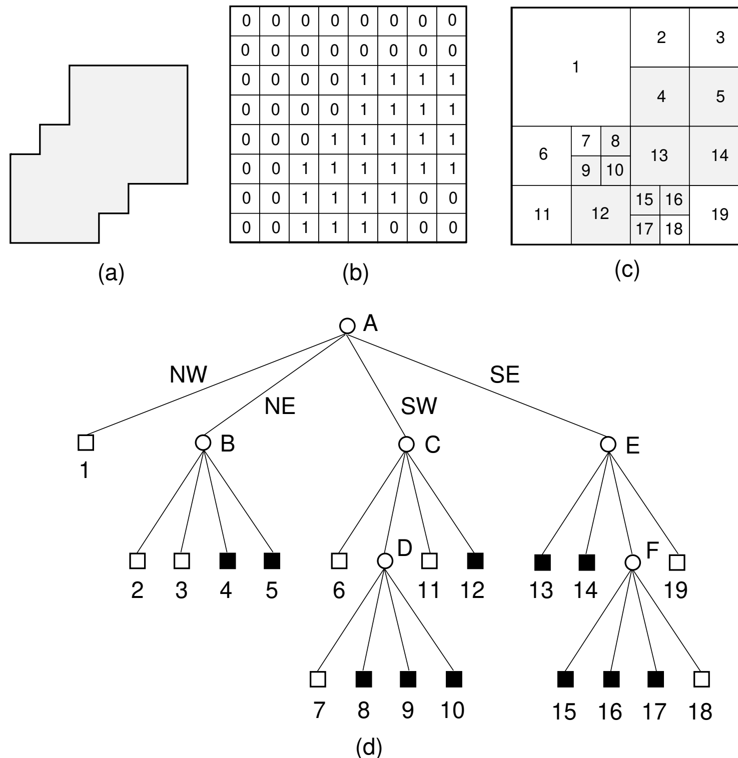


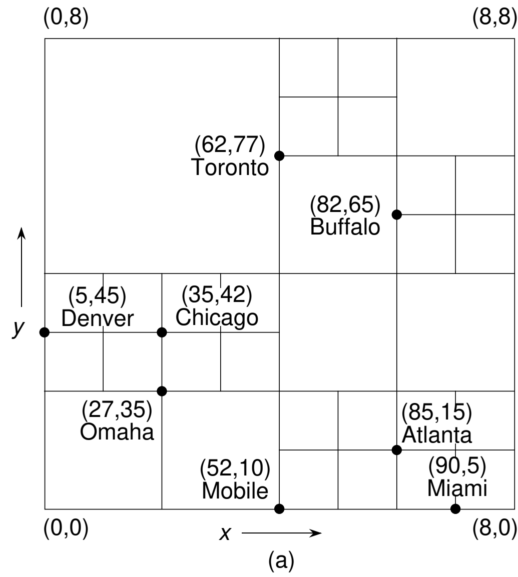
Figure B: (a) Sample region, (b) its binary array representation, (c) its maximal blocks with the blocks in the region being shaded, and (d) the corresponding quadtree.

2 MX 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 were BLACK pixels in a region quadtree. An alternative characterization is to treat the data points as non-zero elements in a square matrix. We shall use this characterization in the subsequent discussion. To avoid confusion with the point and region quadtrees, we call the resulting data structure an *MX quadtree* (MX for matrix).

The MX quadtree is organized in a similar way to the region quadtree. The difference is that leaf nodes are BLACK or empty (i.e., WHITE) corresponding to the presence or absence, respectively, of a data point in the appropriate position in the matrix. For example, Figure C is the $2^3 \times 2^3$ MX quadtree corresponding to the data of Figure D. It is obtained by applying the mapping f such that $f(z) = z \div 12.5$ to both x and y coordinates. The result of the mapping is reflected in the coordinate values in the figure.

