# Topic Notes: Priority Queues

Before we discuss priority queues, recall the following tree terminology and properties:

- A *full* binary tree of height h has all leaves on level h.

- A *complete* binary tree of height h is obtained from a full binary tree of height h with 0 or more (but not all) of the rightmost leaves at level h removed.

- We say T is *balanced* if it has the minimum possible height for its number of nodes.

- **Lemma**: If T is a binary tree, then at level k, T has $\leq 2^k$ nodes.

- **Theorem**: If T has height h then $n =$ num nodes in $T \leq 2^{h+1} - 1$. Equivalently, if T has n nodes, then $n - 1 \geq h \geq \log(n + 1) - 1$.

Array representations of trees

Our representation of the tree:
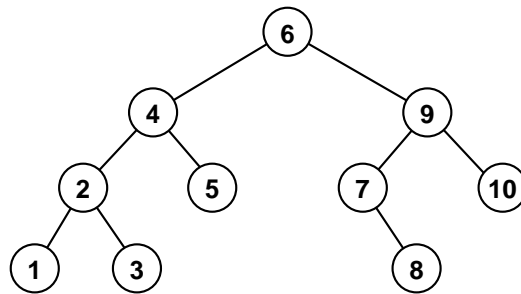
actually looks like this:
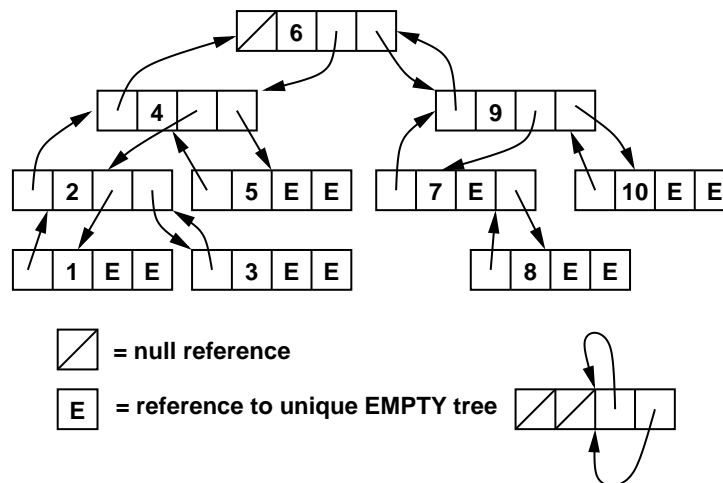
That's a lot of extra references to parents and children, and to empty nodes. So to store 10 actual data values, we need space for 40 references plus the 4 that make up the empty tree instances.

The following array contains exactly the same information:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	4	9	2	5	7	10	1	3				8		

The array, `data[0..n-1]`, holds the values to be stored in the tree. It does not contain explicit references to the left or right subtrees or to parents.

Instead the children of node i are stored in positions $2*i+1$ and $2*i+2$, and therefore the parent of a node j, may be found at $(j-1)/2$

This lets us save space for links, but it is possible that there is a significant waste of storage:

Storing a tree of height n requires an array of length $2^{(n+1)} - 1$ (!), even if the tree only has $O(n)$ elements. This makes this representation very expensive if you have a long, skinny tree. However, it is very efficient for holding full or complete trees. For our example, we need 15 references to hold a tree of 10 items, compared to 40 for the fully constructed `BinaryTree`.

Heaps and Priority Queues

From here, we will look at some ways that trees are used in other structures.

First, we'll consider a structure that seems somewhat like the ordered structures and somewhat like a queue:

A *priority queue* is a structure where the contents are comparable elements and the elements with "small" values are removed before elements with "larger" values.

Waiting at a restaurant can be a priority queue – you get in line but people who are regular customers or who give a tip to the host or hostess may move ahead of you in line.

