Priority Queue, Binary Heaps

Priority Queue ADT

- Priority queues are a collection whose order is indirectly determined by the user
- It allows adding elements, and peeking/removing the **smallest** element

```
public interface PriorityQueue < T extends Comparable < ? super T > > {
    void add(T t);
    T peekMin();
    T removeMin();
}
```

Sorted Array

- O(N) insert (lg(N) to find the right spot, O(N) shifting)
- O(1) peek and remove min (removing from the end of an arraylist)
- Sorting is smart, but keeping stuff in a sorted array means shifting to keep it sorted on insert
- Would using a linked list instead help?

Binary Search Tree

- All operations are O(height), O(lg N) if we maintain balance (coming up)
- Insert needs to traverse down to find a spot to insert (note, PQ's allow duplicate values so we will ALWAYS insert a new leaf)
- Finding the min means going as far left as we can go
- Removing the min means deleting in one of the 2 easy cases, but that's still O(height)
- This seems pretty good, but maybe we can do better

Why not put the min at the root?

- It seems dumb to have travel all the way left in the tree to get the minimum since that's the only value we care about
- Let's put the minimum at the root, then finding it is O(1)
- What do we do about the rest of the tree? How should the nodes be ordered?
- Instead of ordering nodes left to right, as we did with a BST, we'll order them top to bottom

Binary (Min) Heap

- Our new data structure is a binary tree with small values near the root, and big values near the bottom
- The binary heap ordering property is analogous to the binary search tree property: “For each node, it's children's data values must be larger than the node's data value”
- For BSTs, we had to talk about the left/right **subtrees**, here we just said left/right children... did we make a mistake?
- Note, there is no enforced ordering between siblings in a binary heap. Only parent/child relationships are considered

Binary Heap Structure

- For practical implementation considerations, we'll also add one additional constraint on the “shape” of the tree
- Binary heaps are **complete** trees (remember, all levels are totall full, except for the lowest level which is filled in left->right)
- This structure means that binary heaps can be efficiently implemented using the array based binary tree implementation
- For a compelete tree, no space in the array is “wasted” or “empty”

Complete Binary Tree in an Array

- Instead of storing data in nodes and storing left/right pointers, we'll store the whole tree in a contiguous array (arraylist)
- The root at is the start and we'll use array indices to determine parent/child relationships
- We'll store nodes in indices corresponding to the “level order” traversal (top to bottom left to right)
- `[root, root.left, root.right, root.left.left, root.left.right, ... ]`

Index math

- For a given node, what's its parent? I find it easiest to look at the root's children:
- root.left is at index 1, and root.right is at index 2
- For any node, `i`, its parent is at `(i -1)/2` ... check that this is true in general
- For any node it's left child is at index `2*i + 1`, and it's right child is at index `2*i + 2`
- Note, you can also stick the root at index 1 and the indexing is slightly different, but I don't find it any simpler

Binary min heap

- The binary heap is a complete binary tree represented implicitly with an array
- The root, which stores the minimum value, is placed at index 0 or 1 (which convention we pick changes the index math only slightly)
- How do we maintain this structure as we add/remove items?

insert: percolateUp

- In order to keep the tree complete, there's only 1 place to put a new item: the leftmost slot in the bottom row. Conveniently, this is the first unused index in our array
- So, adding the element is simply `backingArrayList.add(newElement)`
- If it's smaller than it's parent, then the ordering property is violated, and we need to fix it. `percolateUp` does that

```
while item isn't the root and item < its parent:
    swap the item with its parent
```

- Does this fix all possible ordering issues?

`insert` analysis

- The cost of `insert` is the cost of `percolateUp` (the actual insertion is O(1))
- In the worst case, the value we inserted is the new min, which would need to be percolated up to the root. Since the tree is complete, we know it's height is O(lg N), so the worst case runtime is O(lgN)
- It's actually probably a LOT better than that in the average case though!

Average case analysis

- Complete trees are REALLY “bottom heavy”
- The last full level has as many elements in it as all the levels above it combined!
- Since “most of the big” elements are near the bottom, on average we only expect to percolate up a few levels, so the average case is actually only O(1)! Pretty awesome!

removeMin: percolateDown

- When we remove the minimum, we need to replace the root. The only node we can delete and keep the tree complete is the rightmost node in the last level (the last element in the array)
- If we move its value to the root, the ordering may be wrong
- We need to “percolate” it down the tree to restore the ordering property
- It seems like we have 2 choices of element to swap with, but we actually don't... why?

percolateDown

```
percolateDown:
    node = root
    while item > either child:
        swap item with the larger child
```

- The analysis is exactly the same as percolate up, except we move the opposite direction across the tree (down instead of up)
- Unfortunately since we're placing a (probably) large item at the root, and since small values are in the top of the tree, in the average case we'll have to percolate almost all the way to the leaves
- So, `percolateDown` is O(lgN) in the worst and average cases

Tweaks

- We can also build a “Max Heap”: everything is the same, except we have `removeMax` instead of `removeMin` and a parent's data must be **greater** than its children, instead of less than them.
- If we have a min heap and want to use it as a max heap, can we do that?

