Lecture 23: Priority Queues
10:00 AM, Mar 19, 2018

Contents

1 Priority Queues 1
2 Binary Heaps 2
3 Heaps 4
4 Aside: The Ordered Trait 5
5 Another Digression: Trees as Arrays 7

Objectives

By the end of these notes, you will know:

- what a heap is
- what a priority queue is

By the end of these notes, you will be able to:

- implement a mutable tree using an array

1 Priority Queues

By now you are familiar with a number of abstract data types, including lists, stacks, queues, maps (i.e., dictionaries), and sets. The next abstract data type you will learn about is the priority queue.

A priority queue is useful for storing a collection of data whenever each datum is accompanied by a value, its so-called priority. These priorities are used to—you guessed it—prioritize elements of the collection. Together they determine an order in which the elements of the collection are visited.

Priority queues generalize both stacks and queues. A stack is a priority queue in which the last element inserted has highest priority (i.e., it is visited first). A queue is a priority queue in which the first element inserted has highest priority (i.e., it is visited first).

The operations required by a typical priority queue interface, in which lower priority elements of the collection are visited first, are:

\[\text{We could just as well implement a priority queue in which elements with higher priorities are visited first.}\]
When are priority queues useful? Anywhere that your program may be adding values with arbitrary priority to the queue. Consider a hospital emergency room: patients are constantly arriving, some with more serious injuries than many who have already been waiting. Those new arrivals should be seen sooner because their needs are more dire. If we just used an ordinary queue, patients would be seen in arrival order!

The main topic of these notes is **heaps**, a data structure for implementing priority queues. But before we discuss heaps, let’s think about why the data structures you’ve already learned about are not suitable for this task.

Take linked lists, for example. We can insert new elements into a linked list in constant time. But, if we were to do so, then we could not find an element of minimal priority in constant time; we would have to search through the entire list to find one.

On the other hand, if we were to maintain a sorted linked list, then we could find an element of minimal priority in constant time. But then, the insert operation would take linear time, because we would have to search for the appropriate location at which to insert the elements.

So linked lists won’t do. What about a hash table? Unfortunately, they suffer from many of the same issues: we can’t hash priorities, because that would obliterate the ordering, and so we’d have to fully traverse all buckets.

What about trees? Well, a heap is actually a kind of a tree, just like a binary search tree is a kind of a tree. So, yes, trees are the answer.
2 Binary Heaps

Recall from CS 17 the definition of binary search trees. A binary search tree is a binary tree in which the following left and right invariants hold:

- Every value in a node’s left subtree must be less than its value; and
- Every value in a node’s right subtree must be greater than its value.

Here’s an example of a binary search tree (shamelessly stolen from the CS 17 notes):

```
      5
     / \
    3   7
   / \   
  1   4   6
    \    
     11
```

A binary heap is like a binary search tree in that it is a binary tree for which a certain invariant holds. More specifically, there are two kinds of heaps, and for each kind, there is an invariant:

- **min-heap**: Every node’s value is less than or equal to its children’s values.
- **max-heap**: Every node’s value is greater than or equal to its children’s values.

The tree below is an example of a min-heap, because 1, the root node, is the smallest value in the tree; 3 is the smallest value in the subtree for which it is the root node; and 4 is the smallest value in the subtree for which it is the root node. (We will swap between min- and max-heaps as needed, the important piece of information is that more “important” entries are always higher in the tree.)

```
      1
     / \  
    3   4
   / \   
  6   5
    \  
     7
      
```

Here are the same values again depicted in a max-heap:
At this point, it’s reasonable to ask whether balance matters. Recall that, for binary search trees, an unbalanced tree could result in linear-time operations—far worse than the logarithmic time provided by balanced trees. It turns out that the same issues arise with heaps. For instance, this heap is not balanced even though it technically satisfies the max-heap condition:

We’ll see why balanced heaps are easier to work with in the next lecture. For now, just keep in mind that balance is a separate property; i.e., not implied by the min-heap or max-heap properties.

Finally, note that the nodes in BSTs and heaps can store objects of any type, including pairs. So just as BSTs are often used to implement dictionaries (i.e., sets of key-value pairs), heaps are often used to implement priority queues (i.e., sequences of items sorted by their priorities).

3 Heaps

Just like we built hash tables atop arrays, we will build the priority queue interface via a still more abstract interface, which we call IHeap. This interface is nearly identical to that of priority queues, only it is parameterized by a single type, rather than two (and, consequently, it has only a single getMin method).
trait IHeap[T] {

/**
 * @return optionally, the lowest-valued item in the heap
 */
def getMin: Option[T]

/**
 * Deletes the lowest-valued item from the heap
 * @return optionally, the lowest-valued item in the heap
 */
def deleteMin: Option[T]

/**
 * Inserts item into the heap
 * @param item - an item to insert into the heap
 */
def insert(item: T)
}

Given this interface for heaps, our strategy for implementing priority queues will be as follows:

1. We will create a class `Heap[T]` that implements `IHeap[T].`

2. We will implement `IPriorityQueue[T, S]` with a heap that stores item-priority pairs. That is, the `T` that parameterizes `Heap` will not be the same `T` as the item-type in `IPriorityQueue.

