# Lecture 12: Priority Queues

*built on 2019/05/29 at 22:11:50*

Here's a standard interview question:

> You're presented with a huge list of numbers that arrive one by one, and you want to find the smallest $k$ numbers. The input list is gigantic; you can't store them all. But $k$ is not all that big compared to the length of the huge list. What are you to do?

One idea is to maintain a list of $k$ smallest numbers we have seen so far and replace it as we observe more elements. To carry out this idea, we need to find the current largest value and replace it as necessary. For example, consider $k = 3$ and the following input numbers: `20 4 9 11 10 9 8 7 6 3 1`.

Translating this idea into code is straightforward. Shown below is a Java implementation of this idea and what happens with the class variable `botK` in a sample run.

```
public class SmallestK {
  List<Integer> botK;
  int k;
  SmallestK(int k_) {
    botK = new ArrayList<>();
    this.k=k_; }

  void newArrival(int elt) {
    botK.add(elt);
    while (botK.size() > k) {
      int lgNum = Collections.max(botK);
      botK.remove(lgNum);
    }}
  List<Integer> bottomK() {
    // copy and return
    return new ArrayList<>(botK);
  }
}
```

On the sample input, the list `botK` looks as follows:

```
[]
[20]
[20, 4]
[20, 4, 9]
[20, 4, 9, 11]
[4, 9, 11, 10]
[4, 9, 10, 8]
[4, 9, 8, 7]
[4, 8, 7, 6]
...
```

The more involved question is, *what is the running time and how much space does it need?* A moment's thought reveals that after each `newArrival`, `botK` is kept at length (at most) $k$. For each arrival, the list grows to length at most $k + 1$. Both `Collections.max` and `botK.remove(.)` have to each look through the list, costing $O(k)$. So per new arrival, we do $O(k)$ work. Furthermore, the space requirement is also $O(k)$. We can further infer that for an input sequence of length $n$, this costs us $O(nk)$ time.

Now what if we tell you that for a sequence of length $n$, the running time to shoot for is $O(n \log k)$ and you can only use $O(k)$ space?

## 1 Priority Queues

A *priority queue* (PQ) stores a collection of items, each a pair `(key, element)`. These keys are also known as *priorities*. It is these priorities that determine the relative positions of the items in the queue.

We will study a priority queue (PQ) data type supporting the following operations:

- `new()` — make a new priority queue instance (starting out empty).
- `add(elt)` — add to the collection the item `elt`.
- `delMax()` — remove and return the item that has the highest priority.
- `findMax()` — return (without removing) the item that has the highest priority.
- `isEmpty()` — return a Boolean indicating whether the priority queue is empty.

In Java, we can define the following interface for the priority queue data type:

```
public interface PriorityQueue<T> {
  // is this empty?
  public boolean isEmpty();

  // add elt to the priority queue
  public void add(T elt);

  // return the maximum value
  public T findMax();

  // delete the maximum value
  public void delMax();

  // how many entries?
  public int size();
}
```

Being able to support these operations correctly and efficiently is important because this data type has found many applications, including event-driven simulation, data compression (we'll see more of this soon), game playing (AI for two-player gaming, like tic-tac-toe, connect4, checker, chess), statistics (maintain the largest $M$ numbers), scheduling (so that your important jobs don't starve).

We remark that this is different from the first-in first-out (FIFO) queue we studied previously. Unlike in a standard queue, a priority queue removes the highest-priority item first.

*How to support this?* There are many ways to support the operations of this data type, some more efficent than others. We're now going to explore a few implementation options.

