Contents

1 Heap Implementation ........................................ 1
  1.1 Insert .................................................. 2
  1.2 Delete ................................................ 3

2 Priority Queue Implementation ............................. 5

Objectives

By the end of this lecture, you will be able to:

- implement a heap
- implement a priority queue using a heap
- use a heap to sort a list

1 Heap Implementation

We are now ready to work on an implementation of heaps. We start by declaring an `AbsHeap` class with constructor parameter `initCapacity` (i.e., initial capacity), which specifies the size of the underlying array. We create a mutable variable `capacity`, whose value is initialized to `initCapacity`. You’ll see how to dynamically adjust the size of the heap array in a future lecture, which is why we’ve included this additional variable. For now, `capacity` will never change.)

```scala
class AbsHeap[T : ClassTag](val initCapacity: Int) extends IHeap[T]
  protected var capacity = initCapacity
```

Additionally, we declare the attributes `heap` and `numItems`, as follows:

```scala
protected var heap = Array.fill[Option[T]](capacity)(None)
protected var numItems = 0
```

With this infrastructure in place, we can immediately write simple methods, like `getMin`:

```scala
override def getMin: Option[T] = heap(0)
```
In this implementation, we have chosen to implement heaps using arrays (of options). Consequently, the root node—the element with the minimum (i.e., highest) priority—is stored at index 0.

It is more challenging to write methods that insert and delete in and out of a heap. The difficulty stems from the fact that we must be sure to maintain our invariants. That is, we must make sure that after each operation, our tree is 1. balanced, and 2. in heap order. These two operations are the next topic in this lecture.

1.1 Insert

Let’s start with insert. As a first attempt, we might try to search the tree for the correct location at which to insert the new element. This is how we proceeded when we implemented BSTs. But we ran into a problem there, and if we pursue this approach, we would run into the same problem here: balance! That is, in addition to maintaining our invariant, we would also like for our trees/heaps to be balanced, so that our operations will be $O(\log n)$ instead of $O(n)$.

We are going to discuss an approach that focuses on balance first, and maintaining heap order second. Here’s the idea: maintain a compact array representation of our heap (i.e., use indices 0 to $n - 1$ inclusive to store a heap of size $n$), and then be sure to always insert a new node at the next available index in our array (i.e., use breadth-first ordering). This approach ensures balance. But after inserting in this way, how will we ever reinstate the invariant?

Here’s the answer: use a promotion strategy. Think of the heap as an organizational hierarchy in a large corporation. Place the new hire at the bottom of the hierarchy and then let her compete to move up the ranks. Each time she moves up, her former boss moves into her previous position. The new hire moves up the hierarchy until she is no longer competent to advance any further.

Why is this strategy—insert at slot $n$ and promote—correct? Because given a heap, it returns a heap: i.e., it maintains the heap invariant. That is, even if heap-order violations are introduced along the way, heap-order is eventually restored. (A heap-order violation, in a min-heap, is a situation in which a child’s value is less than that of its parent.)

To show that this is necessarily the case, we need to argue 3 points:

1. Given a heap to start with, the only possible heap-order violation is the one that results from inserting the new node.

2. No further heap-order violations can be introduced while promoting except those that involve the new node (though the existing one can certainly be removed). In other words, each time the new node is swapped with its parent, the new node remains the only node that is possibly out of heap order.

3. The promotion strategy eventually terminates having removed the possible heap-order violation that results from inserting the new node.

Given a heap to start with, it is clear that the only possible heap-order violation is the one that results from inserting the new node.

More interestingly, let’s argue that the promotion strategy does not introduce any further heap-order violations. The reason for this is because a parent node’s priority should be less than that of its children. So, if the new node is swapped with its parent, this means that new node’s priority is less
than that of (what was) its parent. But then it is also necessarily less than that of its sibling, since its sibling’s priority is greater than that of their parent.

Finally, why does the promotion strategy eventually remove the possible heap-order violation that results from inserting the new node? This is because the promotion strategy is guaranteed to terminate (at the root of the tree, if not before). If it terminates before the root, that is because the new node’s value is greater than that of its parent. So the heap-order violation was removed. If it terminates at the root, then the heap-order violation was again removed, because it has no parents, so its value is not less than those of its parents.

The above argument establishes the correctness of the “insert at slot $n$ and promote” strategy. It also establishes its run time, $O(\log n)$. The reason for this is that the argument shows that it is only ever necessary to traverse one path through the heap while promoting, and the length of each path is $O(\log n)$—remember, balance was our goal, first and foremost!

Here is code for the `insert` method. The promotion strategy is not yet implemented (see a future homework!) but is referred to by the method `siftUp`, which takes as input the index of the node to be promoted (i.e., the node that is possibly out of heap order), and promotes that node until heap order is restored.

