Program Design & Data Structures (Course 1DL201)
Uppsala University – Autumn 2013/Spring 2014

Homework Assignment 2: Quadtrees

Prepared by Tjark Weber

Lab: Tuesday, 17 December, 2013
Submission Deadline: 18:00, Friday, 17 January, 2014
Lesson: TBA (likely in week 4 of 2014)
Resubmission Deadline: TBA (likely in week 5 of 2014)

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 key spaces are straightforward. These kinds of trees are useful in many graphics applications and computer-aided design tools, such as for the design of VLSI (very-large-scale integration) circuits. Unlike nodes in a binary tree, which have 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. Each rectangle is of the following type:

```
datatype rectangle = Rect of int * int * int * int
```

A rectangle `Rect (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. The normal convention in Cartesian geometry is followed—the coordinate system is such that as one goes toward the right and top, the x and y coordinates increase (see Figure 1). We assume the pre-condition `bottom` $<$ `top` and `left` $<$ `right` for any rectangle.

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

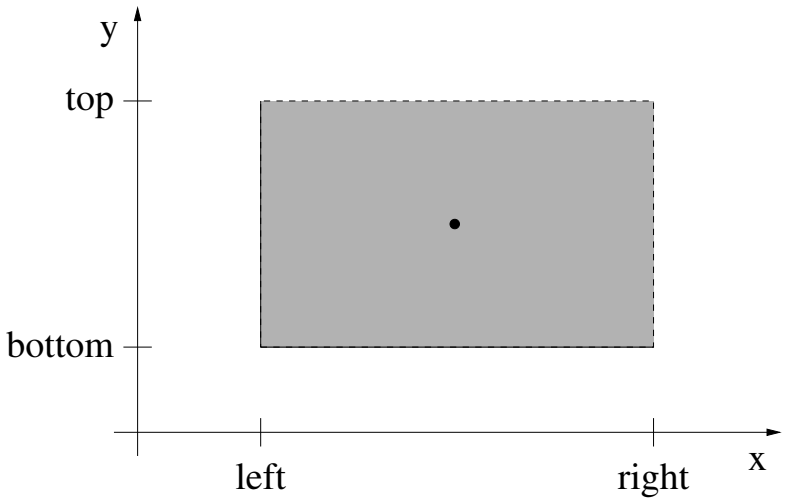

Figure 1: A rectangle. The black dot indicates its centre point.

A quadtree allows us to represent a collection of rectangles so that one can efficiently search for all rectangles containing a given point. (One can obviously also search if all the rectangles are simply kept in a list, but when there are millions of rectangles—for instance, when designing a microprocessor chip—this is very inefficient.) Quadtrees organise such two-dimensional information in the following way:

- A quadtree covers a fixed rectangular region of the plane, itself represented by a rectangle, called the *extent* of the tree. A quadtree only stores rectangles whose points are contained in this region.

- The centre point of the extent `Rect (left, top, right, bottom)` has integer coordinates $x = (\texttt{left} + \texttt{right})$ `div` 2 and $y = (\texttt{top} + \texttt{bottom})$ `div` 2, where `div` denotes integer division.

- This centre point partitions the extent into four smaller rectangles, called *quadrants*, located 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 a minimum quadrant size, is reached.

The next section explains in more detail how rectangles are stored in a quadtree.

Representing Rectangle Collections as Quadtrees

The `quadTree` datatype has the following definition:

```
datatype quadTree = EmptyQuadTree |
  Qt of rectangle * rectangle list * rectangle list *
        quadTree * quadTree * quadTree * quadTree
```

In a non-empty quadtree `Qt (extent, horizontal, vertical, topLeft, topRight, bottomLeft, bottomRight)`,

- the `extent` rectangle, say `Rect (left, top, right, bottom)`, defines the region covered by the quadtree,
- `horizontal` is the list of rectangles that contain some point of the horizontal centre line $y = (\texttt{top} + \texttt{bottom})$ `div` 2 (i.e., if some point of that line is inside a given rectangle, this rectangle is stored in `horizontal`),
- `vertical` is the list of rectangles stored in the quadtree that contain some point of the vertical centre line $x = (\texttt{left} + \texttt{right})$ `div` 2.