4 Aside: The Ordered Trait

Before we can implement heaps, we need to say a few words about ordering. Being able to compare priorities essential to a heap’s operation. Priorities need not always be a single number. For instance, back in our hospital emergency-room example, the policy might be to see patients in worse condition first, but if two patients have the same condition, they should be seen in order of their arrival. Here, a priority is actually a pair: one number for arrival time, and one number for how urgent their case is.

Let’s say that we want to take our standard binary-search implementation and generalize it so that it can search an array of type `T`, rather than an array of any specific type, such as `Int`. But `T` cannot really be just any old type. On the contrary, `T` must implement `Ordered[T]`, an interface that ensures that `T` is `totally ordered`, so that either `a < b`, `a > b`, or `a == b`, for all elements `a` and `b` of `T`.

Scala's `Ordered` trait extends Java's `Comparable` interface. Both have an abstract method `compare`, which must be defined by all classes that implement this trait/interface. Here is the signature for `compare` (using Scala syntax):

```scala
abstract def compare(that: T): Int
```
Based on its signature, we can infer that the \texttt{compare} method returns the result of comparing object \texttt{this} with argument \texttt{that}, with this result encoded as an \texttt{Int}. The convention that governs Scala’s \texttt{compare} method\footnote{and Java’s, and most other mainstream programming languages’} is: return $x$, where

- $x < 0$ if \texttt{this} < \texttt{that}
- $x = 0$ if \texttt{this} equals \texttt{that}
- $x > 0$ if \texttt{this} > \texttt{that}

Because $T$ is totally ordered, exactly one of these cases must hold.

Given a concrete definition of \texttt{compare}, Scala infers the definitions of related operators. In particular, the \texttt{Ordered} trait defines $<$, $>$, $<=$, and $=>$ as follows:

```scala
trait Ordered[T] {
  def <(that: T): Boolean = (this compare that) < 0
  def >(that: T): Boolean = (this compare that) > 0
  def <=(that: T): Boolean = (this compare that) <= 0
  def >=(that: T): Boolean = (this compare that) >= 0
}
```

For example, since \texttt{String} implements \texttt{Ordered[String]}, we can write predicates like "\texttt{cs16}" < "cs18". You don’t need to write these yourself; just define the \texttt{compare} function for your class, and you’ll be able to use $<$, etc. for free.

The syntax for declaring a class that is necessarily \texttt{Ordered} is as follows: class \texttt{MyClass\[T <: Ordered[T]\]}. This is Scala’s analog of the Java syntax class \texttt{MyClass\[T extends Comparable\[T\]\].}

In the interest of efficiency, some basic Scala classes, like \texttt{Int} and \texttt{Double} (i.e., the analog of Java’s primitive types), do not implement the \texttt{Ordered} trait. Consequently, the following class, which is parameterized by an ordered type $T$, cannot be instantiated using \texttt{Int}:

```scala
scala> class MyClass\[T <: Ordered[T]\]
defined class MyClass

scala> val broken = new MyClass[Int]()
<console>:9: error: type arguments [Int] do not conform to class MyClass's type parameter bounds [T <: Ordered[T]]
  val broken = new MyClass[Int]()
```

There are two ways to fix this problem. The first is to use \texttt{RichInt}, which does implement \texttt{Ordered}. But that is somewhat inconvenient and unnatural. Instead, we will just tell the Scala compiler to perform implicit conversion when necessary.

To do this, we subtly change the way we declare that $T$ extends \texttt{Ordered[T]}. Instead of using $<$:, we will use $\%$. This will tell Scala to implicitly convert where appropriate. Thus, the type signature of binary search will change from something like:
def binarySearch(myArray: Array[Int], key: Int): Option[Int] = {
    ...
}

to:

def binarySearch[T <% Ordered[T]](myArray: Array[T], key: T): Option[T] = {
    ...
}

The remarkable thing is that our final implementation requires no changes to the binary search methods themselves!

5 Another Digression: Trees as Arrays

One straightforward way to implement (mutable) trees is as a generalization of linked lists. But trees need not be implemented using a linked data structure. Instead, they can be implemented using arrays. Take, for example, the following binary tree:

![Binary Tree Example](image)

Suppose we label the vertices with numbers assigned in breadth-first order. That is, starting with 0, we label the nodes in order from depth 0 to depth \( n \), and at each level from left to right.

Here’s the corresponding numbering:

![Numbered Tree Example](image)

Given this assignment, we can now store our tree in an array. If we call this array `tree`, then `tree[0] = 1, tree[1] = 3, tree[2] = 4, tree[3] = 6,` and so on.

What can we say about the indices at which a node’s left and right children, and its parent, are stored? Here’s a table that should give us some clues:
The node stored at index 0 (i.e., the root) has no parent. Similarly, the nodes stored at indices 3 through 6 (i.e., the leaves) have no children.

Otherwise, we can derive the following formulas for computing the indices of the left and right children, respectively, of the node stored at index $i$: the left child’s index is $2i + 1$, and the right child’s index is $2i + 2$. Furthermore, the parent’s index is $\lfloor (i - 1)/2 \rfloor$.

---

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: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)

---

*The parents’ index can also be expressed as $\lceil i/2 \rceil - 1$. 