Each data point in an MX quadtree corresponds to a 1×1 square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left corner of the square. This adheres to the general convention followed throughout this presentation that the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. We also



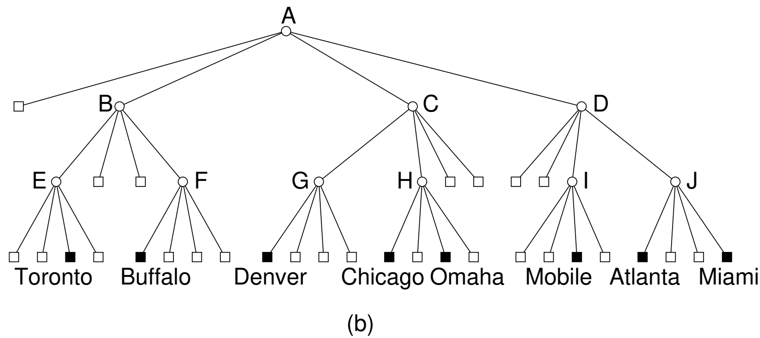


Figure C: An MX quadtree and the records it represents corresponding to the data of Figure D: (a) the resulting partition of space and (b) the tree representation.

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

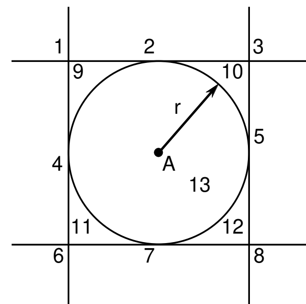
Figure D: Sample list of cities with their x and y coordinate values.

assume that the lower left corner of the matrix is located at (0,0). Note that, unlike the region quadtree, when a non-leaf node in the MX quadtree 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, as each data point is different. On the other hand, the empty leaf nodes have the absence of information as their common property; so, four WHITE sons of a non-leaf node can be safely merged.

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 quadtree-like data structures lies in their 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 point quadtree of Figure A although the extension to an MX quadtree is straightforward. Suppose that we wish to find all cities within eight units of a data point with coordinate values (83,10). In such a case, there is no need to search the `NW`, `NE`, and `SW` quadrants of the root (i.e., Chicago with coordinate values (35,40)). Thus, we can restrict our search to the `SE` quadrant of the tree rooted at Chicago. Similarly, there is no need to search the `NW` and `SW` quadrants of the tree rooted at Mobile (i.e., coordinate values (50,10)).

As a further clarification of the amount of pruning of the search space that is achievable by use of the point quadtree, we make use of Figure E In particular, given the problem of finding all nodes within radius r of point A, use of the figure indicates which quadrants need not be examined when the root of the search space, say R, is in one of the numbered regions. For example, if R is in region 9, then all but its `NW` quadrants must be searched. If R is in region 7, then the search can be restricted to the `NW` and `NE` quadrants of R. For more details on MX quadtrees, see pp. 38–42.



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

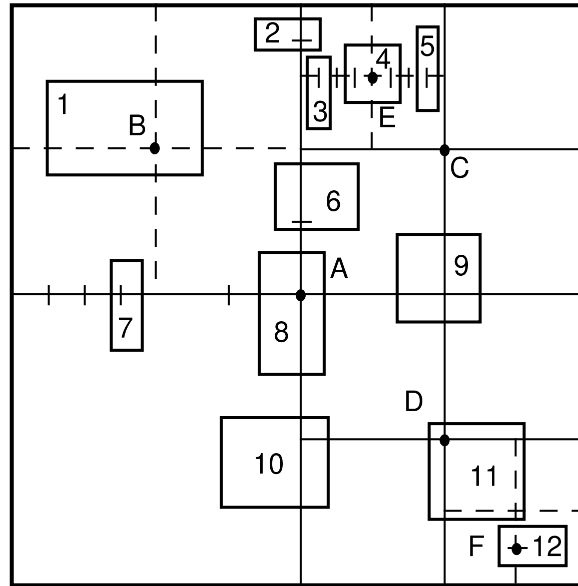
Figure E: Relationship between a circular search space and the regions in which a root of a quadtree may reside.

3 MX-CIF Quadtrees

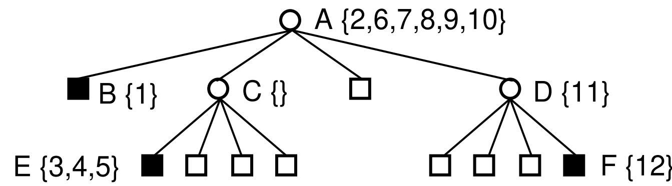
The *MX-CIF quadtree* is a quadtree-like data structure devised by Kedem [5] (who used the term *quad-CIF tree*) for representing a large set of very small rectangles for application in VLSI design rule checking. The goal is to rapidly locate a collection of objects that intersect a given rectangle. An equivalent problem is to insert a rectangle into the data structure under the restriction that it does not intersect existing rectangles.

