Problem Set: Ordered collections

Stuart M. Shieber

March 7, 2023

In this assignment you will use modules to define several useful abstract data types (ADT). The particular ADTs 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).

An ordered collection is a collection of elements, where the elements have an intrinsic ordering to them. For instance, the elements might be integers ordered by the numerical order or strings ordered by alphabetical order.

Natural operations on ordered collections include insertion of an element, deletion of an element, searching for an element, and access to the minimum and maximum elements. Priority queues constitute a special case of ordered collection in which the only operations are insertion of an element and extraction of the minimum element.

The advantage of keeping a collection ordered, as opposed to a collection with unordered elements, is twofold. First, it allows operations that depend on that ordering, for instance, extracting the minimum element in the collection. Second, it allows for more efficient search for elements, for instance, by taking advantage of binary search.

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

- Orderings (see `order.ml`) – A variety of utilities concerning orderings, including
  - A signature for comparable elements [provided]
  - Modules satisfying the signature for integers and integer-string pairs [provided]
- 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 search tree functor 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]

1 *Signatures for ordered collections*

To specify the ordering relation on the elements of an ordered collection, we will make use of the following enumerated type (see the file `order.ml`), which is useful as the result of comparing two values:

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

A 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

    (* The empty collection *)
    val empty : 'a collection
    (* Inserts elt into collection *)
    val insert : ('a -> 'a -> order)
                  -> 'a -> 'a collection -> 'a collection
    (* Searches a collection for the given value. *)
    val search : ('a -> 'a -> order)
                  -> 'a -> 'a collection -> bool
    (* Deletes the given value from a collection. May
       raise NotFound exception *)
    val delete : ('a -> 'a -> order)
                  -> 'a -> 'a collection -> 'a collection
    (* Returns the minimum value of a collection. May
       raise Empty exception. *)
    val getmin : ('a -> 'a -> order) -> 'a collection -> 'a
    (* Returns the maximum value of a collection. May
      raise Empty exception. *)
    val getmax : ('a -> 'a -> order) -> 'a collection -> 'a
  end ;;
```

The signature explicitly lists the types and values that any module implementing this interface must define, as well as the exceptions that the implementation provides and that functions in the interface may raise.[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.

[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 raise and under what conditions.

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 might 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` (provided in the file `order.ml`). 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 *Binary search trees*

A simple – but particularly inefficient – implementation of ordered collections is with a sorted 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. (We'll look at efficiency issues in more detail later.)

Binary search trees provide a much better implementation. We introduced binary trees in Section 11.5. Here, we'll use a particular variant, the BINARY SEARCH TREE.

A binary search tree is a binary tree that obeys the following invariant:

For each node in a binary search tree, all values stored in its left subtree are less than the value stored at the node, and all values stored in its right subtree are greater than the values stored at the node.

What if there are multiple values in the tree, even distinct ones, that are equal in the ordering? We'll store all such elements together at a single node, say as a list.

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

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

