AD1 – Algorithms and Data Structures I (course 1DL210)

Assignment 3: Quadtrees

Due by 23:59:59 on Friday 9 May, 2008

Rectangles and Quadtrees

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 computer-aided design tools, say for the design of VLSI (very large-scale integration) circuits. Instead of 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 collection of rectangles, represented as follows:

```
type rectangle = {
  top    : int,  (* y coordinate of the upper  edge *)
  left   : int,  (* x coordinate of the left   edge *)
  bottom : int,  (* y coordinate of the bottom edge *)
  right  : int   (* 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 *increase*, that is, for any rectangle `r`, we have $\texttt{r.top} < \texttt{r.bottom}$ and $\texttt{r.left} < \texttt{r.right}$. Note that we thus do not consider degenerate rectangles, such as points or line segments.

A point $(\texttt{x}, \texttt{y})$ on this plane is said to be *inside* a rectangle `r` if $\texttt{r.left} \leq \texttt{x} < \texttt{r.right}$ and $\texttt{r.top} \leq \texttt{y} < \texttt{r.bottom}$, that is the points on the right and bottom boundaries of a rectangle are *not* inside it.

A quadtree enables us to find very quickly all rectangles inside which a given point is. One can obviously also do this by keeping all the rectangles in a list, but when there are millions of rectangles (for instance, when designing a VLSI circuit), this will be very inefficient.

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

- Assume that a quadtree covers a fixed rectangular region of the plane, itself represented by a rectangle, called the *extent*.
- The centre point of the quadtree extent has as x coordinate

$$(\texttt{extent.left} + \texttt{extent.right})\ /\ 2$$

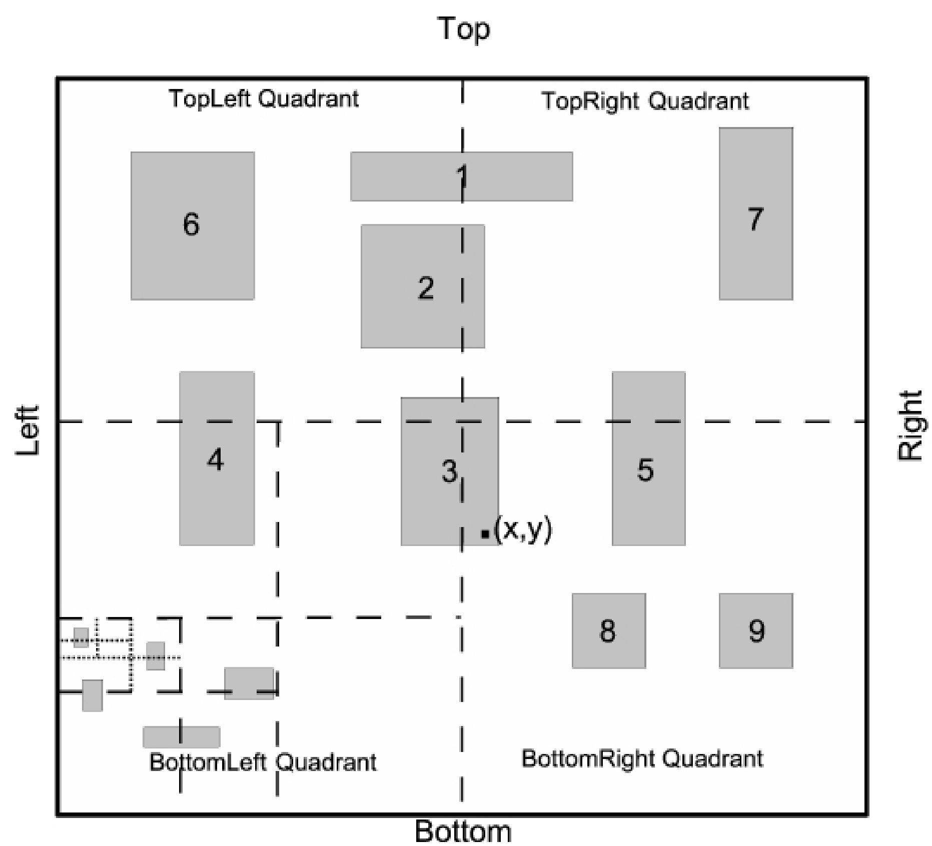

Figure 1: Storage of rectangles in a quadtree

and as y coordinate

$$(\texttt{extent.top} + \texttt{extent.bottom}) \ / \ 2$$

where / represents integer division.

- This centre point defines four smaller rectangles, called *quadrants*, at its top left, top right, bottom left, and bottom right. This can be extended recursively to smaller quadrants within a quadrant, until a minimum rectangle size is reached. See Figure 1 for an example of how a quadtree stores rectangles.

Representing Rectangle Collections as Quadtrees

The `quadTree` datatype has the following definition:

```
datatype quadTree = EmptyQuadTree | Qt of {
  extent      : rectangle,
  vertical    : rectangle list,
  horizontal  : rectangle list,
  topLeft     : quadTree,
  topRight    : quadTree,
  bottomLeft  : quadTree,
  bottomRight : quadTree
}
```

The `extent` rectangle defines the region covered by the quadtree, while `vertical` is the list of rectangles inside which some point of the vertical centre line $x = (\texttt{extent.left} + \texttt{extent.right})\ /\ 2$ is, and `horizontal` is the list of rectangles inside which some point of the horizontal centre line $y = (\texttt{extent.top} + \texttt{extent.bottom})\ /\ 2$ is. If both centre lines have some point inside a given rectangle, then it is inserted only into the `vertical` list. For example, in Figure 1, rectangles 1 to 3 are on the `vertical` list for the root extent, while rectangles 4 and 5 are on its `horizontal` list.

If none of the two centre lines has a point inside a given rectangle, then it is inserted either into the `topLeft` subtree, which covers the extent with

$$\begin{aligned} \texttt{top} &= \texttt{extent.top} \\ \texttt{left} &= \texttt{extent.left} \\ \texttt{bottom} &= (\texttt{extent.top} + \texttt{extent.bottom})\ /\ 2 \\ \texttt{right} &= (\texttt{extent.left} + \texttt{extent.right})\ /\ 2 \end{aligned}$$

or into one of the other three subtrees, called `topRight`, `bottomLeft`, and `bottomRight`, whose extents are defined similarly. Note that the areas of these quadrants need not be the same. Also note that *none* of the two centre lines has a point inside any of the quadrants, because the points on their right and bottom boundaries are *not* inside these quadrants.

Example 1 If the extent has `top=0`, `left=0`, `bottom=4`, and `right=5`, then:

- The `topLeft` quadrant has `top=0`, `left=0`, `bottom=2`, and `right=2`.
- The `topRight` quadrant has `top=0`, `left=3`, `bottom=2`, and `right=5`.
- The `bottomLeft` quadrant has `top=3`, `left=0`, `bottom=4`, and `right=2`.
- The `bottomRight` quadrant has `top=3`, `left=3`, `bottom=4`, and `right=5`.

A given rectangle is thus recursively inserted into either the `vertical` list or the `horizontal` list associated with the subtree of the quadrant whose centre lines have a point inside it.

To search for the rectangles inside which a given point (x, y) is, first collect the rectangles on the `vertical` and `horizontal` lists of the root node inside which (x, y) is. Then continue search recursively in the subtree covering the quadrant, if any, inside which the point is; *no* additional search is needed if (x, y) is on a centre line of the root extent. For example, for the given point (x, y) in Figure 1, one searches in the `vertical` and `horizontal` lists of the root extent and then only in the subtree covering the bottom-right quadrant.

Work To Be Done

Implement the following functions:

- `insert(q,r)` returns the quadtree `q` with rectangle `r` inserted;
- `query(q,x,y)` returns the list of rectangles of quadtree `q` inside which the point `(x,y)` is, where `x` and `y` are integers.

Give, in comments within the program, your explicit reasoning establishing the average-case and worst-case runtime complexities of these functions.

Grading

Your solution is graded in the following way:

- If your program was submitted before the deadline turns hard, loads under Moscow ML version 2.01, and is a serious attempt at implementing and commenting (under at least the coding convention) all the requested functions, then you get 30 points (before any penalty deductions for being late compared to the soft deadline); otherwise, you get 0 points.
- Your program is run on t orthogonal tests, checking also boundary conditions, but no error conditions. Each test is a sequence of rectangle insertions into the initially empty quadtree, followed by a sequence of queries inside which rectangles of the resulting quadtree a given point is. For each fully correct test result, you get $50/t$ points. We reserve the right to run these tests automatically, so be careful with names and argument orders.
- Your program is graded on style and comments (including specifications, representation conventions and invariants, and recursion variants), provided it does not fail on all the tests we perform. This covers 10 points.
- Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers 10 points.

Have fun!