The MX-CIF quadtree is organized in a similar way to the region quadtree. A region is repeatedly subdivided into four equal-size quadrants until we obtain blocks which do not contain rectangles. As the

subdivision takes place, we associate with each subdivision point a set containing all of the rectangles that intersect the lines passing through it. For example, Figure F contains a set of rectangles and its corresponding MX-CIF quadtree. Once a rectangle is associated with a quadtree node, say *P*, it is not considered to be a member of any of the sons of *P*. For example, in Figure F, rectangle 11 overlaps the space spanned by both nodes D and F but is only associated with node D, while rectangle 12 is associated with node F.

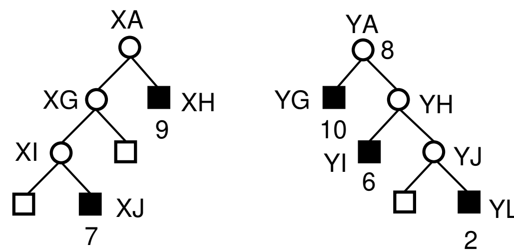


(a)



(b)

Figure F: (a) Collection of rectangles and the block decomposition induced by its MX-CIF quadtree; (b) the tree representation of (a).



(a) (b)

Figure G: (a) Binary trees for the (a) x axis and (b) y axis passing through node A in Figure F.

At this point, it is also appropriate to comment on the relationship between the MX-CIF quadtree and the MX quadtree. The similarity is that the MX quadtree is defined for a domain of points with corresponding

nodes that are the smallest blocks in which they are contained. Similarly, the domain of the MX-CIF quadtree consists of rectangles with corresponding nodes that are the smallest blocks in which they are contained. In both cases, there is a predetermined limit on the level of decomposition. One major difference is that in the MX-CIF quadtree, unlike the MX quadtree, all nodes are of the same type. Thus, data is associated with both leaf and non-leaf nodes of the MX-CIF quadtree. Empty nodes in the MX-CIF quadtree are analogous to WHITE nodes in the MX quadtree. An empty node is like an empty son and is represented by a NIL pointer in the direction of a quadrant that contains no rectangles. For more details on MX-CIF quadtrees, see pp. 466–473 in [9]. It should be clear that more than one rectangle can be associated with a given enclosing block (i.e., node). There are several ways of organizing these rectangles. Abel and Smith [1] do not apply any ordering. This is equivalent to maintaining a linked list of the rectangles. Another approach, devised by Kedem [5] (who used the term *quad-CIF tree*) is described below.

Let P be a quadtree node and let S be the set of rectangles that are associated with P. Members of S are organized into two subsets according to their intersection (or the colinearity of their sides) with the lines passing through the centroid of P's block. We shall use the terms *axes* or *axis lines* to refer to these lines. For example, consider node P whose block is of size $2 \cdot LX \times 2 \cdot LY$ and is centered at (CX,CY). All members of S that intersect the line $x = CX$ form one subset and all members of S that intersect the line $y = CY$ form the other subset. Equivalently, these subsets correspond to the rectangles intersecting the y and x axes, respectively, passing through (CX,CY). If a rectangle intersects both axes (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the subset associated with the y-axis.

These subsets are implemented as binary trees, which in actuality are one-dimensional analogs of the MX-CIF quadtree. Thus rectangles are associated with their minimum enclosing one-dimensional x or y intervals as is appropriate. For example, Figure G illustrates the binary trees associated with the x and y axes passing through A, the root of the MX-CIF quadtree of Figure F The subdivision points of the axis lines are shown by the tick marks in Figure F

Insertion and deletion of rectangles in an MX-CIF quadtree are described on pp. 468–469 in [9] and in the solutions to the exercises on pp. 827–832 in [9]. 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.

The range and overlay operations can be implemented by using variants of algorithms developed for handling set operations (i.e., union and intersection) in region-based quadtrees [4, 10]. In particular, the range query is answered by intersecting the query rectangle with the MX-CIF quadtree. The overlay query is answered by a two-step process. First, intersect the two MX-CIF quadtrees. If the result is empty, then they can be safely overlaid and we merely need to perform a union of the two MX-CIF quadtrees. It should be clear that Boolean queries can be easily handled. An example JAVA applet for the MX-CIF quadtree data structure may be found on the web page of the class.