Same idea when airports are backed up. Planes will get in line for their turn on the runway, but scheduling concerns or fuel issues or whatever else may cause ground control to give a plane which "got in line" later a higher priority and move them up.

An operating system may be scheduling processes that are competing to use the CPU in a multi-programmed system. The next process may be selected based on the highest prioirty process that is seeking a turn on the CPU.

We can define an interface for priority queues that define the following operations:

See Structure Source:
`/home/jteresco/shared/cs211/src/structure5/PriorityQueue.java`

Note that we are restricting to `Comparable` elements, but as in the past, we could develop a version that takes a `Comparator` as an argument to any constructors that could organize objects

of any type in the PQ.

Much like stacks and traditional queues, we only need to define a single `add` and `remove` method. We can add any element at any time, but we can only ever examine or remove the smallest item.

One can implement a priority queue sort of like a regular queue, but where either you work harder to insert or to remove an element (i.e. store in priority order, or search each time to remove lowest priority elements).

An example of this:

See Structure Source:
`/home/jteresco/shared/cs211/src/structure5/PriorityVector.java`

Unfortunately, in these cases either adding or deleting an element will be $O(n)$. (Which one is $O(n)$ depends on which of the two schemes is adopted!)

Using the observation that we don't need to keep the entire structure ordered – at any time we only need quick access to the smallest element – we can provide a more efficient implementation using a structure called at *heap*.

Recall that a complete binary tree is one in which every level is full except possibly the bottom level and that level has all leaves in the leftmost positions. (Note that this is more restrictive than a balanced tree.)

Definition: A *Min-Heap H* is a complete binary tree such that

1. H is empty, or

2. (a) The root value is the smallest value in H and
   (b) The left and right subtrees of H are also heaps

This is equivalent to saying that `H[i] <= H[2*i+1]`, and `H[i] <= H[2*i+2]` for all appropriate values of i in the array representation of trees.

It often makes sense to use the array/vector representation of binary trees here, since the tree is also guaranteed to be complete, so there will never be empty slots wasting space.

Another way of looking at Min-Heap is that any path from a leaf to the root is in non-ascending order.
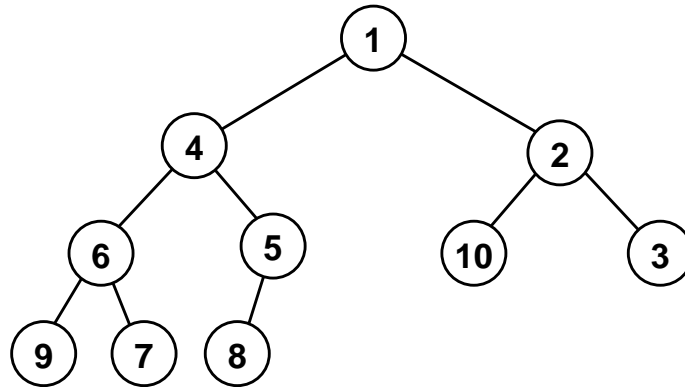
Note that there are lots of possible Min-Heaps that would contain the same set of values. At each node, the subtrees are interchangeable (other than those which have different heights, strictly speaking).

In a Min-heap, we know that the smallest value is at the root, the second smallest is a child of the root, the third smallest is a child of the first or second smallest, and so on.

This turns out to be exactly what is needed to implement a priority queue.

We need to be able to maintain a Min-Heap, and when elements are added or the smallest value is removed, we want to "re-heapify" the structure as efficiently as possible.

Inserting into a Heap

1. Place number to be inserted at the next free position
2. "Percolate" (sift up) it up to correct position

Deleting the Root from a Heap

1. Save value in root element
2. Move last element to root
3. Push down element now in the root position (it was formerly the last element) to its correct position by repeatedly swapping it with the smaller of its two children.

Notice how these heap operations implement a priority queue.

When you add a new element in a priority queue, copy it into the next free position of the heap and sift it up into its proper position.

