

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.

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 this is a much more involved task, so we don't require this for the homework.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf rangelab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. Makefile
2. Tests.sml
3. *MkBSTOrderedTable.sml
4. *MkRangeCount.sml

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

You should also modify Tests.sml to test your code for this lab, even though it will not be handed in, as this lab has no public tests on autolab.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `rangelab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the “*Handin your work*” link.

4 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 elements \mathbb{K} , however, can have additional operations that take advantage of the ordering. In this section, you will implement an ordered table based on binary search trees that conforms to the abstract data type *ordered table*. Then, in the next part, you will use the operations provided by ordered tables to support efficient range queries.

Documentation for the BST signature is on the course website (<http://www.cs.cmu.edu/~15210/docs/>). Your implementations should be as efficient as possible.

Task 4.1 (5%). In `MkBSTOrderedTable.sml`, implement the functions

```
val first : 'a table -> (key * 'a) option
val last  : 'a table -> (key * 'a) option
```

Given an ordered table T , `first T` should evaluate to `SOME (k, v)` iff $(k, v) \in T$ and k is the minimum key in T . Analogously, `last T` should evaluate to `SOME (k, v)` iff $(k, v) \in T$ and k is the maximum key in T . Otherwise, both functions evaluate to `NONE`.

Task 4.2 (5%). In `MkBSTOrderedTable.sml`, implement the functions

```
val previous : 'a table -> key -> (key * 'a) option
val next     : 'a table -> key -> (key * 'a) option
```

Given an ordered table T and a key k , `previous T k` should evaluate to `SOME (k', v)` if $(k', v) \in T$ and k' is the greatest key in T strictly less than k . Otherwise, it evaluates to `NONE`. Similarly, `next T k` should evaluate to `SOME (k', v)` iff k' is the least key in T strictly greater than k .

Task 4.3 (3%). In `MkBSTOrderedTable.sml`, implement the functions

```
val join : 'a table * 'a table -> 'a table
val split : 'a table * key -> 'a table * 'a option * 'a table
```

Given ordered tables L and R , where all the keys in L are strictly less than those in R , `join (L, R)` should evaluate to an ordered table containing all the keys from *both* L and R .

Given an ordered table T and a key k , `split (T, k)` should evaluate to a triple consisting of 1) an ordered table containing every $(k', v) \in T$ such that $k' < k$, 2) `SOME v` if $(k, v) \in T$ and `NONE` otherwise, and 3) an ordered table containing every $(k', v) \in T$ such that $k' > k$.

Task 4.4 (7%). In `MkBSTOrderedTable.sml`, implement the function

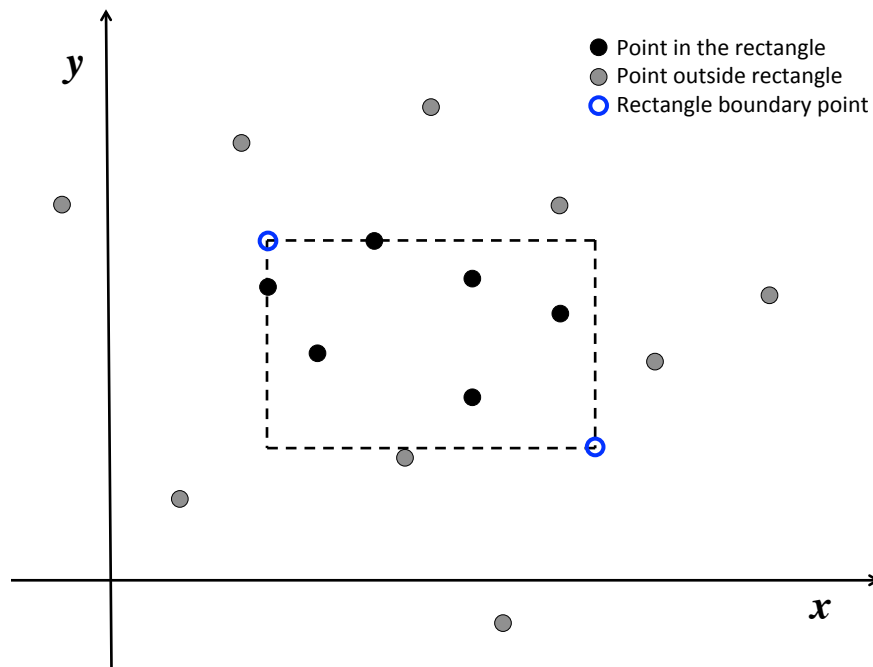
```
val getRange : 'a table -> key * key -> 'a table
```

Given an ordered table T and keys l and h , `getRange T (l, h)` should evaluate to an ordered table containing every $(k, v) \in T$ such that $l \leq k \leq h$.

5 Range Query

Suppose we are given a set of points $P = \{p_1, p_2, p_3, \dots, p_n\}$, 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 diagonally-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 figure below depicts a more generic view (not the example above), of which points belong in the bounding box, and which don't.



There is an obvious $O(n)$ algorithm to answer such queries. In this lab, you will need to create a data structure (using ordered tables) that will allow for $O(\log n)$ work queries on a given set of points.

Common to many computational 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 a horizontal line that sweeps across the point set by increasing y coordinate).

You might want to consider using the sweep line concept when building your efficient data structure. In particular, consider incrementally constructing your data structure: given a valid data structure for the first i points seen, how can you update this structure to be valid for the first $i + 1$ many points? It might also 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 $x = -\infty$.

Task 5.1 (30%). In `MkRangeCount.sml`, define the `countTable` type and implement the function

```
val makeCountTable : point seq -> countTable
```

The type `point` is defined to be `OrdTable.Key.t * OrdTable.Key.t` where `OrdTable` is an ordered table structure provided to you. You should choose the type of `countTable` such that you can implement `count` (range queries) in $O(\log n)$ work and span. For full credit, your `makeCountTable` implementation must have $O(n \log n)$ work and span.

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

Task 5.2 (10%). Briefly describe how you would parallelize your implementation of `makeCountTable` so that it has $O(\log^2 n)$ span. Does the work remain the same? You don't need to formally prove the bounds; just briefly justify them.

Task 5.3 (15%). What is the *expected* space complexity of your `countTable` implementation, 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 just a few short sentences.

Task 5.4 (25%). In `MkRangeCount.sml`, implement the function

```
val count : countTable -> point * point -> int
```

As described earlier, `count T ((x_1, y_1), (x_2, y_2))` will report the number of points within the rectangle defined by 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 `OrdTable.size` function useful here.

For full credit, your implementation must have $O(\log n)$ work and span. **Note:** we are only interested in the *number* of points in the given rectangle, not the points themselves.

6 Testing

You do *not* have to submit test cases for this lab. However, it is in your best interest to test your code thoroughly before submission, as this lab has no public tests on Autolab. A testing framework has been provided for you to make it easy to test your code locally. In the file `Tests.sml`, add your tests to the appropriate list (look at the sample test cases for reference). Then, in the terminal, run:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testFirst ();
- Tester.testLast ();
- Tester.testPrev ();
- Tester.testNext ();
- Tester.testJoin ();
- Tester.testSplit ();
```

- `Tester.testRange ();`
- `Tester.testCount ();`

For this lab, please do not share test cases! Part of the reason we are not running public tests is to force you to think of test cases to test your own code.