Lab 6 - *Searching in Two-dimensional Spaces*

Due: *Wed, Mar 24th, 2021 @ 23:59*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring 2021)

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. You will create a data structure which supports efficient queries for the sum of weights of 2-dimensional points within a specified rectangle.

A common application of such 2-dimensional *range queries* is counting the number of elements in a data set restricted by two ranges. For example, if a database maintained people by height and weight, one could ask for the number of people between 60 and 80 kgs who are between 170 cm and 176 cm.

2 Files

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

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

from a terminal window on a unix/linux system. 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 that will be handed in terms of code

1. * `RangeSum.sml`
2. `Sandbox.sml`
3. `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment. Your written answers should be composed in a suitable word processor such as LaTeX, Word, OpenOffice, etc. *Handwritten (and scanned in) answers will NOT be accepted.* **Make sure your name is also in the `written.pdf` file.**

3 Submission

To submit your assignment to Autolab in Diderot, generate a handin archive `handin.tgz` with the command

```
make package
```

You can then submit your `handin.tgz` file on Diderot. You should submit the answers to the written questions in *written.pdf* to Gradescope at `https://www.gradescope.com/courses/225303`.

Note that your code submitted to Autolab may not be graded until after the due date. It is your responsibility to test your code thoroughly before submitting.

3.1 Important points

1. Before submission, always, *always* verify that the files you will submit are indeed the versions that you intended to submit. Be very careful to NOT submit the wrong lab's `written.pdf`, blank files, or old versions of the files. **We will not entertain "*Oh no, I submitted the wrong file!*" emails or Diderot posts after the due date**. We will adopt a *zero-tolerance* approach to this, and you will receive the grade of your most recent submission (which may be subject to late submission penalties!) *You have been warned*.

2. If the homework has a written component, the answers to each written question in *written.pdf* should be on one page and each answer should fit on one page. If you decide not to answer one of the questions, still have the empty page in the file. This will make everyone's life easier in Gradescope.

3. If the written part has a question that asks you to show that your code has a certain complexity bound, **you should include a copy of your code in your answer and annotate it as suitable with relevant complexity bounds using comments and then argue or compute the resulting bounds. Answers to such question without a copy of the code you submitted will not be accepted.**

4. The questions below ask you to organize your solutions in a number of modules written almost from scratch. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

5. You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

6. This assignment requires the use of library code spread over several files. The compilation of this is orchestrated by the compilation manager through several `.cm` files. Instructions on how to use CM can be found in the file `cm.pdf` under Resources on Diderot.

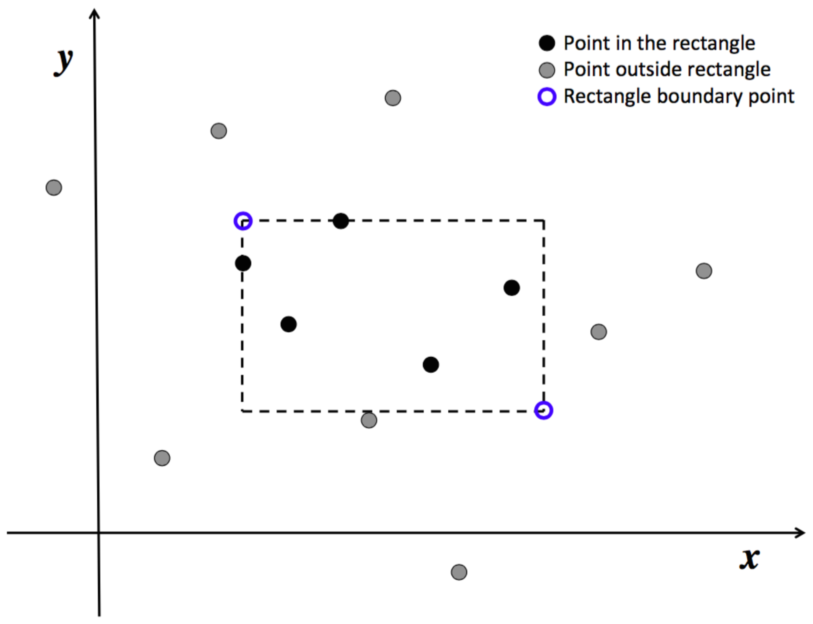
7. Style grading for this assignment will be evaluated on a 5 point scale: -2 through +2. Style points will be added to your final grade from the rest of your homework. **You should review the full style guide available on the Diderot under Resources.**

8. **You should submit your solutions on or before the deadline on the top of the first page. Failure to do so will cause substantial deductions of your grade. Late submissions of up to a day will scale your homework grade by 70% (so if you were to get 100% as your grade, you will now get 70%). Late submissions of up to 2 days will scale your homework grade by 50%. Beyond 2 days, you will receive 0 as your homework grade. This scaling will be handled outside Autolab.**

4 Range Query

Suppose we are given a set of points $P = \{p_1, p_2, p_3, \ldots, p_n\}$, 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. Any point on the edge of the rectangle should be used in the results.

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 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** which, given $(x_{top}, y_{bot}, y_{top})$, produces a valid query for the points inside the rectangle that goes out to $x = -\infty$.

