# Lab 10: Priority Queues

## Representing a Tree With an Array

You've seen two approaches to implementing a sequence data structure: either using an array, or using linked nodes. We extended our idea of linked nodes to implement a tree data structure. It turns out we can also use an array to represent a tree.

Here's how we implement a binary tree:

- The root of the tree will be in position 1 of the array (nothing is at position 0). We can define the position of every other node in the tree recursively:
- The left child of a node at position `n` is at position `2n`.
- The right child of a node at position `n` is at position `2n + 1`.
- The parent of a node at position `n` is at position `n/2`.

## Working With Binary Heaps

### Binary Heaps Defined

In this lab, you will be making a priority queue using a binary min-heap (where smaller values correspond to higher priorities). Recall from lecture: Binary min-heaps are basically just binary trees (but *not* binary search trees) — they have all of the same invariants of binary trees, with two extra invariants:

later)

- **Invariant 2:** every node is smaller than its descendants (there is another variation called a binary *max* heap where every node is greater than its descendants)

Invariant 2 guarantees that the min element will always be at the root of the tree. This helps us access that item quickly, which is what we need for a priority queue.

We need to make sure binary min-heap methods maintain the above two invariants. Here's how we do it:

**Add an item**

1. Put the item you're adding in the left-most open spot in the bottom level of the tree.
2. Swap the item you just added with its parent until it is larger than its parent, or until it is the new root. This is called *bubbling up* or *swimming*.

**Remove the min item**

1. Swap the item at the root with the item of the right-most leaf node.
2. Remove the right-most leaf node, which now contains the min item.
3. *Bubble down* the new root until it is smaller than both its children. If you reach a point where you can either bubble down through the left or right child, you must choose the smaller of the two. This process is also called *sinking*.

## Complete Trees

There are a couple different notions of what it means for a tree to be well balanced. A binary heap must always be what is called *complete* (also sometimes called *maximally balanced*).

## Lab 10 Navigation

except for possibly the last row, which must be filled from left-to-right.

### Writing Heap Methods

The class `ArrayHeap` implements a binary min-heap using an `ArrayList` instead of a manually resized array. Fill in the missing methods in `ArrayHeap.java`.

Respect the abstraction! — `insert`, `removeMin`, and `changePriority` may use the methods `bubbleUp` and `bubbleDown`. `bubbleUp` and `bubbleDown` may use `getLeft`, `getRight`, and `getParent`.

You may find the Princeton implementation of a heap useful. Unlike the Princeton implementation, we store items in the heap as an `ArrayList` of `Nodes`, instead of an array of `Key`. This is because we want to avoid manual resizing, and also because we want to support priority changing operations.

---

## Submission

To submit, you don't need a zip file this time, just `ArrayHeap.java` and `MagicWord10.java`.

---

## FAQ

### What should setLeft and setRight do if a node already exists?

In this case, it's OK to just overwrite the old left or right node.

### The toString method is causing a stack overflow and/or the debugger seems super slow.

The debugger wants to print everything out nicely as it runs, which means it is constantly calling the toString method. If

## Lab 10 Navigation



will cause a stack overflow, which will also make the debugger really slow. The most common culprit seems to be an incorrect `getLeftOf` and/or `getRightOf`.