CS51 PROBLEM SET 4:
MODULES, FUNCTORS, AND PRIORITY QUEUES

STUART M. SHIEBER

You may work with a partner on this problem set. If you choose to work with a partner, you can find instructions for configuring GitHub Classroom here.

Introduction

In this assignment you will learn about OCaml modules and how to use modules to create data structure interfaces with useful abstractions. The particular data abstractions that you'll be implementing are ordered collections (as implemented through binary search trees) and priority queues (as implemented through binary search trees and binary heaps). These are data structures that can store multiple elements taken from an ordered domain and provide efficient operations such as insertion and deletion of elements and access to the minimum and maximum elements.

In the process you will work with the following signatures, modules, and functors (marked as to which we provide and which you write or complete [in italics]):

- Ordered collections (see `orderedcoll.ml`)
  - An ordered collection signature [provided]
  - *A functor for generating implementations of the signature based on binary search trees [you complete]*
  - A module generated using the functor implementing integer binary search trees [provided]
- Priority queues (see `prioqueue.ml`)
  - A priority queue signature [provided]
  - *A functor for generating implementations based on lists [you complete]*
  - *A functor for generating implementations based on binary search trees [you write, using the binary tree work above]*
  - *A functor for generating implementations based on binary heaps [you complete]*
  - Modules generated from all three functors implementing integer priority queues [provided]
  - Sort functions that use these modules [provided]

Questions to consider: The "questions to consider" in this writeup (as opposed to the numbered problems) are not graded. They are provided to help you check your understanding of the material. You are not expected to include answers to them in your submission, but they are things that could arise on exams. Answers to most of the questions are in the comments of the problem set.

Date: February 21, 2018.

Testing: Unit testing is required as usual. Please see `tests.ml` for further information on how to test.

Partners: You are allowed (and encouraged) to work with a partner on this assignment. You are allowed to work alone if that is your preference. If you are an extension student, you are free to pair up with another extension or on-campus student. For more detailed instructions, look here.

Compilation and Submitting: Note that for this problem set, the distribution code does not compile against the unit tests on Gradescope. Make sure to submit early and often.

Downloading: To download the problem set, first create your repository in GitHub Classroom for this homework by following this link. Then, follow the GitHub Classroom instructions found here. Note that, now that you are working with a partner, you should endeavor to keep up-to-date with your partner's code. Run `git pull` to fetch any changes, and `git push` to upload your changes to the repository. It's generally a good practice to `pull` before you `push`, so you can resolve any conflicts between your and your partner's code. More information about Git and its use can be found in Eddie Kohler's guide to Git and in TF Brian Yu's Git video.

1. Ordered collections

An ordered collection is a collection of elements that have an intrinsic ordering to them. Natural operations on ordered collections include insertion of an element, deletion of an element, searching for an element, and extracting the minimum and maximum elements.

To specify the ordering relation on the elements, we will make use of the following enumerated type, which is useful as the result of comparing two values:

```
type order = Equal | Less | Greater ;;
```

A simple module signature for ordered collections, called `ORDERED_COLLECTION_0`, specifies an interface with a data type for collections, as well as `empty`, `insert`, `search`, `delete`, `getmin`, and `getmax` operations. You'll want to read and understand it to familiarize yourself with the syntax for how to write module signatures.

```
module type ORDERED_COLLECTION_0 =
  sig
    exception Empty
    exception NotFound

    (* The type of collections. What this type actually looks like is
       left up to the implementation *)
    type 'a collection

    (* Return an empty collection *)
    val empty : 'a collection
    (* Insert elt into collection *)
```

```
  val insert : ('a -> 'a -> order) -> 'a -> 'a collection -> 'a collection
  (* Search a binary collection for the given value. *)
  val search : ('a -> 'a -> order) -> 'a -> 'a collection -> bool
  (* Delete the given value from a binary collection. *)
  val delete : ('a -> 'a -> order) -> 'a -> 'a collection
  (* Return the minimum value of a binary collection. May raise
     Empty exception. *)
  val getmin : ('a -> 'a -> order) -> 'a collection -> 'a
  (* Return the maximum value of a binary collection. May raise
     Empty exception. *)
  val getmax : ('a -> 'a -> order) -> 'a collection -> 'a
end ;;
```

Notice that it explicitly lists the types and values that any module implementing this interface must define, as well as the exceptions that any function in the interface may throw.[1] For a function like `getmin`, we could instead choose to return an `'a option`, which would avoid the need for an exception. But you should get used to exceptions like these in modules, since OCaml modules tend to use them. Remember, functions *are* values, so functions are also listed with the `val` keyword.

