PPA-S25 hw5

Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There are 7 tasks total: 2 coding tasks, and 5 written tasks.

You should submit your work on Brightspace.

Create a file `written.pdf` with your solutions to the written tasks. Typeset solutions are preferred. Handwritten solutions will be accepted only if they are **easily legible**.

Package your solutions as follows:

```
# replace NYU_NET_ID with your ID, e.g., shw8119.tgz
$ tar czf NYU_NET_ID.tgz Splitters.sml written.pdf
```

The resulting `.tgz` archive should contain only `Splitters.sml` and `written.pdf`. To submit, upload this `.tgz` file.

Due Date: This submission is due at **5:00pm EST on Monday, Mar 3**. Course policy on late submissions is available on the course website. (https://cs.nyu.edu/~shw8119/courses/s25/3033-121-ppa/)

Setup

- Follow the instructions on the course website to access one of the `crunchy` compute servers.
- Install MaPLe. For the `crunchy` servers, you can:
  - Download https://cs.nyu.edu/~shw8119/courses/s25/3033-121-ppa/resources/mpl-v053.tgz and copy it to the server
  - Unpack by running `tar xzf mpl-v053.tgz`. This will create a directory `mpl-v053/`
  - The compiler, `mpl`, is located at `mpl-v053/bin/mpl`
  - Add `mpl-v053/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-v053/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

Quick Preliminary: Parametric, Augmented BSTs

Take a look at the files `TREE_DATA.sml` and `JUST_JOIN.sml`.

The first describes types and values that can be freely defined by a programmer, including:

- the types of the keys-value pairs in the tree, together with a comparison function on the keys
- the types of the *augmented values*, and how they are computed, including an associative combining function for augmented values.

Given these definitions, we can build an implementation of a balanced binary search tree which provides the user with the `JUST_JOIN` interface. The file `AVL.sml` provides one such implementation, with AVL-tree rebalancing. Other balancing criteria could be implemented, providing exactly the same interface. The implementations in `TreeFuncs.sml` then provide common functionality, but are entirely parametric: any module that ascribes to the `JUST_JOIN` interface can be passed as argument to `TreeFuncs`.

The `JUST_JOIN` interface, as the name suggests, is built around the function `join`. This interface has been designed very carefully to ensure that the underlying balancing scheme remains hidden and that the invariants of the balancing criteria are not broken.

The way this is accomplished is by hiding the actual representation of the tree, with an "opaque" type `tree`. The only way to observe a tree is to call `expose`, which returns either `Leaf` or `Node(l,k,v,r)` where `l` and `r` are the two child subtrees and `(k,v)` is one key-value pair. (Note that both `l` and `r` have type `tree`, so you would have to call `expose` again on these to look inside them.)

To put trees back together again, you have to call `join`, which has type `join: exposed -> tree`. This function restores balance if necessary.

Any tree you can possibly construct using this interface will be balanced. You don't have to check this yourself; it is guaranteed by the interface.

```
(* opaque, can't see inside *)
type tree
(* take a look at just the root element of a tree
 * using the function "expose" *)
datatype exposed = Leaf | Node of tree * key * value * tree
val expose: tree -> exposed
(* any exposed tree can be put back together again by
 * calling `join` *)
val join: exposed -> tree
```

Note that the typical usage of this interface often constructs a `Node` and then immediately calls `join` on it. For example, here is an implementation of the function `split` we considered in lecture, now rewritten for this interface.

```
fun split (t, k) =
  case expose t of
    Leaf => (join Leaf, false, join Leaf)
  | Node (l, k', v', r) =>
      case key_compare (k, k') of
        EQUAL => (l, true, r)
      | LESS =>
          let val (ll, b, lr) = split (l, k)
          in (ll, b, join (Node (lr, k', v', r)))
          end
      | GREATER =>
          let val (rl, b, rr) = split (r, k)
          in (join (Node (l, k', v', rl)), b, rr)
          end
```

One funny thing you'll notice is that we write `join Leaf` to construct an "opaque" leaf from an exposed leaf. This may seem silly, but it is actually important for hiding the entire implementation of the "opaque" tree and making sure that no component of it is visible to the client of the library.

Split Paths

For a tree `t`, define the **split path** of a key `k` to be a sequence of the results of all of the calls to `key_compare` that `split(t,k)` makes.

For example, in the figure below, the split path of the key 5 is: `LESS, GREATER, LESS, LESS, EQUAL`.

Task 1 (5 points). In the example tree shown below, what is the split path for key `3`?

