CS 202 – Data Structures (Spring 2007)

Homework Assignment 3: Quadtrees

Assigned on 4 April, 2007 — Due by 23:00:00 on Thursday 3 May, 2007
Graded by Billur Engin (billurengin@...)

PLEASE NOTE:

- The deadline is hard: No exceptions will be made.
- Solutions must be your own: No cooperation is permitted.
- Late solutions will be penalised by 10 points for each day of delay, but solutions that are late by more than 2 days will get 0 points.
- Solutions must be submitted via the WebCT server (whose clock may differ from yours): No other method of submission will be accepted.

1 Quadtrees

Quadtrees are a slightly more complex version of binary search trees. While binary search trees typically work on *one-dimensional* key spaces, quadtrees let us search on *two-dimensional* key spaces, and extensions to higher-dimensional spaces are obvious. These kinds of trees are very useful in many graphics applications and in computer-aided design tools for the automated design of VLSI (very large-scale integration) circuits. Instead of each node having at most two children, quadtree nodes have at most four children. Let us first briefly discuss how we will use these trees. We are given a possibly very large database of rectangles, represented as follows:

```
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
      ...
    };
```

Contrary to convention in Cartesian geometry, the coordinate system in this representation is such that *as one goes toward the right and bottom, the x and y coordinates respectively increase*, that is, for a rectangle, `Top` $<$ `Bottom` and `Left` $<$ `Right`. A list of such rectangles can be represented using the `LinkedList` class.[1]

[1] The source code for linked lists and related classes is available from the course webpage.

A point (x, y) on this plane *is inside* a rectangle `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 a rectangle are not considered to be inside the rectangle.* A quadtree enables us to find very quickly all rectangles that contain a given point. One can obviously also do this by keeping all the rectangles in a linked list and searching through the list by checking if the point is inside one of its rectangles or not, but when there are millions of rectangles,[2] this will not be very efficient.

Quadtrees can organise such two-dimensional information in the following way:

- Assume that each quadtree covers a region of the plane and that this region is fixed. Assume thus that the region covered by the tree is itself represented by some suitable (possibly large) rectangle. Call this rectangle `Extent`.
- The center of the quadtree rectangle is located at the center point whose x coordinate is

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

  and whose y coordinate is

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

  where the / is interpreted as the integer division operator in C++.
- Note that this center point defines four additional smaller rectangles, called *quadrants*, namely at its top left, top right, bottom left, and bottom right. So, for a given point (x, y), one only needs to find into which quadrant it falls, and then only search the rectangles that are known to overlap with that quadrant, but not all the 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 how a quadtree stores rectangles.

2 Representing Rectangle Collections as Quadtrees

The `TwoDimTree` class contains a single private attribute that points to a root node of the `TwoDimTreeNode` class, whose definition has the following structure:

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

[2]For instance, this is the case when designing a VLSI circuit.

