CS 300
Data Structures
Homework 2

Assigned: Nov 3, 2017 Due: November 14, 2017 at 11:55pm

PLEASE NOTE:

- SOLUTIONS HAVE TO BE YOUR OWN.
- NO COLLABORATION OR COOPERATION AMONG STUDENTS IS PERMITTED.
- 10% PENALTY WILL BE INCURRED FOR EACH DAY OF OVERTIME. SUBMISSIONS THAT ARE LATE MORE THAN 3 DAYS WILL NOT GET ANY CREDITS.
- SUBMISSIONS WILL BE MADE TO THE SUCourse SERVER. NO OTHER METHOD OF SUBMISSION WILL BE ACCEPTED.

1 INTRODUCTION

In this assignment, you will implement a slightly more complex version of binary search trees. Remember that binary search trees typically work on *one dimensional key spaces*. The trees that you will work on in this homework let us search on *two-dimensional spaces*, and extensions to higher dimensional spaces should be obvious. These kind of trees are very useful in many graphics applications and in computer aided design tools for automated design of VLSI (very large scale integration) circuits. Instead of each node having at most 2 children, in such trees, nodes have at most 4 children. For reasons that will be clear in a moment, we will call these children as `TopLeft`, `TopRight`, `BottomLeft` and `BottomRight`. Before we go on any further, let us briefly discuss how you will use these trees.

You are given a (possibly very large) database of rectangles. You will represent each of these rectangles by objects which look like

```
class Rectangle {
   public:
      ....
   private:
      int Top; // y coordinate of the upper edge
      int Left; // x coordinate of the left edge
      int Bottom; // y coordinate of the bottom edge
      int Right; // x coordinate of the right edge
      ....
    };
```

