Lecture 23: Implementing Heaps

11:00 AM, Mar 15, 2021

Contents

Objectives

By the end of these notes, you will know:

- the challenge to implementing heaps with classes as we did for binary trees
- how to store binary trees in arrays

1 Implementing Heaps Through Binary Trees

Our goal is to implement heaps via the following interface:

```
trait IHeap {
   def getMax: Option[Int]
   def insert(newElt: Int): Unit
   def deleteMax(): Option[Int]
}
```

Our implementation will build on classes and traits for binary trees:

```
trait IBinTree {}
case object EmptyBT extends IBinTree
case class NonEmptyBT(
                data: Int,
                left: IBinTree,
                right: IBinTree) extends IBinTree
```

Design question: which of the following three approaches we should use to connect the `BinTree` classes and the `IHeap` interface?

```
// OPTION 1: put the heap/tree relationship in the interface
trait IHeap extends IBinTree {
   def getMax: Option[Int]
   def insert(newElt: Int): Unit
   def deleteMax(): Option[Int]
}
```

```
// OPTION 2: put the heap/tree relationship on the (eventual) Heap class
class Heap extends IBinTree with IHeap { ... }
```

```
// OPTION 3: put the tree within the (eventual) Heap class
class Heap extends IHeap {
   var theHeap: IBinTree = EmptyBT
   ...
}
```

Option 1 would require heaps to also implement all methods that ended up in the `IBinTree` trait. Given that some `IBinTree` methods might not make sense for heaps (such as a method to balance a tree by shifting around nodes), this could leak irrelevant content into the heap implementation.

Option 2 has the same problem as Option 1.

Option 3 is the way to go. The core data structure is a binary tree, but the methods in a heap are different from those in a binary tree.

From the type of the `insert` method, we see that we are implementing a mutable Heap class. What might this look like? Here's an initial version, including the `getMax` method:

```
class Heap extends IHeap {
    var theHeap: IBinTree = EmptyBT

    def getMax() = theHeap match {
        case EmptyBT => None
        case NonEmptyBT(data, left, right) => Some(data)
    }

    def insert(newElt: Int):Unit =
        print("not implemented yet")

    def deleteMax() = None // also not implemented
}
```

Step back and think for a moment – how will we implement `insert` and `deleteMax`? At least with the algorithms we sketched on the board last lecture, we need a way to track the lowest level nodes. We could agree to always insert elements in the leftmost available position, so we'd only need to track one node. We could certainly track this in a variable, such as:

```
     var lastInserted: Option[IBinTree]
```

Assume we had been maintaining this variable, and following a discipline of always inserting and deleting at the last node in the lowest level. How would we update it on an insert? Let's say we had the following tree:

```
          9
        /   \
       8     7
      / \   /
     1   5 4
```

The "last inserted" would be the node containing 4. To insert the new node, we would need to go up to the parent and insert the new node to the right of the 7 (and then swap it upwards into place). That node would become the new `lastInserted` node. But doing this operation implies that we need doubly-linked trees, rather than our current singly-linked ones.

Even worse, imagine that the last node we had inserted had been the 5:

```
          9
        /   \
       8     7
      / \
     1   5
```

Now we need to go up two levels (to the 9), so we can come down a level (to the 7) and insert in the left child. And then what if the last inserted node completes a row?

This is seeming a bit complicated (a good indication to step back rather than plow ahead). And all of these extra parent references effectively double the space needed to store the tree. Can we do better?

2 Arrays to the Rescue

Arrays provide a space efficient implementation of doubly-linked binary trees. The key insight is that *we can map the nodes of a binary tree to indices in a way that lets us calculate the indices of parents and children with simple formulas.* Consider the following labeling (where the numbers in square brackets are array indices):

```
         9 [0]
        /  \
   [1] 8    7 [2]
      / \
  [3] 1   5 [4]
```

If we label tree nodes from the top to bottom layer, from left to right, then we can leverage multiples of 2 to "navigate" the tree. Specifically, for the node at index i:

- The left child is in index $2i + 1$
- The right child is in index $2i + 2$
- The parent is in $\lfloor((i - 1)/2)\rfloor$ (the outer brackets mean floor/round down)

More generally, we can track the "last inserted" point as an index into the array. We insert a new element into the next array position. When deleting, we move the element in the last position to the root. Swapping can use our formulas to easily access parent and children nodes. This is a lot cleaner than what we were trying to do with class-based trees.

With this approach, there are never any empty spaces in the tree. We completely fill each level from left to right before adding a node at the next level. The heap is therefore balanced by construction, so we don't need to do anything special to balance it later. The heap invariant, in which the only constraint on how nodes are ordered lies in maximizing a local root, makes this approach possible.

2.1 Outline of a Class

More concretely, what might the class for an array-based implementation look like? Here's the outline of a `Heap` class based on arrays, with the swapping part left to a separate method for now:

```
class Heap[T](val initCapacity: Int)
             (implicit eltType: T => Ordered[T], classTag: (ClassTag)[T])
              extends IHeap[T] {
  private var heap = Array.fill[Option[T]](initCapacity)(None)
  private var numItems = 0

  // gets the elt in index 0 of the array
  override def getMax: Option[T] = heap(0)

  // moves the elt in the given index into the right position in the heap
  private def siftUp(fromIndex : Int): Unit = {
    println("Siftup called but not implemented yet")
  }

  // adds elt to the end of the heap, then move it into place
  override def insert(item: T): Unit = {
    heap(numItems) = Some(item)

    // Restore heap order
    siftUp(numItems)

    // Increment numItems
    numItems += 1
  }
}
```

The part of the class definition that reads `(implicit ...)` defines two additional parameters for the class that Scala fills in automatically (hence `implicit`). The `eltType` parameter requires that the generic type implement the `Ordered` trait (which is like the `Comparable` interface from Java). The `classTag` parameter enables the generic type `T` to be used to define an array. **You do not need to understand anything in that `implicit` line for CS18; we will give it to you if you would need it on an assignment.**

3 Study Questions

Think about these for practice.

- Draw the arrays (on paper) that would capture each of the following two binary trees:

```
     9
   /   \
  8     7
 / \   /
1   5 4

     8
   /   \
  3     6
 /       \
2         5
```

- Are both of these examples heaps?
- Considering the two pictures that you just drew, why do we insist on inserting new elements in the "next" position of the tree, rather than inserting them at any leaf position (which would still give us a heap according to the definition).
- Imagine that you were asked to build a heap of N numbers using the array-based class for the implementation. How many slots in memory would you use, if each number and reference to another item in memory takes one slot?
- Why is it harder to implement a heap using classes for nodes and leaves than it was to implement a basic binary tree?
- Assume you wanted to implement a binary search tree instead of a heap. Could you still store the node values in an array? What aspects of using an array to store heaps would change if we were using arrays to store binary search trees?

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.