  If both centre lines have some point inside a given rectangle, then this rectangle is inserted only into the `vertical` list. For example, in Figure 2, rectangles 1 to 3 are on the `vertical` list for the root quadtree, while rectangles 4 and 5 are on its `horizontal` list.

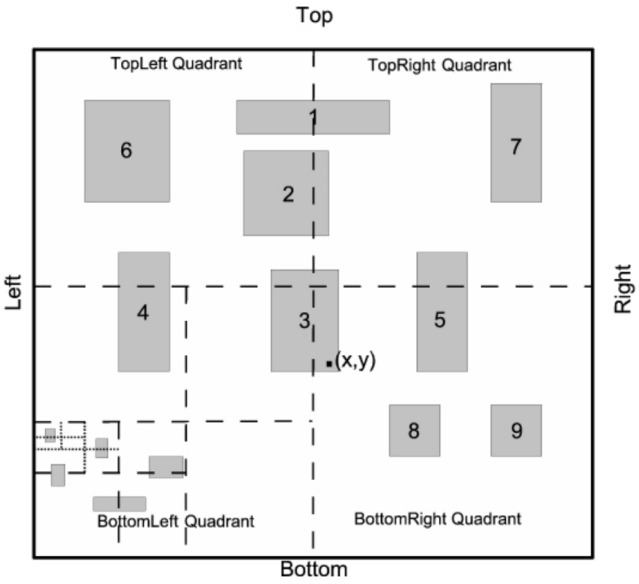

Figure 2: Storage of rectangles in a quadtree. The dashed lines are centre lines.

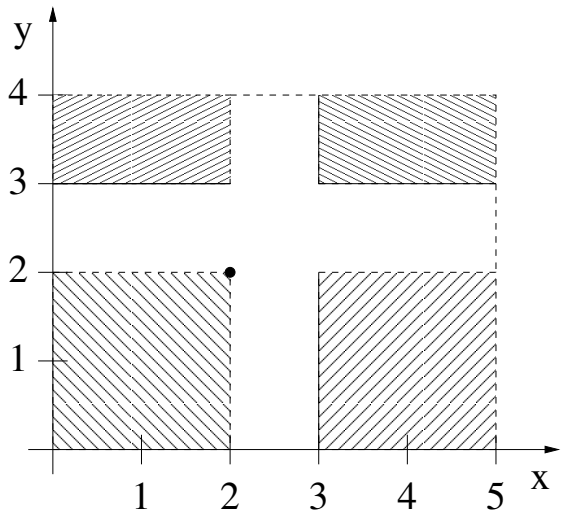

Figure 3: The extent `Rect (0,4,5,0)` with its four quadrants. The black dot indicates the centre point at $(2, 2)$. Note that since we are using integer coordinates, all points inside the extent that are not on its (horizontal or vertical) centre lines are inside one of the quadrants.

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

Example: For the extent `Rect (0,4,5,0)`, the `topLeft`, `topRight`, `bottomLeft`, and `bottomRight` extents are `Rect (0,4,2,3)`, `Rect (3,4,5,3)`, `Rect (0,2,2,0)`, and `Rect (3,2,5,0)`, respectively (see Figure 3).

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 that contain (x, y). Then continue to search recursively in the subtree covering the quadrant, if any, that contains the point; no recursive search is needed if (x, y) is on a centre line of the extent. For example, for the marked point (x, y) in Figure 2, one searches in the `vertical` and `horizontal` lists of the root extent and then only in the subtree that covers the bottom-right quadrant.

Work To Be Done

Implement the following functions in a file called `quadTree.sml`.

- `emptyQtree e` non-recursively returns an (otherwise empty) quadtree with extent `e`.
- `insert (q, r)` returns the quadtree `q` with rectangle `r` inserted, under the precondition that all points of `r` are inside the extent of `q`.
- `query (q, x, y)` returns the list (in any order) of rectangles in the quadtree `q` that contain the point `(x,y)`, where `x` and `y` are integers.

