Programming Assignment 1:
A Data Structure For Game Programming

Abstract

In this assignment you are required to implement a system for handling data similar to that used in game programming. In such an environment the primary entities are small rectangles and the problem 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 problems. Your task is to implement this data structure in such a way that a number of operations can be handled efficiently. 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, C++, or PASCAL 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, C structs, or PASCAL record definitions. One week later you must turn in a C++, C, or PASCAL 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++, C, or PASCAL program to implement the data structure and operations (1)-(9). For the fourth part, you are to implement operations (10)-(15). Operations (16)-(18) are optional and you will get extra credit if you turn them in with part four. If you are a graduate student, part four is not optional.

Copyright © 2011 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 1.14 where a point quadtree of 8 nodes is presented. In our present discussion 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 [8–10]. 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. 26–48 and 423–426 in [11].

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 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 1.29 is the $2^3 \times 2^3$ MX quadtree corresponding to the data of Figure 1.1. 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 $1 \times 1$ square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left

---

1 All numbered figures and page numbers refer to [11].
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 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 1.14 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 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. 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 in [11].

3 MX-CIF and Loose Quadtrees

The **MX-CIF quadtree** is a quadtree-like data structure devised by Kedem [7] (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 ?? 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 3.23, 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.

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 decomposi-
tion. 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 [11].

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 [7], 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 3.24 illustrates the binary trees associated with the $x$ and $y$ axes passing through A, the root of the MX-CIF quadtree of Figure 3.23. The subdivision points of the axis lines are shown by the tick marks in Figure 3.23.

One of the main drawbacks of the MX-CIF quadtree is that the size of the block $c$ corresponding to the minimum enclosing quadtree block of object $o$’s minimum enclosing bounding box $b$ is not a function of the size of $b$ or $o$. Instead, it is dependent on the position of $o$. In fact, $c$ is often considerably larger than $b$ thereby causing inefficiency in search operations due to a reduction in the ability to prune objects from further consideration (i.e., in other words, the larger the minimum enclosing quadtree block, the greater is the number of rectangles that could be associated with it). This situation arises whenever $b$ overlaps the axes lines that pass through the center of $c$.

The cover fieldtree [3, 4], and the equivalent loose quadtree (loose octree in three dimensions) [13], overcome this drawback by expanding the size of the space that is spanned by each quadtree block $c$ of width $w$ by a block expansion factor $p \ (p > 0)$ so that the expanded block is of width $(1 + p) \cdot w$. Note that the MX-CIF quadtree is a special case of the Loose Quadtree where $p = 0$. Thus instead of associating (inserting) objects with (into) their minimum enclosing quadtree blocks, they are associated with (inserted into) their minimum enclosing expanded quadtree block.

The ideal value for $p$ is 1 [13]. The rationale is that using block expansion factors much smaller than 1 increases the likelihood that the minimum enclosing expanded quadtree block is large, and that letting $p$ be much larger than 1 results in the areas spanned by the expanded quadtree blocks being too large, thereby having much overlap. For example, letting $p = 1$, Figure 3.24 is the loose quadtree corresponding to the collection of objects in Figure 2.57(a) and its MX-CIF quadtree in Figure 2.57(b). In this example, there are only two differences between the loose and MX-CIF quadtrees:

1. Rectangle object E is associated with the SW child of the root of the loose quadtree instead of with the root of the MX-CIF quadtree.
2. Rectangle object B is associated with the NW child of the NE child of the root of the loose quadtree instead of with the NE child of the root of the MX-CIF quadtree.

Insertion and deletion of rectangles in a loose quadtree are similar to those of MX-CIF, which is described on pp. 468–469 in [11] and in the solutions to the exercises on pp. 827–832 in [11]. The most common search query is one that seeks to determine if a given rectangle overlaps (i.e., intersects) any of the existing rectangles. Such an operation is similar to a “range query”. 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 [6, 12]. The range query is answered by intersecting the query rectangle with the loose quadtree. An example JAVA applet for the loose 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++, C, or PASCAL. 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 [11].

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 also worth 10 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++, C, or PASCAL 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)-(15) given below. Each is worth 5 points for a total value of 30 points. Operations (16)-(18) 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 two parts of the assignment. For any operation that is not implemented, say 0P, your command decoder must output a message of the form ‘COMMAND 0P IS NOT IMPLEMENTED’.

In order to facilitate your program as well as lend some realism to your task you are to implement the loose 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 \( 128 \times 128 \). 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 \( 1 \times 1 \). This is the smallest unit into which our quadtrees will decompose the world. In order to simplify the project, we stipulate that the centroids and the distances from the centroids to the borders of the rectangles are integers. 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 \( 2^w \times 2^w \). One class meeting date before the due date of each part of the project you will be informed of the availability of and name of the test data file which you are to use in exercising your program for grading purposes. You should also prepare your own test data. A sample file for this purpose will also be provided.

4.1 Data Structure Selection

You are to select a data structure to implement the loose quadtree. Turn in a definition in the form of a set of C++ classes, C structs, or PASCAL record definitions. 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++, C, or PASCAL 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 \texttt{INIT\_QUADTREE}) and the actual values of the parameters if the command has any parameters (e.g., the values of \texttt{WIDTH} and \texttt{P} for the \texttt{INIT\_QUADTREE} command).

4.3 Part Three: Basic Operations

In order to facilitate grading of these operations as well as the optional operations in Section 4.5, 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 loose 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 $4 \cdot (4 \cdot 0 + 1) + 4 = 8$.

(1) (1 point) Initialize the quadtree. The command INIT_QUADTREE(WIDTH,P) 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 $2^\text{WIDTH}$. It also has the effect of starting with a fresh data set. P is the expansion factor of the loose quadtree.