Task 4.1 (80 pts). In `RangeSum.sml` implement the function

```
val rangeSum :  (point * int) Seq.t → (point * point) → int
```

such that given S where $S[i] = ((x_i, y_i), w_i)$, the expression (`rangeSum` S $((a, b), (c, d))$) returns

$$\sum_{\substack{a \le x_i \le c \\ d \le y_i \le b}} w_i.$$

That is, it should return the sum of weights of points which lie within the rectangle given by top-left coordinate (a, b) and bottom-right coordinate (c, d). Note that the boundaries of the rectangle are *inclusive*. The elements in S are not necessarily sorted. You may assume all x_i's are unique, and all y_i's are unique.

Your implementation must be **staged** such that it requires $O(|S| \log |S|)$ work and span upon application of the first argument, and $O(\log |S|)$ work and span upon application of the second. For example, in the following code, line 1 should require $O(|S| \log |S|)$ work and span, while lines 2 and 3 should each require $O(\log |S|)$ work and span. You can read more about staging in the Appendix.

```
1 val query = rangeSum S
2 val _ = query ((a,b),(c,d))
3 val _ = query ((e,f),(g,h))
```

You will need to build your own table structures for this task. Specifically, you should consider using the functors `MkTreapTable` and/or `MkTreapAugTable`. Refer to the Appendix to see how to use them.

Hint/Warning: you will need to use tables augmented with reduced values to solve this problem. This means that you will supply your own reducing function f. In lecture, we argued that as long as f is constant-work, then the cost bounds of all table functions are unaffected by augmentation. For non-constant-work reducing functions, this is not the case. **In this assignment, the reducing function you choose for augmentation must be a constant-work function**.

5 Testing

There are two ways to test your code.

1. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

```
- CM.make "sandbox.cm";
...
- open Sandbox;
```

2. In `Tests.sml`, add test cases according to the instructions given. Then run the autograder:

```
- CM.make "autograder.cm";
...
- Autograder.run ();
```

6 Analysis

Task 6.1 (10 pts). Suppose we augment a binary search tree with a non-constant-work function to compute reduced values. Specifically, suppose the values of our BST are singleton sequences and we augment with `ArraySequence.append`. If the tree is perfectly balanced, how expensive is it to update a key with a new value? (Assume the only change is the value associated with some key. The structure of the tree remains the same: perfectly balanced.) Give your answer as a tight Big-O bound in terms of n, the size of the BST.

7 Appendix

7.1 Tables in SML

The 15-210 library contains functors `MkTreapTable` and `MkTreapAugTable`. How do we use them?

7.1.1 Instantiating Tables

Due to limitations of SML, a table mapping some type α to another type β cannot be written `(`α`, `β`) Table.t`. Instead, we must fix the key type with a substructure. The value type remains polymorphic. For example, a structure `Table : ORDTABLE` contains a substructure `Key : ORDKEY`; tables of type α `Table.t` have keys of type `Table.Key.t` and values of type α.

In order to instantiate a table implementation, we need to provide a key structure to the `MkTreapTable` functor. Many key structures are already provided in the 15-210 library: `IntElt`, `StringElt`, `RealElt`, etc. For example, in the following, a table of type α `IntTable.t` has keys of type `int`.

```
structure IntTable = MkTreapTable (structure Key = IntElt)
```

7.1.2 Tables Augmented with Reduced Values

Our library supports tables parameterized by user-defined augmentation. These tables ascribe to `AUG_ORDTABLE`, and have two substructures:

- `Key : ORDKEY`, which fixes the key type of the table.
- `Val : MONOID`, which fixes the value type of the table.

Inside the `Val` substructure, we define values f and I which are an associative function and identity, respectively, such that the table can calculate reduced values automatically. (We also need to define `toString`.) For example,

```
structure MyVal =
struct
  type t = int
  val f : t * t → t = op*
  val I : t = 1
  val toString : t → string = Int.toString
end

structure MyAugTable =
  MkTreapAugTable (structure Key = StringElt
                   structure Val = MyVal)
```

In the above, a table of type `MyAugTable.t` has keys of type `string` and values of `int`. The expression (`MyAugTable.reduceVal` T) returns the product of all values in T.

7.2 Staging

A staged function is one which accepts multiple arguments, and can be partially evaluated as soon as some of the arguments are known. In SML, the body of a function is never evaluated until all of the arguments are known. In order to stage a function, we must do so explicitly by returning a new function after some of the arguments are known.

For example, suppose that we have a function f : `int` $\rightarrow$ `int` where $f(n)$ performs $\Theta(n)$ work. Then define g as

```
fun g n = let val x = f(n) in (fn m ⇒ x * m) end
```

In the following code, line 1 performs $\Theta(n)$ work while lines 2 and 3 each perform only $\Theta(1)$ work.

```
1 val r = g(n)
2 val _ = r(1)
3 val _ = r(2)
```

If we had instead defined g by **fun** $g\ n\ m = f(n) * m$ then line 1 would perform $\Theta(1)$ work while lines 2 and 3 would each perform $\Theta(n)$ work.