### 1.1 Implementation I: Unordered List

In the first implementation, we'll keep things extremely simple. Our underlying data structure will be a list in Java. We store the items in the collection unordered. Therefore, it is easy to add an item to the collection; however, finding the maximum and removing the maximum, though easy to implement, will take some time. In code, we have:

```
import java.util.*;

public class UnorderedPQ implements PriorityQueue<Integer> {
  List<Integer> entries;
  UnorderedPQ() { entries = new ArrayList<>(); }

    // is this empty?
  public boolean isEmpty() { return 0==entries.size(); }

  // add elt to the priority queue
  public void add(Integer elt) { entries.add(elt); }

  // return the maximum value
  public Integer findMax() { return Collections.max(entries); }

  // delete the maximum value
  public void delMax() {
    int maxValue = Collections.max(entries);
    entries.remove(maxValue);
  }
```

```
  public int size() { return entries.size(); }
}
```

### 1.2 Implementation II: Ordered List

Another idea is to keep the list of items sorted. In this case, finding the max and/or deleting it is easy because we know the max item is at the end of the list, so popping the tail will do. Adding an item, however, will be more complicated. We need to write code to insert an item into an already-sorted list. Coding work aside, the running time for this operation is inevitably $O(n)$, linear in the size of the collection currently. Therefore, we have the running times as follows.

**Running Time At A Glance:** The table below shows the running times for the two implementations we just discussed and the running times we anticipate for our third implementation. Let $n$ be the number of items in the priority queue at that moment.

| **Operations** | Unordered List | Ordered List | Heap |
|---|---|---|---|
| `add` | $O(1)$ | $O(n)$ | $O(\log n)$ |
| `findMax` | $O(n)$ | $O(1)$ | $O(1)$ |
| `delMax` | $O(n)$ | $O(1)$ | $O(\log n)$ |

### 1.3 Implementation III: Heap

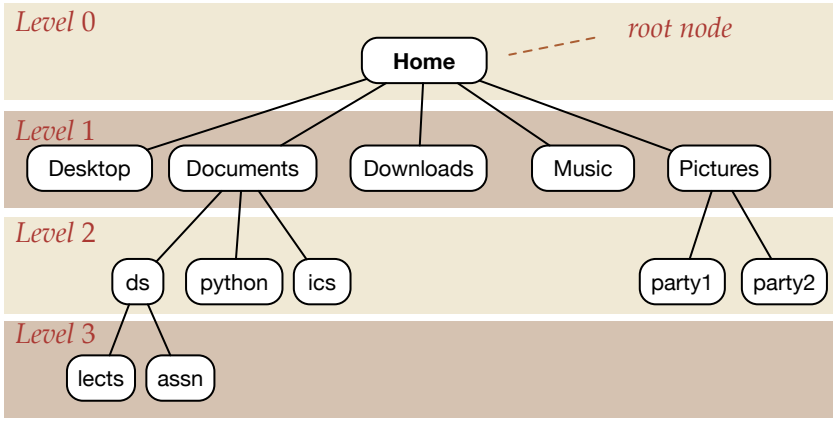
In the previous two ideas, we either keep the collection fully sorted at all times or keep it arbitrarily ordered. But these are the two extremes that either do too much or too little. We strive for a middle ground: *keep the collection partially sorted*—that is, sufficiently sorted to answer our question but without doing too much work.

## 2 Detour: Trees

The data structures we have seen so far share a common property: they are linear—there is a sense of an absolute position on a line from left to right. Often, this confines our thinking and limits what we can achieve.

It is possible, however, to break this pattern. One of the most important "nonlinear" structures is the tree structure, which we'll delve into in future lectures. For today, we'll just try to get a feel for what trees are.

Instead of organizing data linearly, a *tree* is a hierarchical structure with a few notable properties. Each item in a tree is generically called a *node*. There's a special top element known as the *root*. Every node has zero or more *children*. When a node doesn't have any children, it is called a *leaf*. In Computer Science, we often draw trees with its root node at the top and their descendants in subsequent layers below that. An example of a tree structure is given below.

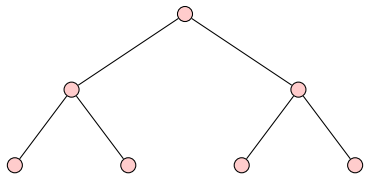


Of particular interest to us is the so-called *binary trees*. These are tree structures where each node can have at most *two* children. We'll see how to take advantage of such a structure to obtain a fast implementation for the priority-queue data type.