Building a heap

- If we have an unsorted array, we could do the following:
- `for i = N -1 to 1: percolateUp(i)`
- this is basically calling add repeatedly but without changing the size of our arraylist
- The average case runtime of this is O(N), but the worst case is O(N lg N) because each percolate could require us to traverse the height of the tree if we're unlucky

Improved algorithm: heapify or build_heap

- We can actually build a heap in O(N) time guaranteed with a bit of cleverness!
- For any node, if it's children are legal binary heaps (satisfy the order property), calling percolate down on it makes a legal binary heap containing it and its children (why?)
- Is this better? We saw that `percolateDown` actually had a worse average runtime than `percolateUp` when we analyzed `removeMin`?
- It turns out yes, we'll see the analysis in a bit. Here's the algorithm:

```
heapify(arr):
    for i = arr.size/2 down to 0:
        percolateDown(arr, i)
```

Correctness of heapify

- Why start at n/2? For any leaf node, we don't need to percolate down, since there are no children. If our array is size n, the last n/2 entries are leaves
- As we go backwards through the array, we're moving from bottom to top of the tree, so the “bottom” part contains valid “subheaps” as we go
- The last call to percolate down will be on the root with it's left and right subtrees both valid heaps
- So it works, but is it fast?

Analysis of heapify

- Let's look at each call to percolate down, and how many levels down they'll have to percolate in the worst case. We'll assume a **perfect** tree
- The first level we perocolate down on is the 2nd to lowest, which will contain n/4 nodes. Since there's only 1 level below it, these calls to percolate could only go down 1 level, worst case
- The next level has n/8 nodes, which could go down 2 levels
- The next has n/16 nodes, which could go down 3, ...
- So the total number of levels that all nodes we call percolateDown on could go down is `1*n/4 + 2*n/8 + 3*n/16 * 4*n/32 ...`
- The sum turns out to be O(N) (this makes sense given that the denominator is shrinking way faster than the numerator of the fractions)

Heapsort

- We can reduce the memory overhead and sort in place if instead of using a min heap, we use a max heap (support `extractMax` instead of `extractMin`), storing big elements up top, small elements at the bottom
- The beginning part of our array is the max heap, and the end of the heap will store the end of the final sorted array
- Every time we extractMax, our heap gets smaller by one when we delete a leaf, and then we can stick in the max value we extract in that spot!
- This is also REALLY easy to implement because `heapify` and `extractMax` both only require the `percolateDown` helper. We don't even need to implement `percolateUp`!
- You can think of this as a backwards, optimized version of selection sort

Heapsort comparison

- While heapsort is O(N lg N) worst + average case, it tends to be slower than mergesort and quicksort
- But, it's simpler to implement than either, runs in-place, and isn't susceptible to bad choices of pivot, so can sometimes be useful

One more use of heap: k-smallest items

- A common problem is to find the k smallest items in a collection (ie the smallest, 2nd smallest, 3rd smallest, ... kth-smallest), and for example return them in an array
- We can use a binaryHeap to help us with this problem as well, how?
- What's the runtime of this operation, and how does it compare to using quicksort to solve the same problem?
- How does the relationsip between k and N change this?

A faster solution to the k-smallest item problem! Your favorite midterm question!

- The median algorithm from the midterm also solves this problem if we tweak it slightly.
- We simply need to pass `k` as a parameter to the recursive calls, and tweak the logic slightly and we get an algorithm called “quickselect”
- The median version was just this algorithm with `k` hardcoded as `N/2`
- With reasonable choice of pivot, this algorith puts the `k`th element in the spot it would go in a sorted list, and partitions the list so smaller items are to its left and bigger items are to its right
- Quickselect is O(N) for a good choice of pivot, which is pretty incredible!

Recap

- Binary heaps are an efficient way to implement the Priority Queue ADT
- An unsorted array can be “heapified” in O(N) time using `percolateDown`
- This leads to an efficient sorting algorithm (heapsort)
- We can “partially heapsort” to get the K-smallest elements efficiently