You may wonder: what happens if we try to insert more elements into the heap than we have space in the array? We’ll talk about how to deal smoothly with this situation in a future lecture. For now, we’ll just focus on getting the usual case—where there is space remaining for the new element—right.

```scala
override def insert(item: T) {  
  // Insert new item into heap  
  heap(numItems) = Some(item)  
  // Restore heap order  
  siftUp(numItems)  
  // Increment numItems  
  numItems += 1
}
```

### 1.2 Delete

Now we are ready to delete the minimum element from our priority queue. We know precisely where this element is. It is at the root, which is stored at index 0 in our array. But if we simply delete this element, we are left with a hole at the root of our tree. What node should we replace this node with? Should we let the past CEO’s two underlings compete for this position, and then recursively fill the hole that is left after this competition is resolved. No, because our first and foremost goal is balance, and repeated applications of this process could inadvertently result in an unbalanced tree/heap.

Keeping in mind our first and foremost goal of balance, we will instead replace the root node with the node stored at the last occupied index of our array. This approach ensures balance. But then, as before, we must reinstate our invariant. This time, we are not in a situation where someone at the bottom of the hierarchy may very well deserve a promotion; on the contrary, someone at the top of the hierarchy very likely deserves a demotion!
The demotion strategy is not as intuitive as the promotion strategy was in the organizational hierarchy example. Nonetheless, it can be understood as replacing the CEO with someone from the mail room, and then demoting that someone again and again until he ends up where he belongs (quite possibly, back in the mail room).

Implementing the demotion strategy is slightly trickier than implementing the promotion strategy, only because we have to decide whether to swap a parent node with its left or right child. We can only swap with one of them. Which one makes sense?

The answer is the smaller of the two, because if we were to swap with the larger of the two, then the two children (one of which is now a parent) would be out of heap order.

For example, consider the following min-heap, which is out of currently out of min-heap order:

```
5
4 3
```

If we swap 4 and 5, then the resulting tree is still out of min-heap order:

```
4
5 3
```

But if we swap 3 and 5 (3 < 4), then the resulting tree is in min-heap order:

```
3
4 5
```

In summary, since we want to reinstate min-heap order, we must promote the smaller of the two.

Like the promotion strategy, the demotion strategy is $O(\log n)$. Again, the reason for this is that it is only necessary to traverse one path through the heap, and the length of each path is $O(\log n)$.

Here is code for the `delete` method. The demotion strategy is not yet implemented (see a future homework!), but is referred to by the method `siftDown`, which takes as input the index of the node to be demoted: i.e., the node that is possibly out of heap order.

```scala
override def deleteMin: Option[T] = {
  if (isEmpty) None
  else {
    val toReturn = heap(0)
    // Perform demotion strategy here
    return toReturn
  }
}
```
2 Priority Queue Implementation

Equipped with an implementation of heaps, we can use them to complete our priority queue implementation. The first thing we need is a `PQPair` class that stores items and their priorities. We declare this class as a case class, so that we get useful methods like `toString` and `equals`, for free:

```scala
case class PQPair[T, S <% Ordered[S]](val item: T, val priority: S)
  extends Ordered[PQPair[T, S]]
```

Furthermore, the `PQPair` class extends `Ordered`. As such, it must implement a `compare` method. This is easy: to compare pairs, we simply compare their priorities, and as priorities implement `Ordered`, we can use the `compare` method on priorities to compare pairs.

```scala
override def compare(that: PQPair[T, S]) = this.priority compare that.priority
```

Given this `PQPair` class, we can now define a `PriorityQueue` class that extends `IPriorityQueue`. Inside this class, we declare an attribute `heap`, a `Heap` of `PQPair`s.

```scala
abstract class AbsPriorityQueue[T: ClassTag, S <% Ordered[S]: ClassTag](val initCapacity: Int = 20)
  extends IPriorityQueue[T, S] {

  import IPriorityQueue.PQPair

  protected var capacity = initCapacity
  protected val heap = new Heap[PQPair[T, S]](capacity)
}
```

With this infrastructure in place, it is now entirely straightforward to implement priority queues:

```scala
override def getMinItem = heap.getMin match {
  case Some(PQPair(u, v)) => Some(u)
  case None => None
```
Sorting via Heaps

One famous application of priority queues (and thus, heaps) is sorting. Suppose that we have a list of objects that extend `Ordered`, and we want to sort that list. What’s a straightforward way to perform the sort? Just pass over the list, putting everything in it into a priority queue. And then pull them out again, putting them back into a list. Because the priority queue handles the work of *ordering the elements*, the second list will magically\[1\] be sorted.

\[1\]Not really! We did all the work in the heap implementation.