The interface for `ORDERED_COLLECTION_0` is not ideal. Consider the following questions:

- Is `ORDERED_COLLECTION_0` a type?
- How would one *use* `ORDERED_COLLECTION_0` ?
- Why do several of the functions require an argument of type `'a -> 'a -> order`?
- Why is `ORDERED_COLLECTION_0` not ideal?
- How could a call to `delete` give you incorrect behavior for a correctly constructed tree?

An improved signature `ORDERED_COLLECTION` is provided in the file `orderedcoll.ml`. To create this better interface, we need to introduce another module type – `COMPARABLE` – which has its own signature. Take a look at the `ORDERED_COLLECTION` signature, and consider these questions:

- Why is `ORDERED_COLLECTION` a better interface?
- Why did we need to introduce another module type `COMPARABLE` ?

2. Implementing ordered collections with binary search trees

A simple – but particularly inefficient – implementation of ordered collections is with an ordered list. Insertion places the element in the proper position; searching can stop once a larger element is found; the minimum element is the first in the list; the maximum, the last.

Binary search trees provide a much better implementation. These are binary trees that obey the following invariant: All elements that are equal in the ordering are stored at a single node in the tree (say, as a list). All elements in the left subtree of a node are less than those stored at the node, and all those in the right subtree are greater.

[1] Because of how OCaml handles exceptions, listing exceptions is optional, and you can't indicate with code which functions may cause which exceptions, but it is good style to mention in a function's comments what exceptions it may throw and under what conditions.

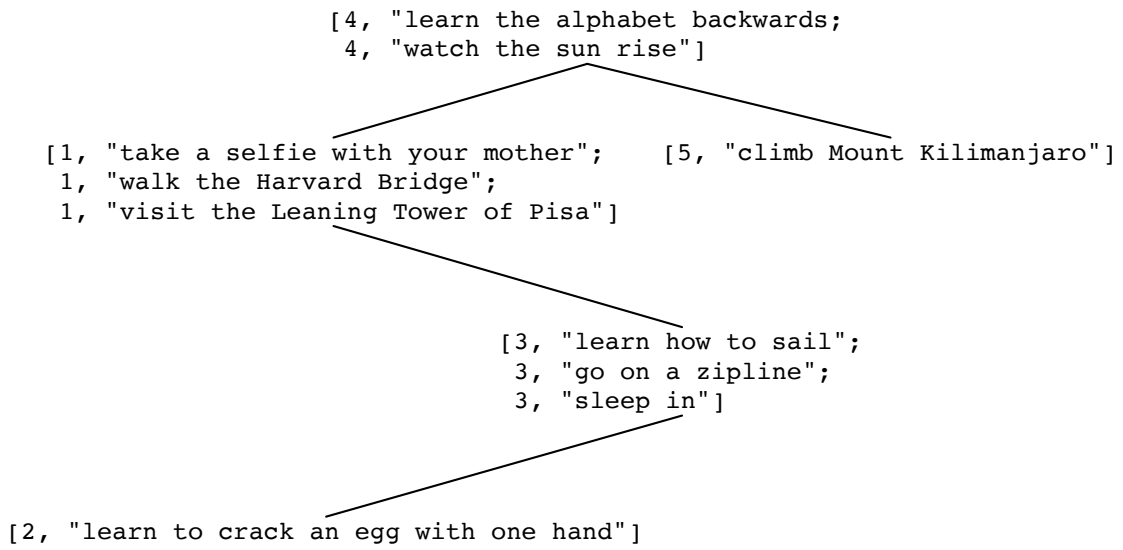

FIGURE 1. A sample binary search tree. The empty children have been elided.

