Lab 5 - *Navigating Binary Search Trees with "Fingers"* **Due:** *Wed, Mar 17th, 2021 @ 23:59*

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

1 Introduction

This assignment is designed to help strengthen your understanding of binary search trees. You will implement a specialized search called *finger search* and analyze its performance in the context of treaps.

2 Files

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

```
tar -xvf fingerlab-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. * `MkFinger.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.

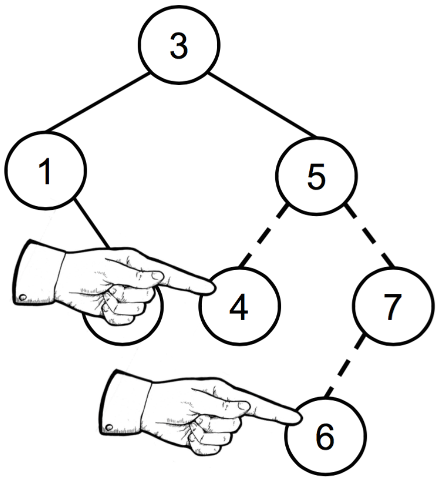
6. This assignment requires the use of library code spread over several files. The compilation of this is orchestrated by the compilation manager through several `.cm` files. Instructions on how to use CM can be found in the file `cm.pdf` under Resources on Diderot.

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 Finger Search

When we search for a node in a binary search tree, we typically start from the root and work our way downwards. But what if we want to start from a node other than the root? The node we're searching for might not be a descendent of the starting point. In general, we may have to walk up the tree a ways before turning around and searching downwards like normal. This type of search is often called a *finger search* because it is facilitated by *fingers*.



Fingers A finger is a structure which provides quick access to nodes near a distinguished location. You should think of a finger as "pointing at" a particular node, and as being easily movable to a parent or child of that node. When finger searching, we provide a finger as a starting point, and in return expect a finger at the destination.

Fast Searches We'll show later that for treaps, we can finger search between the i^{th} and j^{th} largest keys in expected $O(\log(j - i + 1))$ work. This suggests that, for example, we can conveniently iterate across a tree in linear work by expressing it as a sequence of finger searches.

4.1 Implementation

In an imperative setting, fingers are often implemented as pointers. This has the downside of requiring modifications to the underlying tree: every node needs a parent pointer to facilitate travelling upwards. In this lab, we're interested in implementing fingers as auxiliary data structures which are separate from the tree. In theory, we'll be using less space than the alternative, because we don't need an additional pointer at every node just to support a single finger. Our approach is also better suited for a purely functional style.

The following code is parameterized on a `Tree` structure, ascribing to BST (see the library documentation). Since you won't be constructing any new trees, you should only need the `Tree.Key` substructure and the function `Tree.expose`.

Task 4.1 (0 pts). In `MkFinger.sml`, decide on a representation for the type α `finger`. As a hint, consider encoding a finger as a list of ancestors. These should form a path between the distinguished node and the root of the tree.

In order to complete a search in work proportional to the length of the unique path from the starting node to the destination, you should decide, at each node, between moving to the parent or to the child. In particular, you might want to consider supporting queries on a finger that returns a range `(k1, k2)`, where all keys `k` in the subtree rooted at the finger satisfy `k1 < k < k2`. As an example, for the given treap, the ranges for each node are as follows:

Node	1	2	3	4	5	6	7
Range	$(-\infty, 3)$	$(1, 3)$	$(-\infty, \infty)$	$(3, 5)$	$(3, \infty)$	$(5, 7)$	$(5, \infty)$

Task 4.2 (70 pts). In `MkFinger.sml`, implement the following functions.

```
val lift : α finger → Tree.Key.t * α
```

Given a finger f, (`lift` f) should return the key and value stored at the node which f points to. The name is inspired by "lifting" your finger off the tree and looking at the data underneath. Your implementation must take constant work.

```
val root : α Tree.t → α finger option
```

Given a tree t, (`root` t) should return a finger pointing to the root node of t, or `NONE` if t is empty. Your implementation must take constant work.

```
val searchFrom : α finger → Tree.Key.t → α finger option
```

Given a finger f pointing at a node n, and a key k, (`searchFrom` f k) should return a finger pointing to the node containing k, or `NONE` if k is not in the tree. Your search must start at the node n, and must take work proportional to the **length of the unique path** between n and the node containing k. If k is not in the tree, then use the unique path between n and the leaf in k's position instead.