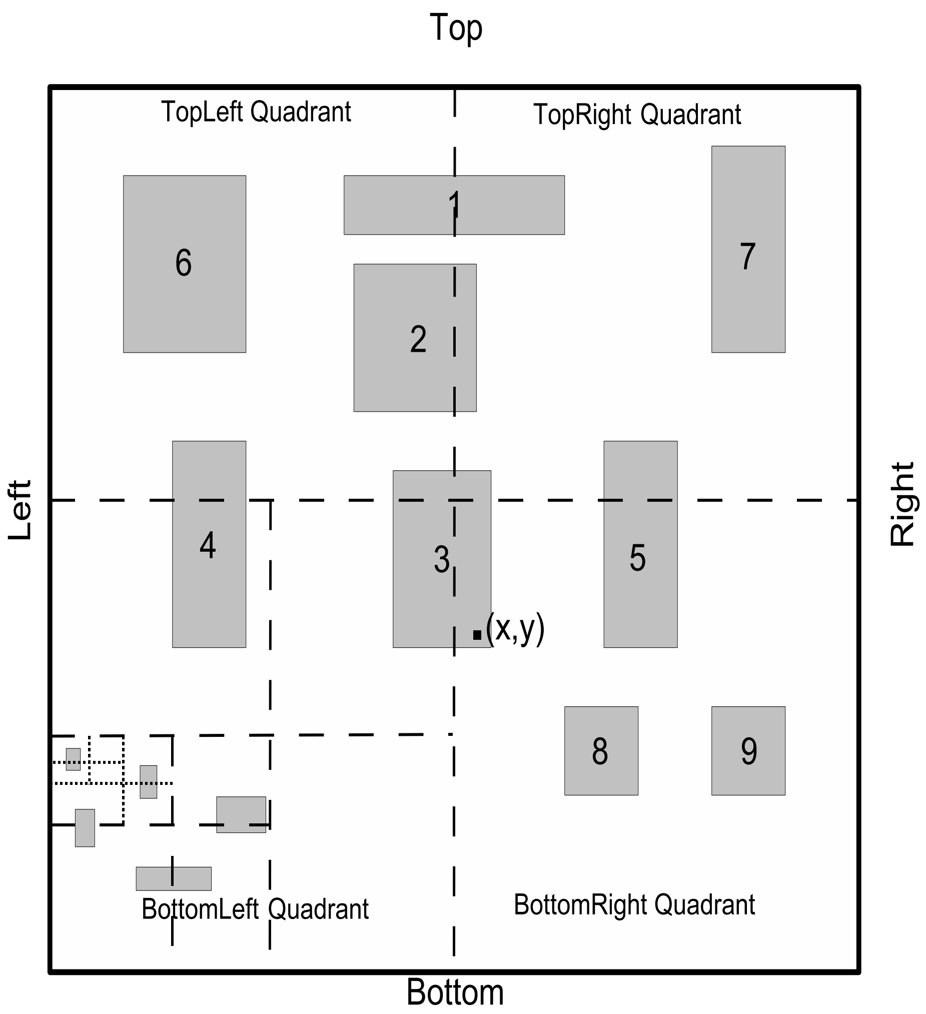

Figure 1: How a two-dimensional tree stores rectangles

Contrary to what you are used to in Cartesian geometry, the coordinate system in this representation is such that, *as you go towards right and bottom, the x and y coordinates (respectively) increase*, that is, for a rectangle, `Bottom` $>$ `Top` and `Right` $>$ `Left`. A list of such rectangles can be trivially represented using the `LinkedList` class you studied earlier.[1]

A point (x, y), on the Cartesian plane is considered to be inside a rectangle object `R` (or intersects `R`), if `R.Left` $\leq x <$ `R.Right` and `R.Top` $\leq y <$ `R.Bottom`, *that is, the points on the right and bottom boundaries of the rectangle are not considered to be in the rectangle.* Given such a point, our tree data structure will enable us to find (hopefully in a very fast way) all rectangles that contain that point. One can obviously do this by keeping all rectangles in a linked list and searching through the list by checking if the point is inside a rectangle or not, but when you have 1,000,000 or 50,000,000 rectangles[2], that will not be very efficient.

The trees that you will design organize two-dimensional information in the following way:

- You will assume that each such tree covers a portion of the Cartesian plane and that this portion is fixed. You will thus assume that the region covered by the tree is represented by some suitable (large) rectangle. Call this rectangle `Extent`.
- The center of the tree rectangle is located at the center point whose x coordinate is

  – (`Extent.Left` + `Extent.Right`)/2

  and whose y coordinate is

  – (`Extent.Top` + `Extent.Bottom`)/2

  where the / is interpreted as the integer division operator in C++.
- Note that, this center defines four additional smaller rectangles (called quadrants) at the top left, top right, bottom left and bottom right of the center point. So, if you are given a point (x, y) you only need to find which quadrant the point falls, and then just search the rectangles that you know (may) overlap with that quadrant, not all rectangles. Note also that this can be extended recursively to smaller quadrants within a quadrant until a minimum rectangle size is reached. See Figure 1 for some of the details of how a two-dimensional tree stores rectangles.

The class definition for *a node* of such a two-dimensional tree (the equivalent of the `BinaryTreeNode` class) will look like the following:

[1]The source code for the linked list and related objects are available from the course web page.
[2]For instance, when you are doing a VLSI circuit design

```
class TwoDimTreeNode {
      public:
      ....
      private:
         Rectangle Extent;
         LinkedList<Rectangle> Vertical, Horizontal;
         TwoDimTreeNode *TopLeft, *TopRight,
                         *BottomLeft, *BottomRight;
};
```

The two dimensional tree class will just contain a single private entry that points to a root node of the class `TwoDimTreeNode` just like the binary search tree definition we discussed in the class. The rectangle `Extent` defines the region covered by the tree. `Vertical` and `Horizontal` are two linked lists of rectangles. `Vertical` keeps the list of rectangles which intersect the vertical line $x = (\texttt{Extent.Left} + \texttt{Extent.Right})/2$ and `Horizontal` keeps the list of rectangles which intersect the horizontal line $y = (\texttt{Extent.Top} + \texttt{Extent.Bottom})/2$. If a rectangle intersects both lines, then it is put on only one of these lists and **not two**. So in Figure 1, rectangles 1,2 and 3 would be on the `Vertical` list for the root extent rectangle and rectangles 4 and 5 would be on the `Horizontal` list for the root extent rectangle.

If a rectangle does not intersect any of these lines then, it is either in the tree pointed by `TopLeft` which covers the rectangular extent with[3]

$\texttt{Top} = \texttt{Extent.Top}$

$\texttt{Left} = \texttt{Extent.Left}$

$\texttt{Bottom} = (\texttt{Extent.Top} + \texttt{Extent.Bottom})/2$

$\texttt{Right} = (\texttt{Extent.Left} + \texttt{Extent.Right})/2$

or in one of the other 3 regions (pointed to by `TopRight`, `BottomLeft`, `BottomRight`) whose extents can be defined similarly. Thus, it is recursively inserted to the subtree of the relevant quadrant unless the width or the length of the extent is 1 so that it can not be subdivided any further. The important point to note is that a given rectangle is inserted to either the `Horizontal` or the `Vertical` lists associated with the subtree whose center lines it intersects. *Note also that the extents of the children subtrees do not intersect either of the center lines.* An example can be of help here. Suppose the tree covers the extent

`Top = 0`, `Left = 0`, `Bottom = 4` and `Right = 5.`

Then,

- the `TopLeft` quadrant of this extent is defined by `Top = 0`, `Left = 0`, `Bottom = 2` and `Right = 2.`

[3]Again remember that the right and bottom boundaries are not part of the rectangles.