4 Assignment

This assignment has four parts. It is to be programmed in C++ o 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. You are strongly urged to read the the description of the MX-CIF quadtree [9].

The first part is to be turned in one week after this assignment has been distributed to you. It is worth 10 points. The second part is worth 15 points. It is to be turned in two weeks after this assignment has been distributed to you. 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. This will be done using a set of drawing programs that we will provide for which you will be provided separate documentation.

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 175 points. Part three consists of operations (1)–(9) given below. They are worth a total of 90 points, with varying point values for the different operations. Part four consists of operations (10)–(14) given below. They are worth 85 points. Operations (15)–(18) are for extra credit and are to be turned in with part four. They are worth up to 9 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 two parts 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’’`.

We will assume that rectangles do not overlap although the MX-CIF quadtree data structure can deal with such a situation. In order to lend some realism to your task you are to implement the MX-CIF 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 in our case is a $2^w \times 2^w$ array of pixels such that each pixel corresponds to a square of size 1×1. Each pixel is referenced by the x and y coordinate values of its lower-left corner. Therefore, the lower-left corner of the array (i.e., the origin) has coordinate values (0,0), and the pixel at the upper-right corner of the array has coordinate values $(2^w - 1, 2^w - 1)$. As a default, we assume $w = 7$, i.e., a pixel array of size 128×128. All rectangles are of size $i \times j$, where $3 \leq i \leq 2^w$ and $3 \leq j \leq 2^w$. In other words, the smallest rectangle is of size 3 by 3 and the largest is $2^w \times 2^w$. Note that the 1×1 pixel is the smallest unit into which our MX-CIF quadtree will decompose the underlying space from which the rectangles are drawn.

In order to simplify the project and for optional operation (17) to be meaningful (i.e., `LABEL` for connected component labeling), we stipulate that the centroids and the distances from the centroids to the borders of the rectangles are integers. Therefore, all rectangles are of size $2i \times 2j$, where $1 \leq i \leq 2^{w-1}$ and $1 \leq j \leq 2^{w-1}$. In other words, the smallest rectangle is of size 2 by 2 and the largest is $2^w \times 2^w$. As we pointed out, the rectangles are specified by the x and y coordinate values of their centroids, and the horizontal and vertical distances from the centroids to their corresponding sides. The representation is further simplified by assuming that the centroids of the rectangles are lower-left corners of pixels. In order to see this, consider the rectangle specification having $CX = 3$, $CY = 2$, $LX = 1$, and $LY = 2$ as shown in Figure H. It corresponds to a rectangle of size 2×4 with diagonally opposite corners at (2,0) and (3,3)—that is, these are the pixels for which these points serve as the lower-left corners. Thus we see that this rectangle is 2 pixels wide and 4 pixels high and has an area of 8 pixels The centroid of this rectangle is at (3,2) which is the lower-left corner of the corresponding pixel.

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.

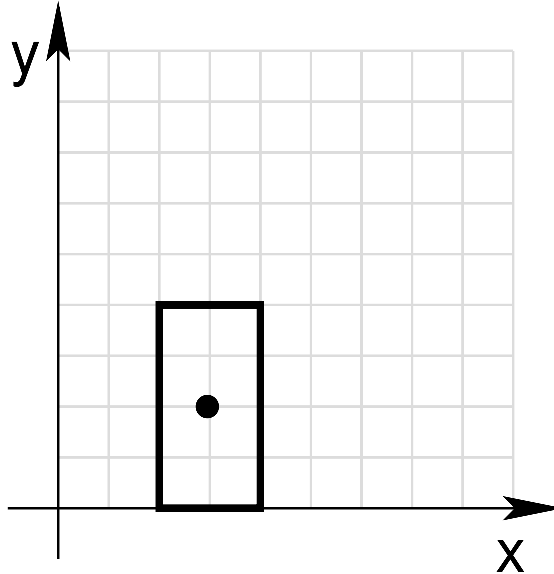


Figure H: Sample rectangle with at pixel boundaries having a centroid at (3,2) whose distance to its horizontal and vertical boundaries is 1 and 2, respectively.