For instance, consider the following set of elements of type `int * string`, gleaned from my personal bucket list:[2]

```
4, "learn the alphabet backwards"
1, "take a selfie with your mother"
5, "climb Mount Kilimanjaro"
1, "walk the Harvard Bridge"
3, "learn how to sail"
1, "visit the Leaning Tower of Pisa"
2, "learn how to crack an egg with one hand"
3, "go on a zipline"
3, "sleep in"
4, "watch the sun rise"
```

where the ordering on the elements is given by integer comparison of the first element of the pair. (Thus, `4, "learn the alphabet backwards"` and `4, "watch the sun rise"` would compare *equal*.)

We might store these elements in a binary search tree that looks like the one in Figure 1. Note how for each node, the elements in the left subtree precede and the right subtree follow in the ordering.

What happens when there are multiple elements that compare equal in the ordering over elements? There are multiple possibilities, but here we will take the approach (as we have in the

[2] I won't say which ones I've completed.

figure) of allowing for that case, and for the purpose of selecting among them (as in searching or deleting elements), choose the one that was inserted first.

You will provide a functor, called `BinSTree`, for generating *implementations* of the `ORDERED_COLLECTION` interface. The `BinSTree` functor implements a binary search tree as above where values that compare equal are compressed into a single node containing a list of those values. Remember that functors are not yet modules; they must be applied to an argument module in order to produce a module. In this case, `BinSTree` takes a module satisfying the `COMPARABLE` signature *as an argument* and returns an `ORDERED_COLLECTION` module. Once you have implemented `BinSTree`, you can create `IntTree` – a binary search tree of integers – by applying `BinSTree` to an integer implementation of `COMPARABLE`.

Problem 1. *Implement the* `insert`, `search`, `getmin`, *and* `getmax` *functions for* `BinSTree`. *(We've provided the rest.) Don't forget to test it well.* □

3. Priority queues

A priority queue is another data structure that can be considered a collection of ordered elements, but specialized for a simpler set of functionality. In particular, elements can be added and the minimum element extracted. That's all. Priority queues are widely useful, for instance, when implementing Dijkstra's algorithm for efficiently computing shortest paths in a network. We have provided the PRIOQUEUE interface for priority queues, which supports `empty`, `is_empty`, `add`, and `take` operations. The `add` function inserts an element into the priority queue and the `take` function removes the minimum element.

Because priority queues allow for only a subset of the operations of other ordered collections, they admit of more efficient specialized implementations. In this section you will be implementing priority queues in three ways – with lists, with binary search trees, and with binary heaps.

Problem 2. *Complete the* `ListQueue` *functor: a naive implementation of a priority queue. In this implementation, the elements in the queue are stored in a simple list in priority order. This implementation is not ideal because either the* `take` *or the* `add` *operation is* $O(n)$ *complexity.* □

Problem 3. *Implement* `TreeQueue`, *which is less naive than* `ListQueue` *(but still not ideal). In this implementation of the PRIOQUEUE interface, the queue is stored as a binary search tree using the BinSTree abstract data type that you've already implemented.* □

Consider these questions:

- Why is the `TreeQueue` implementation not ideal?
- What is the worst case complexity of `add` and `take` for a `TreeQuee`?

Finally, you will implement a priority queue using a *binary heap*, which has the attractive property of $O(log(n))$ complexity for both `add` and `take`. Binary (min)heaps are binary trees for which the following *ordering invariant* holds: the value stored at a node is smaller than all values stored in the subtrees below the node. (They thus have the attractive property that the minimum element is always stored at the root of the tree.) Binary heaps as you will implement them obey a further invariant of being *balanced*: For a given node, its left branch has either the same number or

one more node than its right branch.[3] To define the invariant, we will call a balanced tree *odd* or *even*. A tree is odd if its left child has one more node than its right child. A tree is even if its children are of equal size. All subtrees must be either odd or even. *The functions add and take must return balanced trees for this definition of 'balanced'*.

In the skeleton code for the BinaryHeap functor in `prioqueue.ml`, we have defined the `tree` type for implementing the binary heap. Functions over the type will often need to respect one of these representation invariants:

- *Weak invariant*: The tree is balanced. That is, for any given node in the tree, there are the same number of nodes in the left subtree as in the right subtree, or the left has exactly 1 more. (The tree cannot have more nodes on the right than the left.)
- *Strong invariant*: The tree satisfies the weak invariant, and for every node, the value stored at that node is less than or equal to the values stored at all of its child nodes.