It is up to you whether you want to raise an exception if a precondition is violated. However, do not write any code that *handles* exceptions!

Make sure that your functions pass the test cases in `assignment2_tests.sml`. (These tests are not to be included in your solution.)

In a separate report in PDF format, do the following:

- Give 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 at the root of the quadtree is a constant.
- Establish in similar fashion the worst-case runtime complexity of the `query` function for a quadtree where there is a constant number of rectangles in `horizontal @ vertical` (i.e., in the concatenation of the `horizontal` and `vertical` lists) at every node and where all paths from the root node to leaves are of the same length.

Grading

Your solution is graded on a U/K/4/5 scale in three separate areas: (1) functional correctness, (2) style and comments, (3) complexity analysis.

1. Functional correctness:

Your program will be run on an unspecified number of grading test cases that satisfy all preconditions 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 of rectangles of the resulting quadtree that contain some given points.

We reserve the right to run these tests automatically, so be careful to match exactly the imposed file names, function names, and argument orders.

Advice: Run your code in a freshly started Poly/ML session before you submit, so that declarations that you may have made manually do not interfere when you test the code.

- If your solution was submitted by the deadline, your file `quadTree.sml` loads under Poly/ML version 5.5.1 and it passes all of the test cases in `assignment2_tests.sml`, you get (at least) a K for functional correctness.

  Otherwise (including when no solution was submitted by the deadline), you get a U grade for the homework assignment.

- If your program additionally passes at least 80% of the grading test cases, you get (at least) a 4 for functional correctness.

- If your program passes all grading test cases, you get a 5 for functional correctness. Note that some grading test cases may involve a large number of rectangles; your program needs to pass these tests in reasonable time (i.e., your program should not be unnecessarily inefficient).

2. Style and comments:

Your program is graded for style and comments according to our *Coding Convention*. The following criteria will be used:

- suitable breakdown of your solution into auxiliary/helper functions,
- function specifications and variants,
- datatype representation conventions and invariants,
- code readability and indentation,
- sensible naming conventions followed.

If your program's style and comments are deemed a serious attempt at following these criteria, you get (at least) a K for style and comments. Otherwise, you get a U grade for the homework assignment.

If you have largely followed these citeria, with very few major omissions or errors, you get a 4 for style and comments.

If you have followed these criteria with at most minor omissions or oversights, you get a 5 for style and comments.

3. Complexity analysis:

Your complexity analysis is graded for correctness of results and explicitness of reasoning.

If your report is deemed a serious attempt at establishing the required runtime complexities, you get (at least) a K for your complexity analysis. Otherwise, you get a U grade for the homework assignment.

If your complexity analysis is mostly correct and your reasoning is comprehensible, with very few major omissions or errors, you get a 4 in this area.

If your complexity analysis is correct and your reasoning is explicit, with at most minor omissions or oversights, you get a 5 in this area.

Final Grade

Area grades are converted to a final grade on the usual scale U/3/4/5 as follows:

1. You need to pass all three areas (functional correctness, style and comments, complexity analysis) in order to pass this assignment.

2. A K grade in *any* area means that you are required to attend the lesson discussing the assignment and to subsequently resubmit the assignment.

   After resubmission, your *entire* assignment will be regraded. Area grades of K will improve to 3 if you provide a mostly correct solution (cf. the criteria for grade 4); you cannot get a better grade than 3 in an area where you originally got a K. Area grades of 4 or 5 remain unchanged if your resubmission still meets the relevant grading criteria, but may be lowered otherwise (e.g., if your revised program no longer follows the coding convention).

3. Your final grade is the arithmetic mean of the three area grades.

Modalities

- The assignment will be conducted in groups of two students. Group divisions will be announced on the Student Portal.

- Solutions must be submitted via the Student Portal. Only one solution should be submitted per group. State the names of all group members both in a comment at the top of `quadTree.sml` and in your report.

Good luck!

By submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise. We reserve the right to use plagiarism detection tools and point out that they are extremely powerful.