```
4, "learn the alphabet backwards"
1, "take a selfie with my 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"
```

The integers can be interpreted as the priority I place on the activity. Or not. But in any case, let's take the ordering on the elements to be given simply 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.

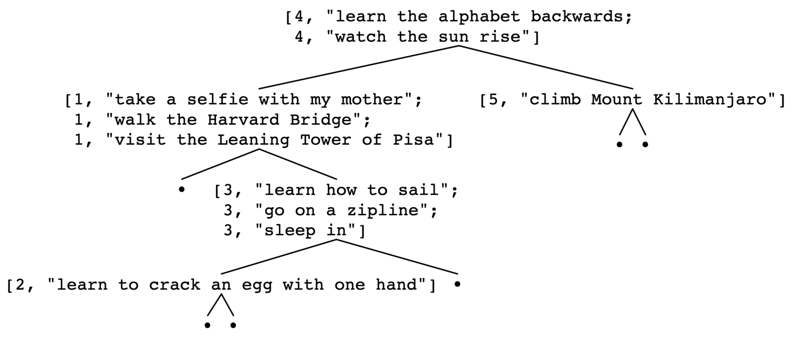
[4, "learn the alphabet backwards;
4, "watch the sun rise"]
[1, "take a selfie with my mother";
1, "walk the Harvard Bridge";
1, "visit the Leaning Tower of Pisa"]
[5, "climb Mount Kilimanjaro"]
[3, "learn how to sail";
3, "go on a zipline";
3, "sleep in"]
[2, "learn to crack an egg with one hand"]

Figure 1: A sample binary search tree. The small gray circles indicate leaves of the tree.

What happens when there are multiple elements that compare equal in the ordering over elements? There are multiple possibilities, but here we take the approach (as we have in the 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. (See Chapter 14.)

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` functor 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 `TreeQueue`?

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 a kind of balanced binary tree for which the following *ordering invariant* holds:

Ordering invariant: The value stored at each node is smaller than all values stored in the subtrees below the node.

Such trees thus have the attractive property that the minimum element is always stored at the root of the tree.

In the skeleton code for the `BinaryHeap` functor in `prioqueue.ml`, we have defined the `tree` type for implementing the binary heap, which provides further clarification:

```
type tree =
  | Leaf of elt
  | OneBranch of elt * elt
  | TwoBranch of balance * elt * tree * tree
```

The `trees` for use in forming binary heaps are of three sorts, corresponding to the three variants in the type definition:

- Leaves store a single value of type `elt` (like the 17 node in Figure 2).
- Along the bottom edge of the tree, a tree with a single child, each storing a value of type `elt`, can appear (like the 9—17 branch in Figure 2).
- Finally, regular nodes of the tree store a value of type `elt` and have two subtrees, a left and right subtree. (See for example, the node with value 3 in Figure 2.) The node also stores its "balance" as described below.

Binary heaps as you will implement them obey a further invariant of being *balanced*:[3]

[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.

Balance invariant: For each `TwoBranch` node, its left branch has either the same number or one more node than its right branch.

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. The invariant says then that all subtrees must be either odd or even.

Functions over the type will often need to respect combinations of the ordering invariant and the balance invariant:

Weak invariant: The tree is balanced.

Strong invariant: The tree is balanced and ordered.

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.

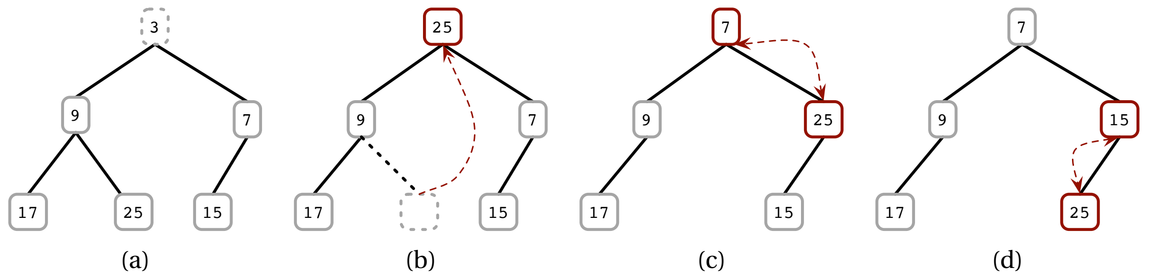

Figure 2: Visual depiction of fixing a binary heap to rebalance it after taking an element from it. Starting with a heap satisfying the strong invariant (a), the minimum element (3) is at the root of the tree. To "take" the minimum element, the root node is replaced (b) with the node with value 25, turning the tree from odd to even. The tree is then "fixed" by swapping nodes from the root down the tree (c) until the value at the root has found its appropriate location (d). Of course, in the implementation there is no actual changing of trees – no mutable state. Rather, new tree nodes are made where necessary that

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?

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 problems*

A reminder: Challenge problems are for your karmic edification only. You should feel free to do these after you've done your best work on the rest of the problem set.

4.1 *A 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.

4.2 *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.

Version information: Commit c650429 from 2023-03-07 by Stuart Shieber. CI build of I_TAG ().