`tag` — Version 1.0 ***due:*** *Mon, Nov 19th @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this assignment you will explore an application of binary search trees to develop efficient algorithms in computational geometry, which also happen to be useful in various database queries. First you will implement an ordered table using binary search trees that conforms to the ordered table signature. Then you will create a data structure that enables answering efficiently queries that report the number of 2-dimensional points within a specified rectangle. You will also analyze your implementation both in time and space. A common application of such 2-dimensional *range queries* is for counting the number of elements in a data set that are within two ranges. For example if a database maintained people by height and weight, one could ask for the number of people between 95 and 100 pounds who are between $5'6''$ and $5'9''$. Range queries can also return all elements in the range, but we don't require this for the homework.

1.1 Submission

Submit your solutions by placing your solution files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn8/`

Name the files exactly as specified below. You can run the check script located at

`/afs/andrew.cmu.edu/course/15/210/bin/check/08/check.pl`

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw08.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful. This file should be included at the top of your `.tex` file with the command `\input{<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
BSTOrderedTable.sml
BSTOrderedTableTest.sml
RangeCount.sml
RangeCountTest.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **Code is Art.** Code *can* and *should* be beautiful. Clean and concise code is self-documenting and demonstrates a clear understanding of its purpose.
2. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments should convince a reader that your code is correct.
3. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
4. **Use the check script and verify that your submission compiles.** We are not responsible for fixing your code for errors. If your submission fails to compile, you will lose significant credit.

2 Ordered Tables

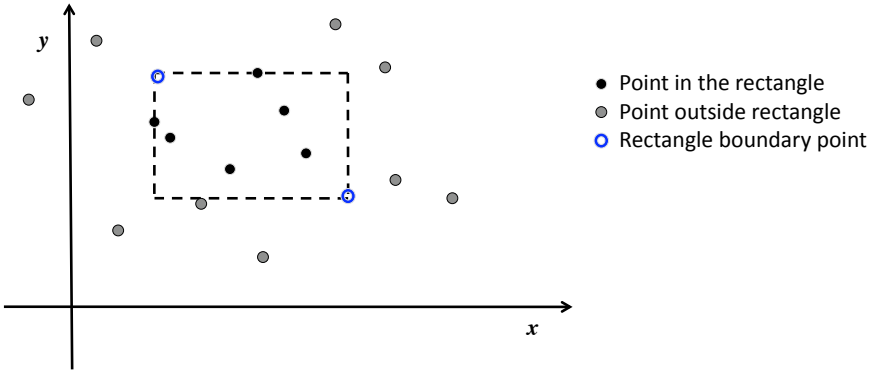
The abstract data types Table and Set allow for implementations based on types that have no natural ordering. Implementations can be based, for example, on hash tables or on binary search trees. The `BSTTable` implementation in the 15-210 library is based on a binary search tree and contains only the core tree functions needed to implement tables and sets. Tables that have keys from a totally ordered universe of element $\mathbb{K}$, however, can have additional operations that take advantage of the ordering. You will implement an ordered table based on binary search trees that conforms to the abstract data type *ordered table*. You will use the operations provided by ordered tables to support range queries in the next part.

Task 2.1 (25%). Create a functor `BSTOrderedTable` in the file `BSTOrderedTable.sml` that produces an implementation of the `ORD_TABLE` signature. Notice that you should not implement the original `Tables` operations, as they are included in the structure directly.

Task 2.2 (5%). In the file `BSTOrderedTableTest.sml`, test your `BSTOrderedTable` functor.

3 Range Queries

Suppose we are given a set of points $P = \{p_1, p_2, p_3, \cdots, p_n\}$, where $p_i \in \mathbb{Z}^2$. That is to say, these points are on the integer lattice. We want to be able to answer questions about the points within axis-aligned rectangular regions quickly. For example, we may want the number of points in the region, or the total "mass" of the points in the region. We will define a region by two diagonal opposing corner points of the rectangle: the upper-left corner and lower-right corner.



