Lecture 24: What are Heaps Used For?

11:00 AM, Mar 17, 2021

Contents

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)
- How to use heaps for sorting

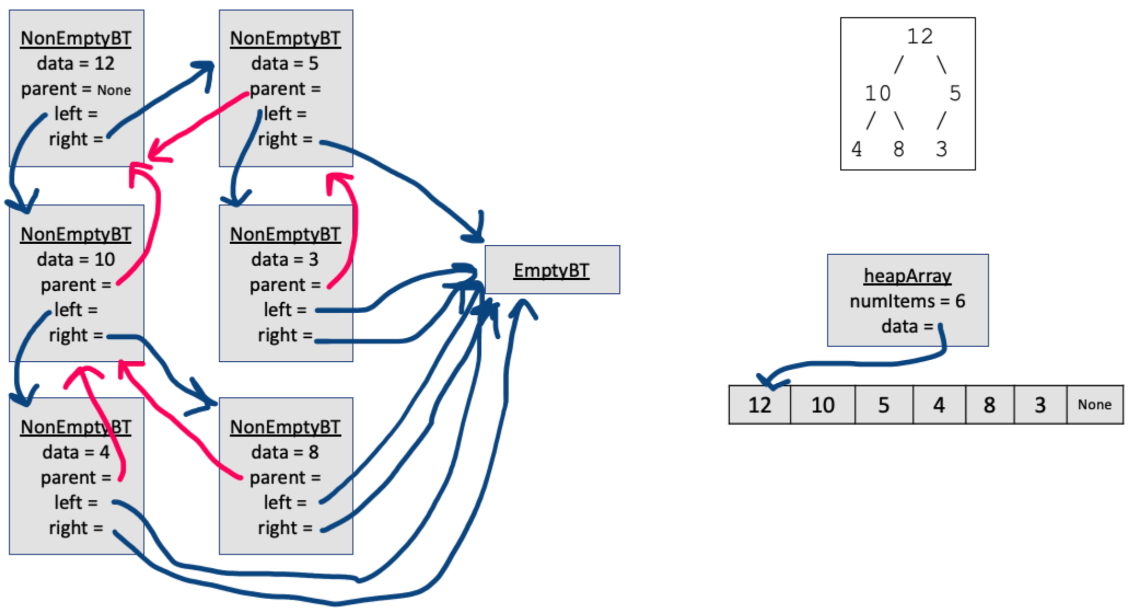
1 Review of Class- and Array-Based Heaps

So far, we've seen two ways to implement heaps: through binary tree objects and classes, and as embedded in arrays. We started with a review of the runtime and space usage of each approach.

The runtimes are similar (assuming a doubly-linked tree) because the main cost in the insert/delete algorithms lies in navigating the underlying tree while we swap elements. Both implementations give constant-time access to the children and parents of a single node, therefore the run-times are the same.

The class-based implementation will use more space, however. If we imagine a `NonEmptyBT` class for the binary-tree nodes with parent and child fields, then each `NonEmptyBT` object takes 5 spots in memory (one for the object itself, one for the data stored at that tree location, one for the parent, and one for each of the two children). On an n-element heap, the class-based version needs $5n$ space. The array version, in contrast, needs one slot for the heap object itself, two fields for the number of elements and the array link, plus an array of n elements. That results in space usage of $n + 3$. While we don't care (much) about multiplicative constants with running time, we care more when it comes to space usage. The array is therefore much tighter.

The following diagram shows the two memory layouts:


NonEmptyBT data = 12 parent = None left = right =
NonEmptyBT data = 5 parent = left = right =
NonEmptyBT data = 10 parent = left = right =
NonEmptyBT data = 3 parent = left = right =
NonEmptyBT data = 4 parent = left = right =
NonEmptyBT data = 8 parent = left = right =
EmptyBT
12 / \ 10 5 / \ / 4 8 3
heapArray numItems = 6 data =
12 | 10 | 5 | 4 | 8 | 3 | None

The class-based layout has a single `EmptyBT` object because the case-class definition we saw for heaps (lecture 23) defined `EmptyBT` as an `object`. In Scala, declaring something as `object` means that only one object from that class gets created.

2 Heaps in Practice: 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 A Priority Queue of Numbers

Priority queues are built into many languages, including Scala. Here's an example (the `insert` operation is called `enqueue` and the `delete` operation is called `dequeue`. The `head` operation returns the value at the top of the heap (aka, front of the queue) without removing it.

```
import scala.collection.mutable.PriorityQueue // data structure built on
   heaps

object Main extends App {
  // Make/manage a heap of numbers
  val numHeap = new PriorityQueue[Integer]
  // add items
  numHeap.enqueue(6)
  numHeap.enqueue(1)
  numHeap.enqueue(3)
  numHeap.enqueue(8)
  println("the heap contains: " + numHeap)
  // look at max item
  println("the top value is: " + numHeap.head)
  println("the heap contains: " + numHeap)
  // remove and return max item
  val max = numHeap.dequeue()
  println("the max was: " + max)
  println("the heap contains: " + numHeap)
}
```

2.2 Putting Alerts into a Priority Queue

Putting a class that we've defined, like `Alert` in a priority queue (or heap) requires Scala knowing when one item is "larger" than another. This is built-in for types like Integer and String. But what about alerts, for which there is no default notion of "greater than"?

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. This method returns 0 if the two items are "equal" with regards to ranking, a positive number if the `this` object is larger, or a negative number if the `that` number is larger.

```
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

scala> a1 > a2
res3: Boolean = true
```

That is the same comparison that would get used inside of the `siftUp` method in the `Heap` class implementation that we started in class.