When you remove the next element from the priority queue, remove the element from the root of heap (first elt, since it has lowest number for priority), move the last element up to the first slot, and then sift it down.

The implementation of this in structure:

See Structure Source:
`/home/jteresco/shared/cs211/src/structure5/VectorHeap.java`

`VectorHeap` uses a `Vector` as its internal storage and provides the interface required for `PriorityQueue`.

Since we use the array/vector representation of binary trees, we can make much of the code more readable (and tree-like) if we have some `protected` helper methods to get the parent and children of the node represented by a given position.

We can easily get the next value to be returned – it is always in position 0 in the `Vector` (assuming there's anything at all in the `Vector`).

Removing an element involves getting the smallest, removing it from the `Vector`, and "heapifying" the remaining values. As we did in the example, we remove the first value, move up the last value to that first position, and "sift" it down to a valid position.

Once you've dealt with the `remove`, an `add` is pretty straightforward. Just put it at the end of the `Vector` and "percolate" it up to restore the heap condition.

.

How expensive are `percolateUp` and `pushDownRoot`?

Each are $O(\log n)$. This is much better than storing the priority queue as regular queue and inserting new elements into the right position in the queue and removing them from the front.

Thought: how might a priority queue could improve the efficiency of a program to compute a Huffman code tree?

Sorting with a Heap (HeapSort)

The priority queue suggests to a new way to sort data. If we have n values to sort, we can add them all to the priority queue, then remove them all, and they come out in order. We're done.

What is the cost of this? If we use the naive priority queue implementations (completely sorted or completely unsorted internal data, making either add or remove $O(n)$ and the other $O(1)$), we need, at some point, to do an $O(n)$ operation for each of n elements, making an overall sorting procedure of $O(n^2)$. That's not very exciting – we had that with a bubble sort.

But what about using the heap-based priority queues?

We can build a heap from a collection of objects by adding them one after the other. Each takes at most $O(\log n)$ to insert and "percolate up" for a total time of $O(n \log n)$ to "heapify" a collection of numbers. That's actually the cost of the entire sorting procedure for merge sort and quicksort, and here, we've only achieved a heap, not a sorted structure. But we continue..

Once the heap is established, we remove elements one at a time, putting smallest at end, second smallest next to end, etc. This is again n steps, each of which is at most an $O(\log n)$ operation.

So we have an overall cost of $O(n \log n)$, just like our other good sorting procedures.

We can actually do a little better on the "heapify" part. Consider this example, which I will draw as a tree, but we should remember that it will really just all be in an array or a `Vector`.
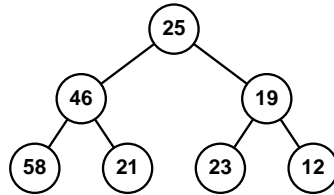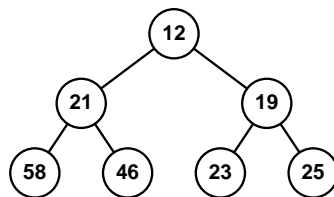

We want to heapify. Note that we already have 4 heaps – the leaves:
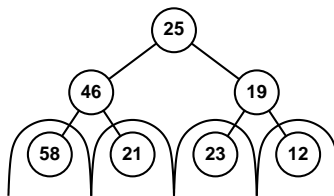
We can make this 2 heaps by "sifting down" the two level 1 nodes:
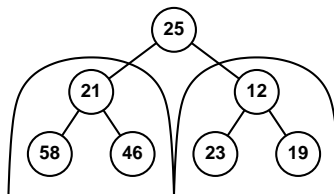
Then finally, we sift down the root to get a single heap:

How expensive was this operation, really?

We only needed to do the `pushDownRoot()` operation on about half of the elements. But that's still $O(n)$ operations, each costing $O(\log n)$.

The key is that we only call `pushDownRoot` on the first half of the elements of the list.