## 3 The Heap Invariant

The basic idea is to keep the items sorted enough so that the maximum item can be easily discovered but at the same time, it must not be so rigid so that each `add` operation only has to move a few things around.

To accomplish this, we are keeping a binary tree of a special kind. Remember that a binary tree is a tree where each parent can have at most 2 children. Below, we give an example of a binary tree. This is a *perfect binary tree*—a binary tree where every node except the leaves has two children and the leaves are all at the same level.

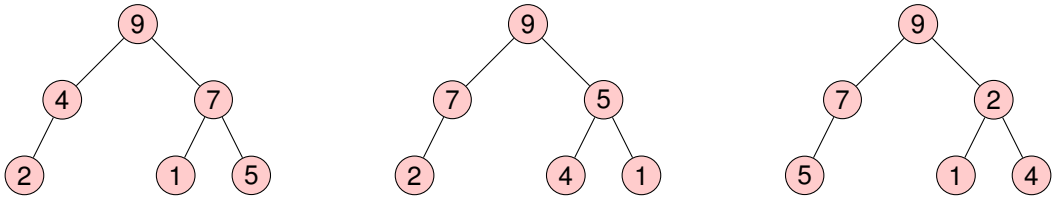


A binary tree isn't always perfect, but in general, one hopes to get close to the shape of the perfect tree.

**Definition 3.1 (Heap Tree)** *A heap tree is a binary tree that maintains the following two invariants:*

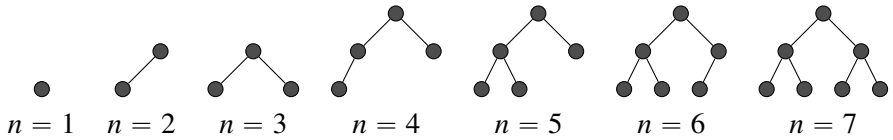
- ***(I1)*** *The priority of an item is greater than or equal to the priorities of its children. (Among the children any order is fine.)*
- ***(I2)*** *The tree is full at all levels except perhaps at bottom level (the leaves) where it must be "left-aligned."*

Let us examine these invariants in turn: The first invariant guarantees, among other things, that the root of the tree (the top node) always has the max item in the collection. As we will soon see, it further guarantees that adding an item or deleting the max will never cost more than the number of levels the heap tree has. The following trees demonstrate the first invariant (or the lack thereof); they don't necessarily respect the second invariant.

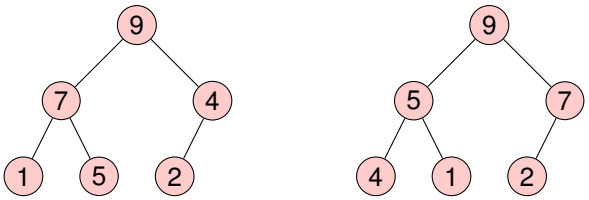



The left and the middle figures show examples of trees that respect the first invariant, though not the second. They have the same set of items. The right figure violates the invariant because although 2 is bigger than 1, it is smaller than 4. Notice that as we made an observation earlier, the maximum element is necessarily at the top.

The second invariant controls the height. By indicating that the tree must be filled level by level so that it is full at all levels except the last, we know that the shape of the tree is completely determined by the number of items—and not at all by what the data items are. For example, the following figure shows the shapes for $n = 1, 2, \ldots, 7$.




We conclude this section by giving two examples of heap trees (satisfying both the invariants) on the set of items $\{1, 2, 4, 5, 7, 9\}$:




## 4 Operations On The Heap

The main operations we have to support are `findMax`, `add`, and `delMax`. We will discuss them in turn.

