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 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 of the following type:

```
type rectangle = int * int * int * int
```

where a rectangle (left, top, right, bottom) has left as x coordinate of the left edge, top as y coordinate of the upper edge, right as x coordinate of the right edge, and bottom as y coordinate of the bottom 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 we assume top < bottom and left < right for any rectangle. Note also that we thus do not consider degenerate rectangles, such as points or line segments.

A point with integer coordinates (x, y) on this plane is said to be inside a rectangle (left, top, right, bottom) if and only if left ≤ x < right and top ≤ y < bottom. Note that the points on the right and bottom boundaries of a rectangle are not inside it. We also say that a rectangle contains any point inside it.

A quadtree allows us to represent a collection of rectangles such that one can efficiently search for all rectangles containing a given point. 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 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, say \((\text{left}, \text{top}, \text{right}, \text{bottom})\), has integer coordinates \(((\text{left} + \text{right}) / 2, (\text{top} + \text{bottom}) / 2)\), where the symbol / 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 some termination criterion, such as minimum rectangle size. See Figure 1 for an example of how a quadtree stores rectangles.

### Representing Rectangle Collections as Quadtrees

The `quadTree` datatype has the following definition:

```haskell
datatype quadTree = EmptyQuadTree |
Qt of rectangle * rectangle list * rectangle list * quadTree * quadTree * quadTree * quadTree
```
In a non-empty quadtree $Q_t$ (extent, vertical, horizontal, topLeft, topRight, bottomLeft, bottomRight), the extent rectangle, say $\{ \text{left}, \text{top}, \text{right}, \text{bottom} \}$, defines the region covered by the quadtree, while vertical is the list of rectangles containing some point of the vertical centre line $x = (\text{left} + \text{right}) / 2$, and horizontal is the list of rectangles containing some point of the horizontal centre line $y = (\text{top} + \text{bottom}) / 2$.

If both centre lines have some point inside a given rectangle, then this rectangle is inserted only into the horizontal list. For example, in Figure 1, rectangles 1 and 2 are on the vertical list for the root extent, while rectangles 3 to 5 are on its horizontal list.

If none of the two centre lines has a point inside a given rectangle, then this rectangle is inserted either into the topLeft subtree, which covers the extent $(\text{left}, \text{top}, (\text{left} + \text{right}) / 2, (\text{top} + \text{bottom}) / 2)$, or into one of the other three subtrees, called topRight, bottomLeft, and bottomRight, whose extents are defined similarly, such that none of the two centre lines has a point inside any of the quadrants. The areas of these quadrants need thus not be the same.

**Example 1** For the extent $(0, 0, 5, 4)$, the topLeft, topRight, bottomLeft, and bottomRight extents are $(0, 0, 2, 2)$, $(3, 0, 5, 2)$, $(0, 3, 2, 4)$, and $(3, 3, 5, 4)$, respectively. (Recall that the $y$ coordinates increase when moving downward.)

A given rectangle is thus recursively inserted into either the vertical list or the horizontal list associated with the subtree of the quadrant whose vertical respectively horizontal centre line has a point inside that rectangle.

To search for the rectangles containing a given point $(x, y)$, first collect the rectangles on the vertical and horizontal lists of the root node containing $(x, y)$. Then continue search recursively in the subtree covering the quadrant, if any, containing the point; no additional search is thus needed if $(x, y)$ is on a centre line of the extent. For example, for the marked 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, making sure that they pass the training test cases (at http://www.it.uu.se/edu/course/homepage/pkd/ht10/labs/inlupp3-training.sml):

- **emptyQtree** $e$ non-recursively returns the empty quadtree with extent $e$;
- **insert** $(q, r)$ returns the quadtree $q$ with rectangle $r$ inserted, assuming all points of $r$ are within the extent of $q$.
- **query** $(q, x, y)$ returns the list (in any order) of rectangles of quadtree $q$ containing the point $(x, y)$, where $x$ and $y$ are integers.

In a separate report in PDF format, give an explicit reasoning (including recurrences and their closed forms) establishing the worst-case runtime complexities of these functions, under the assumption that the size of the extent is a constant. Also establish the runtime complexity of the query function for a quadtree where there is a constant number of rectangles in the union of the horizontal and vertical lists of every node and where all paths from the root to leaves are of the same length.
Grading

Your solution is graded with 0 to 100 points in the following way:

a. If your solution was submitted by the deadline, your program loads under working version 110.72 of SML/NJ and passes all the training test cases, and your solution is deemed by us to be a serious attempt at implementing, commenting (under at least the coding convention), and analysing all the requested functions, then you get 20 points; otherwise (including when no solution was submitted by the submission deadline), you get a U grade for this assignment (even if an insufficiently commented program is actually correct).

b. Your program is run on an unspecified number $t$ of orthogonal grading test cases that satisfy all pre-conditions but also check boundary conditions. Each test case is a quadtree creation for some extent $e$, followed by a sequence of insertions of rectangles, whose points are all inside $e$, followed by a sequence of queries for the lists (in any order) of rectangles of the resulting quadtree that contain some given points. For each fully correct test result, you get $50/t$ points. We reserve the right to run these tests automatically, so be careful with function names and argument orders.

c. Your program is graded for style and comments (including function specifications, datatype representation conventions and invariants, as well as recursion variants), provided it does not fail on all the grading test cases. This covers 10 points.

d. Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers the remaining 20 points.

Have fun!