For example, given the points $\{(0,0),(1,2),(3,3),(4,4),(5,1)\}$, there are 2 points in the rectangle defined by $\{(2,4),(4,2)\}$ and 3 points in $\{(1,3),(5,1)\}$.

The obvious $O(n)$ algorithm to answer such queries is too slow. You will need to put the data into a data structure that will allow for $O(\log n)$ work queries. To do so, you should take advantage of persistence in functional programming and ordered tables.

Code from this section must be inside `RangeCount.sml` in the functor called `RangeCount` that ascribes to the signature `RANGE_COUNT`, defined as follows (the detailed specifications of these functions are given in the tasks below):

```
signature RANGE_COUNT =
sig
   structure Table : ORD_TABLE where type Key.t = int

   type countTable (* self type *)

   val makeQueryTable: Point2D.point Table.Seq.seq -> countTable
   val countInRange: countTable -> Point2D.point * Point2D.point -> int
end
```

Common to many computation geometry problems is the concept of a *sweep line*: Each point is considered when a vertical line crosses the point as it sweeps across the plane by increasing x coordinate (or by horizontal line that sweeps across by increasing y coordinate).

In this task, you will build a data structure from a sequence of two-dimensional points that will enable answering fast range queries. You might want to consider using the sweep line concept when building your data structure. In particular consider what data structure you maintain as you sweep and how you update this structure each time a new point is encountered. Also it might be helpful to first think about

how to answer a three sided query that given $(x_{top}, y_{bot}, y_{top})$ returns the number of points inside the rectangle that goes out to negative infinity on the x coordinate.

Task 3.1 (25%).

Implement the function `makeQueryTable: point seq -> countTable`. The type `point` is defined in the structure `Point2D`, available in the file `helperModules.sml`. The type `countTable` is a self type for your data structure that you will need specify. The choice of your data structure should allow you to to achieve $O(\log n)$ range query time.

Note: You may assume that the n input points have unique x coordinates and unique y coordinates and these integer coordinates are of type `int`.

For full credit, your `makeQueryTable` must run within $O(n \log n)$ expected work.

Task 3.2 (10%). Briefly describe how you would parallelize your code so that it runs in $O(\log^2 n)$ span. Does the work remain the same? You don't need to formally prove the bounds, just briefly justify them.

Solution 3.2 Instead of using `iterh` with `insert` to build a tableof tables, we can use `scan` with `merge`, which will run in parallel and achieve the same effect. As we analyzed before in Practice Midterm 1, the work of the parallel version is $O(n \log^2 n)$, where the merge routine merges tables of sizes m and n, $m < n$ in $O(m \log(1 + \frac{n}{m}))$ work.

Task 3.3 (5%). What is the expected space complexity of your `countTable` in terms of n the number of input points? That is, how many nodes in the underlying binary search tree(s) does your `countTable` use in expectation? Explain in a few short sentences.

Solution 3.3 The expected space complexity is $O(n \log n)$. Since the algorithm in Task 3.1 has work $O(n \log n)$, we can't possibly use more than that much space. The parallel version generates the same set of Treaps as the sequential version (Treaps are unique for the same set of keys and priorities) and hence has the same bound. Indeed, if we look more closely, we'll see that when inserting into a Treap, we're effectively creating about $O(\log n)$ nodes. This means that after n iterations, the number of distinct nodes can be at most $O(n \log n)$, again, in expectation.

Task 3.4 (25%). In this task, you will implement the function `countInRange: countTable -> Point2D.point * Point2D.point -> int`, which takes as input a `countTable` and two points that demarcate the bounds of an axis-aligned rectangle. More specifically, `countInRange T` $((x_1, y_1), (x_2, y_2))$ will report the number of points within the rectangle with the top-left corner (x_1, y_1) and bottom-right corner (x_2, y_2). Your function should return the number of points *within and on* the boundary of the rectangle. You may find the `Table.size` function useful here. Its use is an example of augmenting a binary search tree to enable fast queries.

Task 3.5 (5%). In the file `RangeCountTest.sml`, test out your `RangeCount` functor.