### 4.1 Finding The Maximum Item

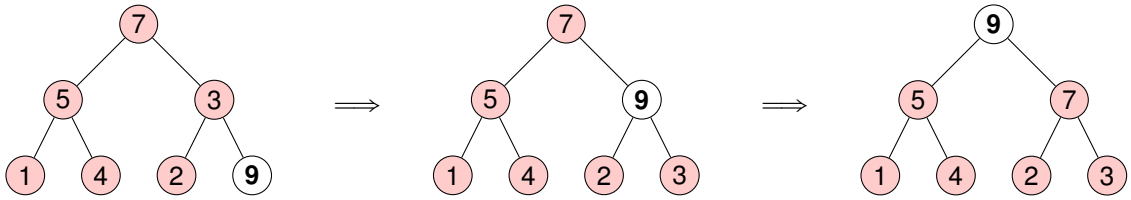
By the heap invariant, we know that the maximum item is located at the root of the heap tree. Thus, to support `findMax`, we can simply return the element at the top of the tree (aka. the root).

### 4.2 Inserting Into A Heap

We have established that the shape of a heap tree is completely determined by the number of nodes. Therefore, there's no question what shape the new tree is going to look like. Specifically, because the tree is filled level by level, the new tree takes the shape of the old tree plus a new node attached to the bottom-right end. We, however, cannot simply put the new item there since it will violate the heap ordering invariant. The more involved question is, therefore, *how we can add the new item and maintain the heap ordering invariant?*

**Idea:** Put it at the bottom-right node anyway but fix what's broken.

To understand how we might fix the tree, let's look at an example (the labels inside the nodes are their priorities). Suppose we're adding 9 to an existing heap (see the left panel). Placing that 9 at the newly-created node at the bottom-right violates the heap ordering property because 9 is larger 3, which at the moment is the parent of 9. We can easily fix this: swap 9 and 3. The resulting tree is shown in the middle figure. Still, the heap ordering property remains violated—9 is bigger than 7, which is 9's parent. Once again, we swap them, resulting in the tree on the right. At this point, we have fully restored the heap ordering invariant.




In general, notice that we can violate the heap ordering invariant where the new item is, and as we exchange the new item and its current parent, it "swims" up the tree, potentially unvealing another location where the invariant is violated. However, because we don't change the tree anywhere else, the violation is contained locally. We summarize this process as follows.

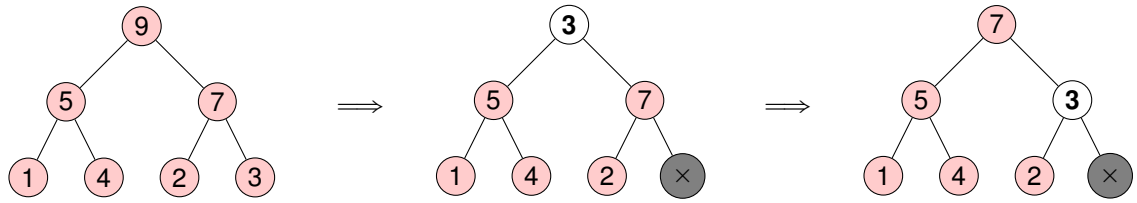
**Promotion in a heap:** As the priority of a node becomes larger than its parent's priority, we can fix this in a few simple steps, as outlined by the pseudocode below:

```
def swim(current_node):
  while current_node is not the top (i.e. the root):
    if current_node vs. its parent violates the invariant:
       exchange them
       current_node = parent(current_node)
    else:
       break
```

### 4.3 Deleting The Maximum Item

We know already that the max item lives at the top of the tree (i.e., the root). To delete the maximum item, we will remove the item at the root, but then, we'll also need to fix up the tree—if simply delete the root, the result isn't really a tree. However, we can take a hint from the heap invariants: because the shape of the tree is completely determined by its size, we know what the tree must look like after we are done.

> **Idea:** Take the node at the bottom-right, place it in place of the root (which we wish to delete), and fix what's broken.




Similar to the insertion case, we check if the node we place at the top violates the heap ordering—if it does, we swap it with the larger of its two children, causing this node to "sink" down the tree. Heap-ordering violation may still take place, and if it does, it will follow where we sent the new root down. But this eventually has to stop because it will sink to the bottom of the tree or stop if the violation has all been eliminated. We summarize this process as follows.