(2) (1 point) Generate a display of a $2^\text{WIDTH} \times 2^\text{WIDTH}$ square from the loose quadtree. It is invoked by the command DISPLAY(). To draw the loose quadtree, you are to use the drawing routines that we provide. In particular, we provide you with an handout that describes their use, and the working of utilities SHOWQUAD and PRINTQUAD, that you will use 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) (3 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 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 centroid and the distances from the centroid to its borders, and assign it a name for subsequent use. It is invoked by the command 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 $LY$ may be real or integer numbers. However, in the case of this assignment, 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 endpoints. Note that any rectangle can be created — even if it is outside the space spanned by the loose quadtree.

(5) (5 points) Insert a rectangle in the loose quadtree. If any part of the rectangle is outside the space spanned by the loose 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 LOOSE 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 loose quadtree is built by a sequence of CREATE_RECTANGLE and INSERT operations.

(6) (4 points) Move a rectangle in the loose 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. 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 loose quadtree, or if after the operation $N$ lies outside the space spanned by the loose quadtree.

(7) (4 points) Given a point, return the names of the rectangles that contain it. The names of the rectangles are returned in a lexicographical order. 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 loose 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 (could be more than one rectangle) to be deleted whose $x$ and $y$ coordinate values are given by PX and PY, respectively. DELETE_POINT returns as its value the names of the rectangles that have been deleted and prints an appropriate message indicating their names. 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 for both variants of the operation only delete the rectangles from the loose quadtree and not from the database of rectangles.

(9) (5 points) Determine whether a query rectangle intersects (i.e., overlaps) any of the existing rectangles. This operation is invoked by the command REGION_SEARCH($N$) where $N$ is a name of a rectangle. If the rectangle does not intersect an existing rectangle, then REGION_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 the names of the intersecting rectangles (i.e., if it intersects more than one rectangle) to output one of the following two messages: ‘‘$N$ INTERSECTS RECTANGLE [NAMES OF RECTANGLES]’’. The names of the intersecting rectangles in the output are sorted in a lexicographical order. Note that if an endpoint of the query rectangle touches the endpoint of an existing rectangle, then REGION_SEARCH returns false. You are only to check against the rectangles that are in the loose 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).

4.4 Part Four: Advanced Operations

(10) (5 points) Determine all the rectangles in the loose 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 loose 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) (5 points) Determine all of the rectangles in the loose 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
\( \text{WITHIN}(N, D) \) 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 loose 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) (5 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 \( \text{HORIZ}_-\text{NEIGHBOR}(N) \) where \( N \) is the name of the query rectangle. \( \text{VERT}_-\text{NEIGHBOR}(N) \) 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 \( \text{NIL} \) otherwise as well as an appropriate message. Rectangle \( N \) need not necessarily be in the loose 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) (5 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 \( \text{NEAREST}_-\text{RECTANGLE}(PX, PY) \) 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) (5 points) Find all rectangles in a rectangular window anchored at a given point. It is invoked by the command \( \text{WINDOW}(LLX, LLY, LX, LY) \) 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 loose 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 \( \text{WINDOW} \) 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.**

(15) (5 points) Change the expansion factor of the loose quadtree. This operation is invoked by the command \( \text{CHANGE}_-\text{EXPANSION}_-\text{FACTOR}(P) \), where \( P \) is the new value of the expansion factor. **You will NOT be given credit for a solution that recreates a new loose quadtree by reinserting all the rectangles.**
4.5 Optional Operations

(16) (4 points) Find the nearest neighbor in all directions to the boundary of a given rectangle. It is invoked by the command \texttt{NEAREST\_NEIGHBOR(N)} where \texttt{N} is the name of a rectangle. By “nearest,” it is meant the rectangle \texttt{C} with a point on its side or corner, say \texttt{P}, such that the distance from \texttt{P} to a side or corner of the query rectangle is a minimum. \texttt{NEAREST\_NEIGHBOR} returns as its value the name of the neighboring rectangle if one exists and \texttt{NIL} otherwise as well as an appropriate message. Rectangle \texttt{N} need not necessarily be in the loose 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 \texttt{N} are not considered by this query.

(17) (4 points) Given a rectangle, find its nearest neighbor with a name that is lexicographically greater. It is invoked by the command \texttt{LEXICALLY\_GREATER\_NEAREST\_NEIGHBOR(N)} where \texttt{N} is the name of a rectangle. By “lexicographically greater nearest” it is meant the rectangle \texttt{C} whose name is lexicographically greater than that of \texttt{N} with a point on \texttt{C}’s side, say \texttt{P}, such that the distance from \texttt{P} to a side of the query rectangle is a minimum. \texttt{LEXICALLY\_GREATER\_NEAREST\_NEIGHBOR} returns as its value the name of the neighboring rectangle if one exists and \texttt{NIL} otherwise as well as an appropriate message. Rectangle \texttt{N} need not necessarily be in the loose 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 \texttt{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 lexically greater nearest neighbor. Thus you should use an incremental nearest neighbor algorithm (e.g., [5] which is described on pages 490–501 in [11]).

(18) (4 points) Perform simple ray tracing operation on the loose quadtree. To keep things simple, let us reduce this problem to that of finding rectangles in the loose quadtree that intersect a line segment. In particular, we are only interested in finding the “first” rectangle that the “ray” intersects, assuming the ray enters the scene from \((A, 0)\) and leaves the scene from \((B, 2^w-1)\), where \(A\) and \(B\) are positive numbers. This operation is invoked by the command \texttt{RAYTRACE(A, B)}. Output the first rectangle that intersects the ray. If no such rectangle exists, then output a suitable message.

References