```
val first : α Tree.bst → α finger option
```

Given a tree t, (`first` t) should return a finger pointing to the leftmost node of t, or `NONE` if t is empty. Your implementation must take work proportional to the length of the left spine of t.

```
val next : α finger → α finger option
```

Given a finger f, (`next` f) should return a finger pointing to the next node in the tree, or `NONE` if f points to the rightmost node of the tree. This function is a special case of `searchFrom`, and as such must have the same cost bounds. For example, (`next` f) where f is pointing at the rightmost node should return `NONE` in constant work. Note that the combination of `first` and `next` allows us to iterate over a tree.

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 for Treaps

For any two nodes in a treap, how long is the path between them?

Consider a treap formed from the sorted sequence of keys S, $|S| = n$, and recall that every key is assigned a uniform random priority. Let $\ell_{i,j}$ be a random variable for the number of distinct nodes on the path between S_i and S_j, inclusive on both ends. Here are a couple more useful random variables:

$$A_i^k = \begin{cases} 1, & \text{if } S_k \text{ is an ancestor of } S_i \\ 0, & \text{otherwise} \end{cases} \qquad A_{i,j}^k = \begin{cases} 1, & \text{if } S_k \text{ is an ancestor of } \textit{both } S_i \text{ and } S_j \\ 0, & \text{otherwise} \end{cases}$$

For simplicity, we'll say that every node is its own ancestor. We'll also assume throughout that $0 \le i \le j < n$.

From lecture, we know that $\mathbf{E}\left[A_i^k\right] = \frac{1}{|k-i|+1}$.

Task 6.1 (10 pts). Give $\mathbf{E}\left[A_{i,j}^k\right]$ in terms of i, j, and k.
(*Hint:* separately consider the cases $k < i$, $i \le k \le j$, and $j < k$.)

Task 6.2 (10 pts). Give $\ell_{i,j}$ in terms of the random variables A_i^k and $A_{i,j}^k$. Be careful about off-by-one errors.

Task 6.3 (17 pts). Show that

$$\mathbf{E}\left[\ell_{i,j}\right] = 4H_{j-i+1} + (H_{i+1} - H_{j+1}) + (H_{n-j} - H_{n-i}) - 3$$

where H_k is the kth harmonic number. This question relies on having correct answers for the previous tasks. You should double check them before continuing.

Task 6.4 (10 pts). Use the result of the previous task to show that finger searching between S_i and S_j is expected $O(\log(j - i + 1))$.[1] You should also use the fact that $H_k \le 1 + \ln k$.

[1]We often pronounce this bound as either “logarithmic in the distance between the keys,” or “logarithmic in the difference in ranks of the keys.”

7 Additional Analysis

(This task pertains to treaps, but not necessarily fingers.)

Recall the BST function `join`, which combines two binary search trees in logarithmic work. For treaps, its implementation is as follows. We've omitted the polymorphic value stored at each node for brevity.

```
fun join ($T_1$, $T_2$) =
  case ($T_1$, $T_2$) of
    (Leaf, _) ⇒ $T_2$
  | (_, Leaf) ⇒ $T_1$
  | (Node ($L_1$, $k_1$, $p_1$, $R_1$), Node ($L_2$, $k_2$, $p_2$, $R_2$) ⇒
      if ($p_1 > p_2$) then
        Node ($L_1$, $k_1$, $p_1$, join ($R_1$, $T_2$))
      else
        Node (join ($T_1$, $L_2$), $k_2$, $p_2$, $R_2$)
```

Notice that this implementation stores priorities explicitly at each node. It is sometimes advantageous (or necessary, even) to not store priorities. Instead, we'd like to simply "witness their effects" – i.e., we'd like to simulate the outcome of "$p_1 > p_2$" without actually knowing the values of p_1 and p_2.

Task 7.1 (8 pts). Suppose we rewrite line 6 as a biased coin flip with probability of heads equal to $\mathbf{Pr}[p_1 > p_2]$. Give $\mathbf{Pr}[p_1 > p_2]$ in terms of $|T_1|$ and $|T_2|$. Please provide a justification for your reasoning.