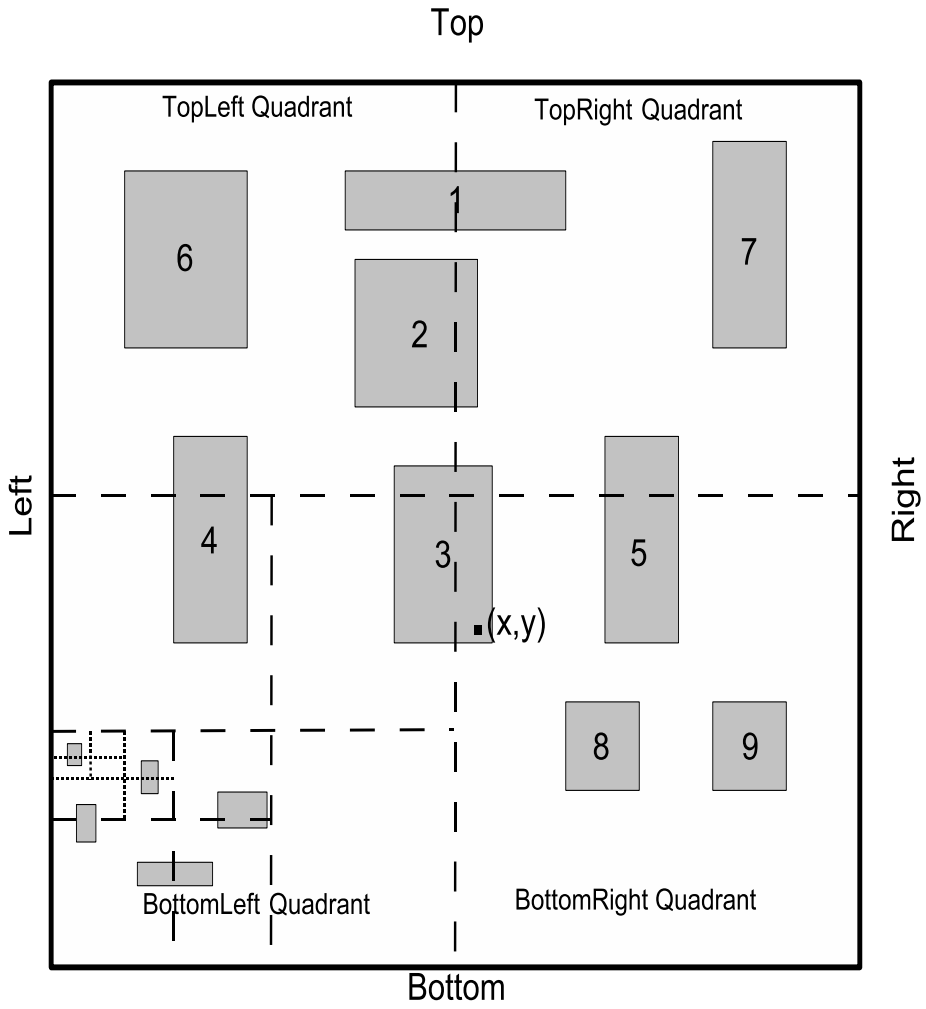

Figure 1: Storage of rectangles in a quadtree

The rectangle `Extent` defines the region covered by the tree. `Vertical` and `Horizontal` are two linked lists of rectangles, such that `Vertical` keeps the list of rectangles that intersect the vertical line $x = (\texttt{Extent.Left} + \texttt{Extent.Right})/2$ and `Horizontal` keeps the list of rectangles that intersect the horizontal line $y = (\texttt{Extent.Top} + \texttt{Extent.Bottom})/2$. If a rectangle intersects both lines, then it is inserted into only one of these lists, but **not into both**. So, in Figure 1, rectangles 1 to 3 are on the `Vertical` list for the root extent rectangle, while rectangles 4 and 5 are on the `Horizontal` list for the root extent rectangle.

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

$$\begin{aligned} \texttt{Top} &= \texttt{Extent.Top} \\ \texttt{Left} &= \texttt{Extent.Left} \\ \texttt{Bottom} &= (\texttt{Extent.Top} + \texttt{Extent.Bottom})/2 \\ \texttt{Right} &= (\texttt{Extent.Left} + \texttt{Extent.Right})/2 \end{aligned}$$

[3]Recall that the right and bottom boundaries are *not* part of the rectangles.

or in one of the other three regions, pointed to by `TopRight`, `BottomLeft`, and `BottomRight`, whose extents are defined similarly. Thus, it is recursively inserted into the subtree of the relevant quadrant, unless the width or the height of the extent is 1 so that it cannot be subdivided any further. A given rectangle is thus inserted into either the `Horizontal` list or the `Vertical` list associated with the subtree whose center line it intersects. *The extents of the subtrees do not intersect either of the two center lines* and the areas of these quadrants need not be equal.

Example 1 If a tree covers the extent `Top = 0`, `Left = 0`, `Bottom = 4`, and `Right = 5`, then:

- The `TopLeft` quadrant of this extent has `Top=0`, `Left=0`, `Bottom=2`, and `Right=2`.
- The `TopRight` quadrant of this extent has `Top=0`, `Left=3`, `Bottom=2`, and `Right=5`.
- The `BottomLeft` quadrant of this extent has `Top=3`, `Left=0`, `Bottom=4`, and `Right=2`.
- The `BottomRight` quadrant of this extent has `Top=3`, `Left=3`, `Bottom=4`, and `Right=5`.

