Lecture 24: Implementing Heaps

10:00 AM, Mar 21, 2018

Contents

1 Recap on Heaps 1
  1.1 The Heap Interface .............................................. 1

2 Using Class-Based Binary Trees 2
  2.1 An Aside: Case Classes ........................................... 2
  2.2 A Heap Using Case-Class Trees .................................. 3

3 Arrays to the Rescue 4
  3.1 Outline of a Class .................................................. 5

4 Back to the Security Monitor 5
  4.1 Priority Queues .................................................... 5
  4.2 Putting Alerts into a Priority Queue ............................. 6

Objectives

By the end of these notes, you will know:

• How to implement a Heap
• How to use arrays to implement doubly-linked binary trees

In this lecture, we will practice

• Setting up classes to implement data structures

1 Recap on Heaps

Last class, we learned about heaps, which are a special class of binary trees that are designed to provide constant-time access to the max (or min) element in a set, while also supporting logarithmic time insertion and deletion operations.

• Heaps give constant-time access to the max element by keeping that element in the root of the tree.
• The key to achieving log-time insertion and deletion is to make sure that the tree is always balanced (has no nodes with height differentials more than 1) – this makes sure the leaves of the tree can be reached in log-time. Given that heaps are agnostic to which subtree a smaller value ends up in, we saw how one could perform at most \( \log(n) \) swaps to restore the heap property during insertion and deletion (where \( n \) is the number of elements).

It’s all well and good as drawing on the board, but how do we implement this in a way that actually gets these time benefits? That’s the topic of today’s lecture.

1.1 The Heap Interface

Our goal for these notes is to implement the following interface:

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

2 Using Class-Based Binary Trees

Since Heaps are a form of binary trees, we should be able to implement heaps using a variation on the binary trees that we implemented earlier this semester in Java. What does a basic binary tree data structure look like in Scala?

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

2.1 An Aside: Case Classes

This definition uses a new feature of Scala called a case classes. Case classes are useful for implementing recursive datatypes with different cases (like binary trees, or programming language syntax trees). In particular, they have several features:

• constructor params are val by default (contents are immutable)

• constructors can be used without new

• equals methods are provided automatically that compare the internal fields, rather than just the outermost memory addresses
Case classes are designed to be used with match expressions, as we will see shortly.

Here’s an example that illustrates the use of case classes and the automatically-created `equals` method:

```scala
object BTExamples {
  val bt1 = NonEmptyBT(
    5,
    NonEmptyBT(4, EmptyBT, EmptyBT),
    EmptyBT)
  val bt2 = NonEmptyBT(
    5,
    NonEmptyBT(4, EmptyBT, EmptyBT),
    EmptyBT)
}
```

At the console, if we run `BTExamples.bt1.equals(BTExamples.bt2)`, we get `true`. In contrast, if we tried this with a non-case class and we had not written a custom `equals` method, we would get `false`.

### 2.2 A Heap Using Case-Class Trees

Now let’s implement a heap using a binary tree. Here are three proposals of how to set this up via classes and traits. Which do you prefer and why?

```scala
// 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]
}
```

```scala
// put the heap/tree relationship on the class
class Heap extends IBinTree with IHeap {
  ...
}
```

```scala
// put the tree within the class
class Heap extends IHeap {  
  var theHeap: IBinTree = EmptyBT  
  ...
}
```

The third option is the way to go. The first two require the heap to implement any operations that end up in the `IBinTree` interface. Since binary trees support more than heaps, this would require adding irrelevant methods to the heap.

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:

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

```scala
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?
3 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]
/ \
```

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.

3.1 Outline of a Class

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

```scala
class Heap[T <% Ordered[T]: ClassTag](val initCapacity: Int) extends IHeap[T] {

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

  override def getMax = heap(0)

  override def insert(item: T) {
    heap(numItems) = Some(item)
    // Restore heap order
    siftUp(numItems)
    // Increment numItems
    numItems += 1
  }
}
```
The part about `Ordered` in the top line requires that there be a way to determine which of two heap elements is “larger” than the other (straightforward for numbers, but we’d have to define this on alerts). The `ClassTag` is something you just have to write in for now (it has to do with how Scala resolves type parameters through traits).

### 4 Back to the Security Monitor

#### 4.1 Priority Queues

Managing alerts calls for a data structure called a **priority queue**. A regular queue is a data structure in which elements are retrieved in the order that they were inserted (think of a checkout line in a supermarket – people pay in the order that they entered the line). In a priority queue, the item with highest priority item takes precedence, regardless of when it was inserted (think hospital triage).

Heaps are the most common data structure used to implement priority queues, but there are other options (such as sorted lists, or specialized data structures for specific heap contents).

In general, priority queues are the data structure that gets mentioned for applications like the security monitor, not heaps.

#### 4.2 Putting Alerts into a Priority Queue

Putting items in order within a priority queue (or heap) requires Scala knowing when one item is “larger” than another. This is built-in for types like Int and String. But what about alerts, for which there is no default notion of “greater than”?

If you have a class that you want to use in orderable contexts (like sorting), you have the class implement a trait called `Ordered`, which in turns requires a method `compare` to indicate when one item is larger than another. Here’s how we would add ordering to the `Alert` class:

```scala
class Alert(
  val username: String,
  val descr: String,
  val priority: Int) extends Ordered[Alert] {

  // method required by the Ordered trait
  def compare(that: Alert) = {
    if (this.priority == that.priority) 0
    else if (this.priority > that.priority) 1
    else -1
  }
}
```

Now, if we had two alerts, we can use `<` and `>` to compare them:

```scala
scala> val a1 = new Alert("Kathi", "login", 7)
scala> val a2 = new Alert("David", "saving", 5)
```
scala> a1 < a2
res2: Boolean = false

scala> a1 > a2
res3: Boolean = true