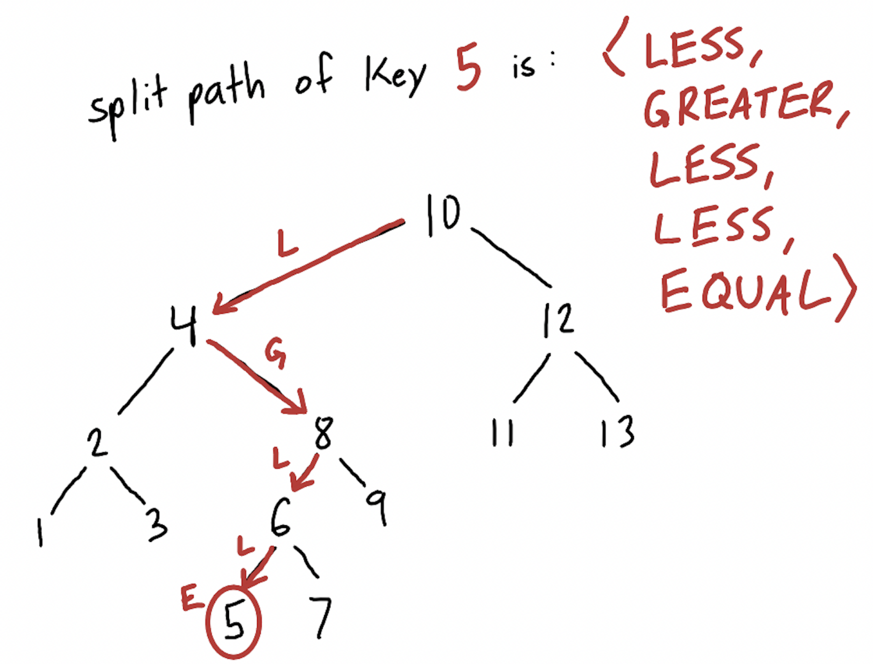

Figure 1: Example split path

Taking only what you need

Often, you want to split a tree but only need one of the two output trees. Defining special-purpose splitting routines (that don't construct the unneeded output) is beneficial for practical performance.

Task 2 (20 points). In `Splitters.sml`, implement the function `split_leq: tree * key -> tree` where `split_leq(t, k)` returns a tree containing all keys from `t` that are less-than-or-equal-to `k`. Your solution should call `join` **at most** $g + 1$ times, where g is the number of times either `GREATER` or `EQUAL` appears in the split path for `k`.

Task 3 (20 points). In `Splitters.sml`, implement the function `split_geq: tree * key -> tree` where `split_geq(t, k)` returns a tree containing all keys from `t` that are greater-than-or-equal-to `k`. Your solution should call `join` **at most** $\ell + 1$ times, where ℓ is the number of times either `LESS` or `EQUAL` appears in the split path for `k`.

You can test these functions as follows. Feel free to add more tests to the top of `test.sml`.

```
$ make test
$ ./test
```

Split Efficiency?

Suppose you wanted to extract all key-value pairs that lie within a range of keys defined by two keys, `lo: key` and `hi: key`. Below are two possible implementations.

The first implementation, called `split_between`, uses the `split_leq` and `split_geq` functions you completed in the previous task. We know that `split` takes $O(\log n)$ work on balanced trees, and therefore each of `split_leq` and `split_geq` require at most $O(\log n)$ work. Altogether, the amount of work required for `split_between` is $O(\log n)$ on a balanced tree.

```
fun split_between (t, lo, hi) =
  split_leq (split_geq (t, lo), hi)
```

The second implementation, `split_between_alt`, takes a different approach. The code cases on three possibilities for where the key at the root lies relative to the range `[lo,hi]`: below, above, or somewhere within.

```
fun split_between_alt (t, lo, hi) =
  case expose t of
    Leaf => t
  | Node (l, k, v, r) =>
      if key_compare (k, lo) = LESS then
        (* k < lo, so we know [lo,hi] lies entirely within r *)
        split_between_alt (r, lo, hi)
      else if key_compare (k, hi) = GREATER then
        (* k > hi, so we know [lo,hi] lies entirely within l *)
        split_between_alt (l, lo, hi)
      else
        (* pieces of both l and r overlap with the range *)
        let val (l', r') =
              ForkJoin.par (fn () => split_between_alt (l, lo, hi),
                            fn () => split_between_alt (r, lo, hi))
        in join (Node (l', k, v, r')) end
```

This `split_between_alt` implementation is correct, but it might not be immediately clear how well it performs.

Task 4 (10 points). Consider some call `split_between_alt(t,lo,hi)`. Let n be the number of keys in the input `t`, and let b be the number of keys in the output. **Exactly how many calls to `join` are executed?** (Specifically, we call `join` only on the last line. If we count 1 every time the last line of code occurs, what will the total count be, **exactly**? No asymptotics here! Please give an exact count in terms of n and/or b.)

Task 5 (15 points). Argue (informally, in words, in just a few sentences) that the amount of work required for `split_between_alt(t,lo,hi)` can be upper-bounded by $O(b \log(1 + \frac{n}{b}))$, where n is the number of keys in `t` and b is the number of keys in the output.

Hint: You may assume that `union`, `intersection`, and `difference` require $O(m \log(1 + \frac{n}{m}))$ work on trees of size n and m where $n \geq m$. There's a connection here...

Task 6 (10 points). Describe an example where `split_between_alt` would require $O(n)$ work, but `split_between` requires only $O(\log n)$.

Non-constant Augmentation

Imagine you had a function `update_key: tree * key * value -> tree` where `update_key(t, k, v)` finds key `k` in the tree and replaces that node with the key-value pair `(k,v)` (and constructs new ancestor nodes appropriately). This function does **not** change the structure of the tree, it only changes values (and augmented values) stored at nodes.

In lecture, we considered only constant-time augmentation of trees, i.e., where the functions `f: key * value -> avalue` and `g: avalue * avalue -> avalue` each require $O(1)$ work and span. In this setting, the cost of `update_key(...)` is always $O(\log n)$ for balanced trees.

Suppose we instead augmented a tree as follows. These would cause the augmented value at the root to be a sequence of all the keys in the tree.

```
type key = ...    type value = ...   type avalue = value Seq.t
fun f(k, v) = Seq.singleton v
fun g(v1, v2) = Seq.append (v1, v2)
val z = Seq.empty()
```

Task 7 (20 points). Suppose you have a *perfectly balanced* tree (where for every node, the heights of its two children are exactly equal), with n elements, augmented with `f` and `g` as defined above. Write work and span recurrences for `update_key` in terms of n, and solve these recurrences, providing tight big-O work and span bounds. You may assume that `append(`s_1`,`s_2`)` requires $O(|s_1| + |s_2|)$ work and $O(\log(|s_1| + |s_2|))$ span.