2.3 Heaps that prioritize smaller numbers

The built-in `compare` method on integers prioritizes larger numbers. If you want to prioritize smaller numbers, you have to provide a different comparison to the priority queue. The following StackOverflow post shows two approaches:

https://stackoverflow.com/questions/27119557/what-is-the-easiest-and-most-efficient-way-to-make-a-min-heap-in-scala

3 Heapsort

Last semester, you studied how to sort lists using algorithms such as insertion sort, mergesort, and quicksort. Each of these algorithms produces a new sorted list, rather than mutating the original list. As a reminder, insertion sort has worst-case quadratic time (in the number of elements), mergesort has worst-case $nlog(n)$ time, and quicksort has worst-case quadratic time that is often $nlogn$ (depending on whether the elements split fairly evenly around the pivot).

When you studied these last semester, you talked only about runtime, but not space consumption. All three of these algorithms, when implemented with immutable lists, create multiple intermediate lists (e.g., the lists of smaller and larger elements during quicksort).

It turns out that 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 repeatedly pull the max element out into a list until the heap is empty. Here's a sketch of the idea, assuming an input list of `List(33, 157, 18, 155, 32, 17, 51)`:

```
val heap = new PriorityQueue[Int]
// enqueue all the elements
heap.enqueue(33)
heap.enqueue(157)
heap.enqueue(18)
...
// dequeue all the elements into a List
List(heap.dequeue, heap.dequeue, heap.dequeue ...)
```

Stop and Think: What are the running time and space usage of heapsort?

In terms of running time, each call to `enqueue` and `dequeue` is $log(n)$ (for n elements). Since we add and remove each element once, heapsort has $nlog(n)$ worst case running time.

In terms of space, the heap requires an array of size n (as we saw earlier in these notes). Heapsort thus uses less intermediate space than the algorithms you saw last semester, with a better guaranteed runtime than quicksort.

Stop and Think: Does heapsort mutate the original list?

No. Heapsort mutates the underlying array, but the original list is left unchanged.

4 What would a Mutating Sort Look Like? (Optional)

The `sort` operation in some languages (like Java) mutates the existing list. What does that look like? This section is just for fun, for those who want to see an outline of how this works. This outline shows how to implement quicksort within a single array by swapping around elements (any of the algorithms you studied last semester could be done within an array). Such algorithms are called "in-place".

Here is an example of quicksorting an array in-place:

- Initial Array:

33	157	18	155	32	17	51

- Step 1: Choose 51 as the pivot element.
- Step 2: Move the elements less than 51 to the left part of the array, and move the elements greater than or equal to 51 to the right part of the array:

33	18	32	17	155	157	51
L	L	L	L	R	R	pivot

- Step 3: Put the pivot element in between:

33	18	32	17	51	157	155
L	L	L	L	pivot	R	R

  and then recursively sort the left and right parts of the array:

17	18	32	33	51	155	157
L	L	L	L	pivot	R	R

- Final (Sorted) Array:

17	18	32	33	51	155	157

Interestingly, when we sort an array in place (and, more generally, when you do any sort of in-place operation), we need not return anything. The effect of quicksorting an array in place is not to produce a new sorted array; rather, it is to modify the given unsorted array.

The most interesting part of in-place quicksort implementation is the partitioning scheme. How would we go about moving all the elements less than the pivot to the left part of the array, and all the elements greater than or equal to the pivot to the right part of the array, in place (and efficiently)? There are a couple of approaches; we will show one of them.

4.1 Partitioning

Use two indices: `left`, which is initialized to index the first element of the array, and `right`, which is initialized to index the last element of the array. The array is then traversed (*sans* the pivot element) by moving the left index to the right and the right index to the left, while maintaining the following properties:

- All data stored at indices less than `left` have values less than the pivot
- All data stored at indices greater than or equal to `right` have values greater than or equal to the pivot.

Here is a simple iterative algorithm that does this:

1. Increment `left` if it indexes a cell whose value is less than the pivot value, otherwise leave `left` in place.
2. Decrement `right` if it indexes a cell whose value is greater than or equal to the pivot, otherwise leave `right` in place.
3. If `a[left]` $\geq$ `pivot` $>$ `a[right]`, then swap `a[left]` and `a[right]`, and then increment `left` and decrement `right`.
4. Repeat until `left` $>$ `right`, at which point the entire array has been processed.

For example, consider the following array in which the pivot is 33. Initially, `left` indexes 51 and `right` indexes 32.

51	31	155	18	181	157	32
left						right

Because $51 \geq 33 > 32$, swap 51 and 32, then increment `left` and decrement `right`.

32	31	155	18	181	157	51
	left				right	

Now, since $31 < 33$, increment `left`, and since $157 \geq 33$, decrement `right`.

32	31	155	18	181	157	51
		left		right		

Since $155 > 33$, `left` cannot be further incremented, unless there is first a swap. But `right` can be decremented, since $181 \geq 33$.

32	31	155	18	181	157	51
		left	right			

And now there should be a swap, since $155 \geq 33 > 18$. After swapping, increment `left` ($18 < 33$) and decrement `right` ($155 > 33$).

32	31	18	155	181	157	51
		right	left			

At this point, `left` exceeds `right`, meaning the entire array has been processed. Observe: the values to the left of `left` are less than the pivot, while values to the right of `right` are greater than or equal to the pivot.

5 Programming Practice (Optional)

If you want some extra programming practice, you could implement this algorithm for yourself:

5.1 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.

```
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.

6 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. If you had a heap of numbers in which the same element appeared more than once (e.g., List(5, 8, 2, 4, 1, 5, 9), would the two uses of the same number be in adjacent array locations? Would one be in the "parent" position of the other? What if the same number appeared twice in a binary search tree?
6. Think about the previous question again, but for binary search trees instead of heaps.

7. Assume you had N numbers to store in a binary-search tree, 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)?

8. 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?