4.1 Data Structure Selection

You are to select a data structure to implement the MX-CIF 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 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 centroid, and the horizontal and vertical distances from the centroid to its borders. 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 (i.e., a data structure) to efficiently keep track of the names of the rectangles (i.e., to enable their retrieval, updates, etc.). 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 MX-CIF 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. Since we also want to keep track of the nodes of the binary trees corresponding to the one-dimensional MX-CIF quadtrees that have been visited, we need to be able to assign unique numbers to them as well. The root of the binary tree is assigned the number 0. Given a node with number N, its `LEFT` and `RIGHT` children are labeled $2 \cdot N + 1$ and $2 \cdot N + 2$, respectively. For example, starting at the root, the `right` child is numbered 2, while the `LEFT` child of the `RIGHT` child of the root is numbered 2*(2*0+2)+1=5. In order to distinguish between nodes in the binary trees associated with the x and y axes of the MX-CIF quadtree, we append the character 'X' and 'Y', respectively to the number. The presence of this additional character also serves to distinguish between nodes of the quadtree and those of the binary trees.

(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 $2^{\texttt{WIDTH}}$. It also has the effect of starting with a fresh data set. This also enables you to start working with a new rectangle collection. 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 set of course slides titled "Alternative Quadtree Representations" (slide ar6 where the label is in the upper left corner) which is present in our class web page. Besides the initialization, you should also be able to output the tree representation of the current MX-CIF quadtree, which is invoked by the operation `ARCHIVE_QUADTREE()`. Most importantly, it provides you with an MX-CIF quadtree that is correct in case you are not able to build the MX-CIF quadtree correctly or to execute the most recent operation correctly. In other words, `ARCHIVE_QUADTREE` is the inverse of `BUILD_QUADTREE`.

(2) (10 points) Generate a display of a $2^{\texttt{WIDTH}} \times 2^{\texttt{WIDTH}}$ square from the MX-CIF quadtree. It is invoked by the command `DISPLAY()`. To draw the MX-CIF quadtree, you are to use the drawing routines provide. In particular, we provide you with an handout that describes their use, and the working of 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. Along with the name of a rectangle R, you should also print the node-number of the node containing R. 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) (10 points) List the names of all of the rectangles in the database in lexicographical 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 centroid, and the horizontal and vertical distances to its borders from the centroid. 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 the names of all of the rectangles in the database whether or not they have been deleted. The output should consist of several lines and each line should contain the name of just one rectangle.

(4) (5 points) Create a rectangle by specifying the coordinate values of its centroid and the distances from the centroid to its borders, and assign it a name for subsequent use. Use `CREATE_RECTANGLE(N,CX,CY,LX,LY)` where `N` is the name to be associated with the rectangle, `CX` and `CY` are the *x* and *y* coordinate values, respectively, of its centroid, and `LX` and `LY` are the horizontal and vertical distances, respectively, to its borders from the centroid. `CX`, `CY`, `LX`, and `LY` are integer numbers (although it could also be a real number in the more general case). However, as we pointed out earlier, in the case of this assignment, in order for the optional operation (17) (i.e., `LABEL` for connected component labeling) to be meaningful, recall from the introduction to Section 4 that we stipulate that the centroids and the distances from the centroids to the borders of the rectangles are integers. Output an appropriate message indicating that the rectangle has been created as well as its name and the *x* and *y* coordinate values of its centroid, and the horizontal and vertical distances to its borders from the centroid. Note that any rectangle can be created — even if it is outside the space spanned by the MX-CIF quadtree.

(5) (10 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.

(6) (10 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 MX-CIF 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 message: `‘‘N INTERSECTS RECTANGLE [NAME OF RECTANGLE]’’` for each of the intersecting rectangles. 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 MX-CIF 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 (i.e., in the database of rectangles).

(7) (5 points) Insert a rectangle in the MX-CIF 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 MX-CIF 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 MX-CIF 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 MX-CIF quadtree is built by a sequence of `CREATE_RECTANGLE` and `INSERT` operations.

Please note/recall our previously stated convention that the lower and left boundaries of each rectangle and block are closed while the upper and right boundaries of each block are open. For example, this means that when trying to insert rectangle *r*, once we have determined the minimum enclosing quadtree block *b* of *r*, we then check if the left and/or bottom sides of *r* are coincident with the top and/or right sides of *b*. If this is true, then associate *r* with the parent quadtree block *c* of *b*. In particular, *r* is associated with one of the binary trees of *c* (these binary trees are one-dimensional MX-CIF quadtrees in the parlance of the book and the assignment). Otherwise, *r* is associated with *b*.