Now, to search for the rectangles that a point (x, y) intersects, check if it falls into any of the rectangles on the `Vertical` and `Horizontal` lists of the root node. If so, output these rectangles (or insert them into a result list) and continue search recursively in the subtree covering the quadrant the point falls into. *No additional search is needed if the point falls on a center line of the root extent.* For instance, for the given (x, y) point in Figure 1, one searches in the `Vertical` and `Horizontal` lists of the root extent and then only in the subtree corresponding to the bottom-right quadrant; none of the other three quadrants needs to be searched.

3 Work To Be Done

Implement the necessary classes, some of which were sketched out above. You may reuse any other available classes from the course slides/textbook, such as the `LinkedList` class.

Write a simple application program that uses these classes:

1. Read the data of the rectangle database from a file called `rectdb.txt`, starting with four space-separated integers (in the order `Top`, `Left`, `Bottom`, `Right`) on one row, defining the extent rectangle of the root node.
2. Read from the same file the `Top` value for the next rectangle. If this value is negative, then the input of rectangles is over and proceed to step 3. Otherwise, the `Left`, `Bottom`, and `Right` values for that next rectangle come space-separated on the same row and that rectangle is inserted into the tree. Repeat this step.
3. Read, from standard input, query points given as pairs of integers (first x, then y). If a query point has $x < 0$, then exit. Wrote, to standard output, lines with the following content for each query point:
   (a) The space-separated coordinates of the query point itself.

(b) The number of found rectangles that contain it.

(c) The space-separated coordinates of the intersecting rectangles (in the order `Top`, `Left`, `Bottom`, `Right`), one intersecting rectangle per line.

Assume that all the coordinates of all the rectangles are non-negative integers, and that no rectangle has any portion that falls outside the extent of the tree.

Example 2 A typical database file `rectdb.txt` looks like:[4]

```
0 0 1000 1000    (extent rectangle of the whole tree)
0 0 100 100      (first  rectangle)
650 700 600 400  (second rectangle)
...
-1               (end of rectangles)
```

If the query data from standard input look like:

```
3 4              (first  query point)
700 800          (second query point)
...
-1 70            (end of query points)
```

then the outputs look like:

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

[4]The texts in the parentheses are explanations, but *neither* part of the file *nor* part of the input or output.

What and How to Submit When?

Your solution must be in a single folder. Take our warning on plagiarism **very seriously**. We assume that by submitting your solution, you are certifying that you are submitting your own work. Take the following steps:

1. Name the folder containing your files as *LastnameName-StudentID-hw3*. Do not use any special Turkish characters in the folder name. Remove any executables (debug or release), as they take up too much space.

2. Compress your folder, giving *MehmetogluAli-15432-hw3.zip* for example. Make sure it uncompresses properly, reproducing your folder exactly, and actually corresponds to assignment 3.

3. Submit this compressed file via WebCT by the deadline given on the first page.

Your solution will be graded in the following way:

- If your program does not use the classes as outlined above, then you will get 0 points, even if your program works correctly otherwise.

- If the program does not compile, then you get 0 points, otherwise you get 30 points.

- We will run n tests on your homework. Some of these check boundary conditions. Your outputs must be *exactly* in the format described above, without a single additional character: otherwise your program is considered to function incorrectly. Each fully correct test result will earn you $50/n$ points.

- The style of your own code will be graded on clarity, comments, etc. This will cover 20 points. You will however not get any of these points 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.

For late submissions, we will first grade your solution as indicated above and then discount accordingly. So, assuming you got 90 points, if you were late at least one second but at most one day, then your actual grade will be 80 points; if you were late more than one day but at most two days, then your actual grade will be 70 points; if you were late more than two days, then your actual grade will be 0 points and your solution will not be graded.

Have fun!