PSET5: Ordered Collections
Priority Queues

Anusha Murali

March 17, 2023

Tasks to do

Part II: Implement Ordered Collections with Priority Queues

1. Complete ListQueue: elements are stored a list
2. Complete TreeQueue: elements are stored in a BST
3. Complete HeapQueue: elements are stored in a balanced binary tree

HeapQueue

Task: Use a Binary Heap to implement the signature of PRIOQUEUE .

Step 1 Load Files

Load Files

1. # #use "order.ml"
2. # #use "orderedcoll.ml"
3. # #use "prioqueue.ml"

Step 2: Complete the BinaryHeap functor

Complete get_top

```
let get_top (t : tree) : elt =
  failwith ”BinaryHeap get_top not implemented”
```

get_top

```
let get_top (t : tree) : elt =
    match t with
    | Leaf e -> e
    | OneBranch (e, _) -> e
    | TwoBranch (_, e, _, _) -> e
```

Step 2: Complete the BinaryHeap functor

Complete fix

```
let fix (t : tree) : tree =
    failwith "BinaryHeap fix not implemented"
```

fix_top: A helper function, which changes the top element of t with e

```
let fix_top (e : elt) (t : tree) : tree =
   match t with
   | Leaf _ -> Leaf e
   | OneBranch (_, e2) -> OneBranch(e, e2)
   | TwoBranch (x, _, t1, t2) -> TwoBranch(x, e, t1, t2)
```

Step 2: Complete the BinaryHeap functor

fix

```
let rec fix (t : tree) : tree =
     match t with
     | Leaf _ -> t
     | OneBranch(e1, e2) ->
       (match Elt.compare e1 e2 with
        | Less -> t
        | Equal
        | Greater -> OneBranch(e2, e1) )
     | TwoBranch(x, e, t1, t2) ->
       (* Get the top elements of t1 and t2 and determine which is smaller *)
       let (e1, e2) = (get_top t1, get_top t2) in
       (match Elt.compare e1 e2 with
        | Less
        | Equal ->  (match Elt.compare e e1 with
                     | Less -> t
                     | Equal
                     | Greater ->
                       TwoBranch(x, e1, fix (fix_top e t1), t2))
        | Greater -> (match Elt.compare e e2 with
                      | Less -> t
                      | Equal
                      | Greater -> TwoBranch(x, e2, t1, fix (fix_top e t2))))
```

Step 2: Complete the BinaryHeap functor

Complete get_last

let get_last (t : tree) : elt * queue =
failwith ”BinaryHeap get_last not implemented”

get_last

```
let rec get_last (t : tree) : elt * queue =
     match t with
     | Leaf e -> (e, Empty)
     | OneBranch (e1, e2) -> (e2, Tree (Leaf e1))
     | TwoBranch (even_or_odd, e, t1, t2) ->
       (match even_or_odd with
        | Odd -> (match t1 with
                  | Leaf last -> (last, Tree (OneBranch (e, get_top t2)))
                  | _ -> (fst (get_last t1),
                          Tree (TwoBranch
                                (Even, e, (extract_tree (snd (get_last t1))),
                                           t2))))
        | Even -> (match t2 with
                  | Leaf last -> (last, Tree (OneBranch(e, get_top t1)))
                  | _ -> (fst (get_last t2),
                          Tree (TwoBranch
                               (Odd, e, t1,
                                          (extract_tree (snd (get_last t2))))))))
```

Step 2: Complete the BinaryHeap functor

Re-load prioqueue.ml (with your completed functions)

1. # #use "prioqueue.ml"

Example

Now the examples in the next slides should work

Test add function

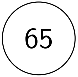
Example

1. First create an empty HeapQueue called myQ

   ```
   # let myQ = IntHeapQueue.empty;;
   ```

2. Insert 65

   ```
   # let myQ = IntHeapQueue.add 65 myQ;;
   ```



Example

Print the queue:

```
# IntHeapQueue.to_string myQ;;
```

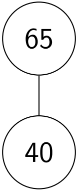
Output: - : string = "Leaf 65"

Test add function

Example

1. Now insert 40

   # **let myQ = IntHeapQueue.add 40 myQ**;;



Example

Print the queue:

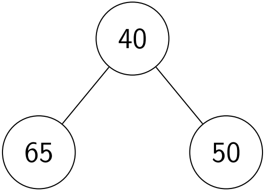
# **IntHeapQueue.to_string myQ;;**

Output: - : string = "OneBranch (40, 65)"

Test add function

Example

1. Now insert 50
   # **let myQ = IntHeapQueue.add 50 myQ**;;



Example

Print the queue:

# **IntHeapQueue.to_string myQ;;**

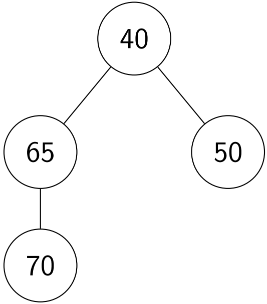
Output: - : string = "TwoBranch (Even, 40, Leaf 65, Leaf 50)"

Test add function

Example

1. Now insert 70

   # **let myQ = IntHeapQueue.add 70 myQ**;;

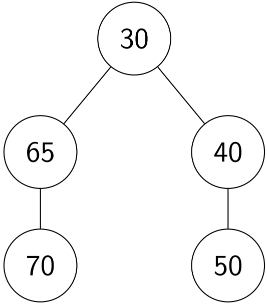


"TwoBranch (Odd, 40, OneBranch (65, 70), Leaf 50)"

Test add function

Example

1. Now insert 30
   # **let myQ = IntHeapQueue.add 30 myQ**;;

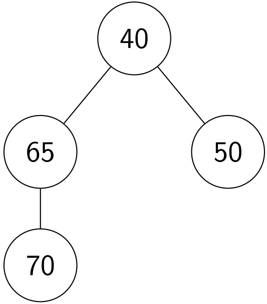


"TwoBranch (Even, 30, OneBranch (65, 70), OneBranch (40, 50))"

Test take function

Example

1. The take function returns the element with the highest priority (i.e: smallest value) and the remaining queue
   # **let (hiPri, myQ) = IntHeapQueue.take myQ**;;
2. The value of **hiPri = 30** and the new **myQ** is shown below



Important comment on HeapQueue

Average and worst-case time complexity

1. The Heap Queue is always (almost) balanced. So, the height of the tree is $\log n$. Hence the average time complexity and the worst-case time complexity to search an element are both equal to $O(\log n)$