(8) (15 points) Delete a rectangle or a set of rectangles from the MX-CIF 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 MX-CIF quadtree and not from the database of rectangles.

(9) (5 points) Move a rectangle in the MX-CIF 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 MX-CIF quadtree, or if after the operation `N` lies outside the space spanned by the MX-CIF 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) (20 points) Determine all the rectangles in the MX-CIF 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 MX-CIF 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) (20 points) Determine all of the rectangles in the MX-CIF 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 MX-CIF 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) (15 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 MX-CIF quadtree. If more than one rectangle is at the same distance, then return the name of just one of them.

(13) (15 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) (15 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 MX-CIF 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) (9 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 MX-CIF 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.

(16) (9 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 "lexicographically 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 MX-CIF 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., [3] which is described on pages 490–501 in [9]). Section 5.

(17) (9 points) Perform connected component labeling on the MX-CIF 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 MX-CIF quadtree where all touching rectangles are shown with the same label. Use integer labels.

(18) (9 points) Given a pair of MX-CIF quadtrees, find the pairs of intersecting rectangles. This is accomplished by the command `SPATIAL_JOIN`. The result of the operation is a list of all pairs of intersecting rectangles of the form (A,B) where A and B are pairs of intersecting rectangles, one from the first set and one from the second set, respectively.

5 Hints

In the following, we represent every point by its x and y coordinate values and every rectangle by a pair of points (as opposed to our intended definition of center with length and width where you would compute those 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/maximum 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

INPUT
INIT_QUADTREE(5)
LIST_RECTANGLES()
CREATE_RECTANGLE(D,8,3,6,1)
CREATE_RECTANGLE(A,24,16,2,2)
CREATE_RECTANGLE(B,26,28,4,2)
CREATE_RECTANGLE(C,11,8,1,1)
LIST_RECTANGLES()
TRACE ON
INSERT(A)
INSERT(B)
INSERT(C)
INSERT(D)
SEARCH_POINT(25,29)
SEARCH_POINT(4,4)
RECTANGLE_SEARCH(D)
DELETE_RECTANGLE(A)
DELETE_POINT(25,29)
INSERT_RECTANGLE(D)
MOVE(D,-5,1)
TRACE OFF

OUTPUT

```
INIT_QUADTREE(5): initialized a quadtree of width 32
LIST_RECTANGLES() : listing 0 rectangles
CREATE_RECTANGLE(D,8,3,6,1): created rectangle D
CREATE_RECTANGLE(A,24,16,2,2): created rectangle A
CREATE_RECTANGLE(B,26,28,4,2): created rectangle B
CREATE_RECTANGLE(C,11,8,1,1): created rectangle C
LIST_RECTANGLES(): listing 4 rectangles:
A 24 16 2 2
B 26 28 4 2
C 11 8 1 1
D 8 3 6 1
INSERT(A): inserted rectangle A
INSERT(B): inserted rectangle B
INSERT(C): inserted rectangle C
INSERT(D): failed: intersects with A
SEARCH_POINT(25,29)[0 0X 2X 0Y 2 0X 0Y 2Y]: found rectangle B
SEARCH_POINT(4,4)[0 0X 2X OY 3 0X 2X 0Y 15 0X 0Y]: no rectangle found
RECTANGLE_SEARCH(D)[0 0X 2X]: found rectangle A
DELETE_RECTANGLE(A): deleted rectangle A
DELETE_POINT(25,29): deleted rectangle B
INSERT_RECTANGLE(D): inserted rectangle D
MOVE(D,-5,1): rectangle D moved successfully
```

References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, Oct. 1983.

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

[3] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also University of Maryland Computer Science Technical Report TR–3919, July 1998.

[4] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, Apr. 1979.

[5] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982. Also University of Rochester Computer Science Technical Report TR–91, September 1981.

[6] A. Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

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

[8] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[9] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006. (Translated to Chinese ISBN 978-7-302-22784-7).

[10] M. Shneier. Calculations of geometric properties using quadtrees. *Computer Graphics and Image Processing*, 16(3):296–302, July 1981. Also University of Maryland Computer Science Technical Report TR–770, May 1979.