The `add` and `take` functions must return trees that respect the strong invariant, and should assume they will only be passed trees that also obey the strong invariant. That is, they *preserve the strong invariant*. We have provided stubs for helper functions that operate on trees that are required to preserve only the weak invariant. Hint: Your nodes should track whether they are odd or even. This will help you keep your tree balanced at all times.

Notice that we have encoded the difference between odd and even nodes in the `tree` type that we've provided for `BinaryHeap`. You should probably first write a `size` function for your tree type. This will help you check your representation invariant. You should *not* be calling `size` in the implementation of `take`; rather, you should be using `size` to test `take`. We have provided you with the implementation of `add` and a partial implementation of `take`. Below are some guidelines when implementing `take` and its helper functions, as well as in understanding `add`.

3.1. **`add`.** The `add` function inserts a node into a spot that will either turn the main tree from odd to even or from even to odd. We implement this function for you, but you should understand how it works.

3.2. **`take`.** The `take` function removes the root of the tree (the minimum element) and replaces it by a leaf of the tree that, when removed, turns the tree from odd to even or from even to odd.

After removing and replacing the root node your tree will respect the weak invariant. You must "fix" the tree to respect the strong invariant, as depicted in Figure 2.

Some questions to consider:

- How do we know that our binary heap stays balanced?
- How might you test your binary heap?
- How might you test the helper functions used in implementing your binary heap?
- Why is it useful to use ListQueue, TreeQueue, and BinaryHeap behind a PRIOQUEUE interface?

[3] This definition of balance is a bit different from a traditional variant for balanced binary trees that requires the last (lowest) level in the tree to be filled strictly from left to right.

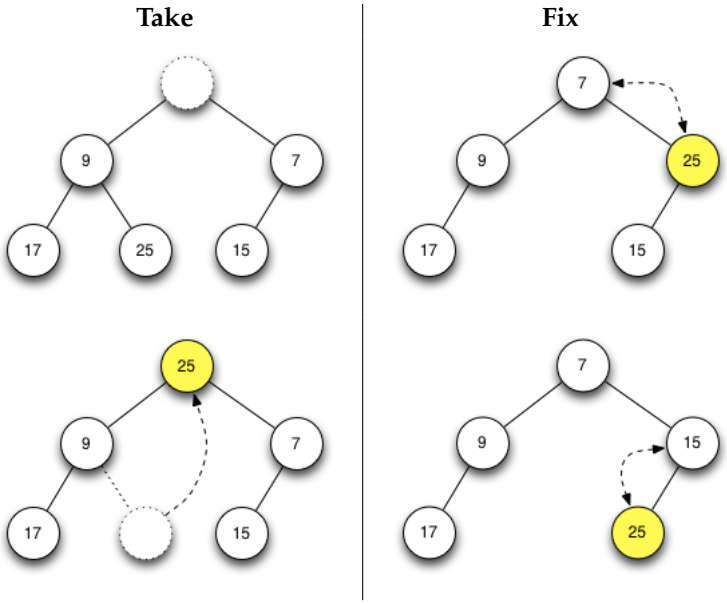

FIGURE 2. Visual depiction of fixing a tree to rebalance it after taking an element from it.

Problem 4. *Complete the implementation of the binary heap priority queue by providing definitions for* `get_top`, `fix`, `get_last`, *and* `run_tests`, *and completing the definition for* `take`. □

Now that you've provided three different implementations of priority queues, all satisfying the PRIOQUEUE interface, we give you an example of how to use them to implement sort functions. You should use these for testing (in addition to testing within the modules).

4. Challenge problem: Sort functor

Write a functor for sorting which takes a COMPARABLE module as an argument and provides a sort function. You should abide by the following interface:

```
type c
val sort : c list -> c list
```

You should use your BinaryHeap implementation, and test it.

5. Challenge problem: Benchmarking

Benchmark the running times of heapsort, treesort, and selectionsort. Arrive at an algorithmic complexity for each sorting algorithm. Record the results of your tests. Be convincing with your data and analysis when establishing the algorithmic complexity of each sort.

6. Submission

Before submitting, please estimate how much time you and your partner each spent on each section of the problem set by editing the line in each file that looks like

```
let minutes_spent_on_part () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) the part took you to complete. Make sure it still compiles.

Then, to submit the problem set, follow the Gradescope instructions found here.