- the `TopRight` quadrant of this extent is defined by `Top = 0`, `Left = 3`, `Bottom = 2` and `Right = 5`.
- the `BottomLeft` quadrant of this extent is defined by `Top = 3`, `Left = 0`, `Bottom = 4` and `Right = 2`.
- and the `BottomRight` quadrant of this extent is defined by

  `Top = 3`, `Left = 3`, `Bottom = 4` and `Right = 5`.

Also note that areas of the quadrants need not be equal but they are close.

To search for the rectangles that a point (x, y) intersects with, you first checks if the point falls in any of rectangles on the `Vertical` and `Horizontal` lists of the root node. If so, these rectangles are output (or inserted to a result list of rectangles) and search continues recursively with the subtree covering the quadrant the point falls into. (*No additional search is needed if the point falls on the center lines of the tree.*) For instance looking at Figure 1, for the given (x, y) point one would search the `Vertical` and `Horizontal` lists of the main root rectange and then only the subtree corresponding to the bottomright quadrant. None of the other 3 quadrants need to be searched.

2 WORK TO BE DONE

As a part of this homework, you should implement the necessary classes (some of which were sketched out above). Obviously you can reuse any other classes available (such as the `LinkedList` class).

You should then write a simple application program that uses these classes.

1. In order to simplify certain matters, you will assume that the data for the rectangle database will be in a file called **rectdb.txt**. So, your program reads from a file called **rectdb.txt** four integers (in the order `Top`, `Left`, `Bottom`, `Right`) that define the extent rectangle for the top main tree (you can assume that all the coordinates of all the rectangles are nonnegative integers, and no rectangle has any portion that falls outside the extent of the whole tree).
2. It then reads again from the same file, the `Top` value for a rectangle, if this value is -1 (or any negative number) then the input of rectangles is over. The program proceeds to step 3. Otherwise the `Left`, `Bottom`, `Right` values are read and inserted the rectangle to the right place in your two-dimensional tree. You then repeat this step.
3. Your program then reads from the standard input, query points given as pairs of integers (in the order x, y) and then prints out to standard out, for each query point,

   (a) the query point itself,
   (b) the number of rectangles found and then
   (c) the coordinates of the intersecting rectangles (with the order top, left, bottom, right) one rectangle on a line.

4. If the program reads a query point with x = -1 the program exits.

For example, a typical database file **recdb.txt** will be like:[4]

```
0 0 1000 1000 (extent rectangle of the whole tree)
0 0 100 100 (first rectangle)
650 700 750 800 (second rectangle) ....
.... -1  (end of rectangles)
```

The query data that will come from the standard input will look like:

```
3 4 (first query point)
700 800 (second query point)
...
-1 70 (last query point)
```

The output would be like

```
3 4  (the query point)
2 (assume there are two rectangles)
0 0 100 100 (the first intersecting rectangle)
2 1 10 12 (the second intersecting rectangle)
700 800 (the query point)
0  (so this point intersects with no rectangles)
..
..
```

[4]The text in the parentheses are provided as explanations– they are not part of the file nor part of the input or output.

Your code should be submitted to SUCourse at the deadline given on the first page. You should follow the following steps:

- Name the folder containing your source files as *XXXX-NameLastname* where XXXX is your student number. Make sure you do NOT use any Turkish characters in the folder name. You should remove any folders containing executables (Debug or Release), since they take up too much space.
- Use the Winzip program to compress your folders to a compressed file named, for example, *5432-AliMehmetoglu.zip*. After you compress, please make sure it uncompresses properly and reproduces your folder exactly.
- You then submit this compressed file in accordance with the deadlines above.

Your homework will be graded in the following way:

- If your program does not use the classes as we outlined above (for example you come up with a solution that just uses linked lists as the database and does not use two-dimesional trees), you will get 0 points, and that is it. This will be the case even if your program works correctly otherwise.
- We will run 9 tests on your homework with 3 different rectangle databases and check the results. Each correct test will earn you 10 points. Note that your outputs should be in the exact same format we described above.
- The style of your OWN code will be graded on clarity, commenting, etc. This will cover 10 points. You will however NOT get this credit if your program fails in ALL the tests we perform. A nice looking but useless program is just a useless program no matter how nice it is.

Good luck