**Demotion in a heap:** As the priority of a node becomes smaller than one or both of it's children's priorities, we can fix this in a few simple steps, as outlined by the pseudocode below:

```
def sink(current_node):
  while current_node has at least one child:
    let m = the child of current_node that has the larger priority
    if m's priority is more than current_node's priority:
       exchange m and current_node
       current_node = m
    else:
       break
```

### 4.4 Running Time

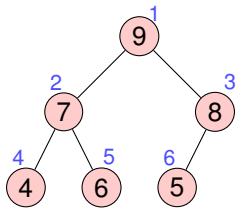
We can't really talk about the actual running time because we haven't specified how the tree is to be represented, but it's worth pointing out that for insertion and deletion, the cost, in the worst case, is proportional to the number of levels in the tree (i.e., we have to go the whole length of the path from top to bottom or vice versa). Hence, if the heap tree is $d$-level deep, `add` and `delMax` will each cost $O(d)$ time.

## 5 Storing The Heap In A List

How can we economically and efficiently store the heap? It turns out the heap invariant gives us an elegant way of mapping the heap tree's nodes into locations in an `ArrayList`. We'll label nodes in level order. That is, the root at the top of the tree is associated with the location 1, and we recursively define the location mapping as follows:

> *If a node is associated with location $k$, its left child is at location $2k$ and its right child is at $2k+1$.*

This means the following tree shape is mapped onto the list using the number next to each node, and hence the keys are stored in a list shown on the right:




| *index* | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| entries | ∅ | 9 | 7 | 8 | 4 | 6 | 5 |

Therefore, we have:

```
def parent(k): return k//2
def lchild(k): return 2*k
def rchild(k): return 2*k+1
```

With these relations, one thing falls out nicely. Using the parent function and noticing the node with the largest index has index $n$, we can argue how deep any heap tree on $n$ node is.

**Lemma 5.1** *A heap tree with n nodes can be at most* $\log_2(n)$ *deep.*

Combining this mapping from the conceptual tree and the list data type, we can implement the priority queue data type using the heap structure as follows:

```
import java.util.*;

public class BinaryHeap implements PriorityQueue<Integer> {
  List<Integer> entries;

  BinaryHeap() {
    entries=new ArrayList<>();
    entries.add(Integer.MAX_VALUE); // dummy value
  }

  public int size() { return entries.size()-1; }
  public boolean isEmpty() { return 0==size(); }

  private int parentOf(int me) { return me/2; }
  private int leftOf(int me) { return 2*me; }
  private int rightOf(int me) { return 2*me+1; }
  private boolean isLeaf(int me) {
    return (me < entries.size()) &&
      (leftOf(me) >= entries.size());
  }

  private void swp(int i, int j) {
    Integer t=entries.get(i);
    entries.set(i, entries.get(j));
    entries.set(j, t);
  }

  private void swim(int loc) {
    int me=loc;
    int p=parentOf(me);
    while (p > 0) {
      if (entries.get(p) < entries.get(me)) { swp(p, me); me = p; }
      else                                  { break; }
      p = parentOf(me);
    }
  }

  private int findMaxChild(int me ) {
    if (rightOf(me) >= entries.size()) return leftOf(me);
    else {
```

```
      if (entries.get(leftOf(me)) > entries.get(rightOf(me)))
        return leftOf(me);
      else
        return rightOf(me);
    }
  }

  private void sink(int loc) {
    int me=loc;
    while (!isLeaf(me)) {
      int maxChildIndex = findMaxChild(me);
      if (entries.get(maxChildIndex) > entries.get(me)) {
        swp(maxChildIndex, me);
        me = maxChildIndex;
      }
      else break;
    }
  }

  public void add(Integer e) {
    entries.add(e);
    swim(entries.size() - 1);
  }

  public Integer findMax() { return entries.get(1); }

  public void delMax() {
    Integer lastElt = entries.remove(entries.size()-1);
    if (!isEmpty()) {
      entries.set(1, lastElt);
      sink(1);
    }
  }


}
```

## 6 Java's `PriorityQueue`

Java has a built-in `PriorityQueue` implementation. See the documentation for details. For many applications, this will work just fine. But as it turns out, there will still be occasions where you'll need to write your own.