Lecture 25: What are Heaps Used For?
11:00 AM, Apr 3, 2020

Contents

1 Review of Class- and Array-Based Heaps 1
   1.1 Memory Layouts .......................................................... 1
   1.2 Insert and Delete on Arrays ......................................... 2

2 Heaps Implement Priority Queues 2
   2.1 Datatypes vs Data Structures ......................................... 3
   2.2 Putting Alerts into a Priority Queue ............................... 3

3 Heapsort 4

4 Programming Practice (Optional) 4
   4.1 Implement Heapsort ...................................................... 4

5 Study Questions 4

Objectives

By the end of these notes, you will know:

- Two common uses of heaps
- The Priority Queue datatype (which can be implemented with heaps)
- The difference between data structures and datatypes
- How to use heaps for sorting

1 Review of Class- and Array-Based Heaps

We started with a review of how class and array based heaps are laid out in memory, and a review of how insert and delete work when trees are represented as arrays.

1.1 Memory Layouts

These diagrams show the same heap as laid out in memory in each of the class and array based implementations. The array layout has an extra number (13) showing where an insert to the heap
would land in the array.

![Heap Diagram]

The class-based layout has a single EmptyBT object because the case-class definition we used for heaps defined EmptyBT as an object. In Scala, declaring something as object means that only one object from that class gets created.

### 1.2 Insert and Delete on Arrays

When we insert a new element into the array-based version, it goes in the next empty spot in the array. We then have to shift the new element up the tree until it lands in place. Consider the new 13 added in the array-based diagram. Last class, we talked about the formulas that compute the indices of parents and children of a given index. Using those formulas, we would check the 13 (index 6) against the item in its parent index (index 2). Since 13 is larger than 5, we would swap elements 13 and 5 in the array, then check 13 against its new parent (in index 1).

Recalling the notes on deletion: when we delete the top element, we move the last element into index 0 (replacing the removed/max element), then shift it down into place. The same formulas are used to navigate the array. The swapping mechanism is described in our deletion notes from two lectures ago.

### 2 Heaps Implement Priority Queues

For our security monitoring example, we needed to be able to quickly retrieve the most urgent alert, while also supporting frequent addition and deletion of alerts. We showed how heaps support this usage pattern efficiently in terms of time, and that array-based heaps were also efficient in terms of space.

Usually, when we talk about managing information according to priorities, we use a datatype called a priority queue. A regular queue is a datatype 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 or security alerts).

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).

### 2.1 Datatypes vs Data Structures

Note here that we are referring to heap as a data structure and priority queue as a datatype. Datatypes have a specific set of operations and can be implemented many different ways (with different time or space efficiencies). In this sense, they are like interfaces or traits. Data structures, in contrast, fix a structure of the data. For heaps, the structure is a binary tree with a particular organization on the nodes. While you might implement that structure in different ways (classes or arrays, as we saw last class), in both cases the structure is a binary tree with a specific item at the root.

### 2.2 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”?

In the prep work for this lecture, we asked you to think about what modifications the Alert class would need in order to be inserted into a heap. The answer is that the Alert class needs to implement the Ordered trait, which would allow two alerts to be compared. In Scala, the Ordered trait 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(
  private val username: String,
  private val descr: String,
  private 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 could use `<` and `>` to compare them:

```
scala> val a1 = new Alert("Kathi", "login", 7)
scala> val a2 = new Alert("David", "saving", 5)
scala> a1 < a2
res2: Boolean = false
```
That is the same comparison that would get used inside of the `siftUp` method in the `Heap` class implementation that we started in class.

Priority Queues are a sufficiently common data structure that they are provided as a built-in in many languages, including Scala. (docs for built-in priority queue:)

### 3 Heapsort

Heaps are also used for sorting elements (the algorithm is called `heapsort`). Conceptually, given a collection of items, you insert them into a heap one at a time, then pull the max element out into a list until the heap is empty.

If you want programming practice, you can implement heapsort (see Section 4).

### 4 Programming Practice (Optional)

#### 4.1 Implement Heapsort

Fill in the following class with an implementation of heapsort. You can use Scala’s built-in priority queue as your heap implementation.

```scala
import scala.collection.mutable.PriorityQueue

class HeapSorter[T <% Ordered[T]] {
  def sort(toSort: List[T]): List[T] = {
    ...
  }
}
```

**Note:** A more general input type to `sort` would be `Seq[T]`. `Seq` is a trait for any kind of ordered data (`List` extends `Seq`). If you want to explore the `Seq` trait, change the input and output types to `Seq` instead.

### 5 Study Questions

1. Given a collection of \( n \) items, what is the running time to sort them using heapsort? How much space is required?
2. Why does the `Alert` class extend `Ordered[Alert]`?
3. What is the difference between a heap and a priority queue?
4. Assume you had a non-empty heap from which you removed the root (max) element, then you reinserted that element back into the heap. Would you get the original heap back again or not? Develop an example or two to illustrate your answer.

5. Think about the previous question again, but for BSTs instead of heaps.

6. Assume you had $N$ numbers to store in a BST, and you were using an array to store the tree contents (with the same formulas as in heaps for computing relationships between parent and children nodes). What size of array would you need (in terms of $N$)?

7. If you had to implement trees with an arbitrary number of children, rather than two children as in binary trees, what would be the advantages and disadvantages between using classes or arrays to store the 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