That is, no calls of `pushDownRoot` are made on subscripts corresponding to leaves of the tree (corresponding to $\frac{n}{2}$ of the elements).

For those elements sitting just above the leaves ($\frac{n}{4}$ of the elements), we only go through the loop once (and thus we make only two comparisons of priorities).

For those in the next layer ($\frac{n}{8}$ of the elements) we only go through the loop twice (4 comparisons), and so on.

Thus we make

$$2 * \left(\frac{n}{4}\right) + 4 * \left(\frac{n}{8}\right) + 6 * \left(\frac{n}{16}\right) + \cdots + 2 * (\log n) * (1)$$

total comparisons.

Since $2^{\log n} = n$, we can rewrite the last term to fit in nicely:

$$2 * \left(\frac{n}{4}\right) + 4 * \left(\frac{n}{8}\right) + 6 * \left(\frac{n}{16}\right) + \cdots + 2 * (\log n) * \left(\frac{n}{2^{\log n}}\right)$$

We can factor out the n, multiply in the 2 (to reduce each denominator by 2) and put things into a more suggestive format:

$$n * \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\log n}{2^{\log n}}\right)$$

The sum inside the parentheses can be rewritten as

$$\sum_{i=1}^{\log n} \frac{i}{2^i}$$

This is clearly bounded above by the infinite sum,

$$\sum_{i=1}^{\infty} \frac{i}{2^i}$$

Let's see if we can evaluate this infinite sum:

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$$

We can rewrite this in a triangular form to be able to use some tricks to get the sum:

$$
\begin{aligned}
\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots &= 1 \\
\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots &= \frac{1}{2} \\
\frac{1}{8} + \frac{1}{16} + \cdots &= \frac{1}{4} \\
\frac{1}{16} + \cdots &= \frac{1}{8} \\
\cdots &= \frac{1}{16} \\
&\quad \frac{\cdots}{2}
\end{aligned}
$$

Thus

$$
n * \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\log n}{2^{\log n}} \right) <= 2n
$$

and hence the time to heapify an array or Vector in place is $O(n)$.

The second phase, removing each successive element, still requires n removes, each of which will involve a $O(\log n)$ heapify.

We can, however, do this in place in our array or Vector, by swapping each element removed from the heap into the last position in the heap and calling the heap one item smaller at each step.
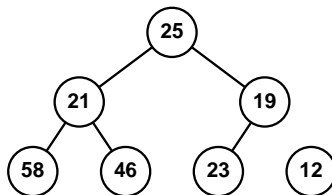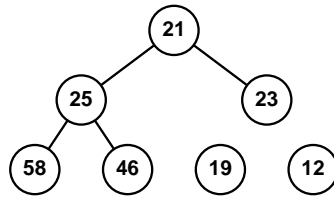

And then we do a sift down of the root (which just came up from the last position):
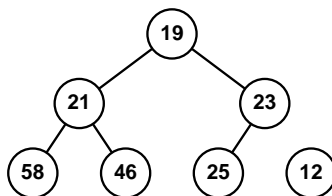
Then the next item (19) comes out of the heap by swapping into the last position, and we sift down the 25:

And the process continues until there is no more heap.

The entire process of extracting elements in sorted order is $O(n \log n)$.

Therefore, the total time is $O(n \log n)$.

Plus, there's no extra space needed!

Note that using a Min-Heap, we end up with the array sorted in descending order. If we want to sort in ascending order, we will need a Max-Heap.

So we have a new efficient sorting procedure to add to our arsenal.

Heapsort Implementation

We can develop a method to perform a heap sort on an array of `Comparable` elements.

See Example:
`/home/jteresco/shared/cs211/examples/SortingComparisons/HeapSort.-java`

All of the comparisons and most swapping happens in the `pushDownRoot` method, which is very similar to the `pushDownRoot` method of `VectorHeap`.

Skew Heaps

The heaps we have considered so far are all required to be stored in complete binary trees. A heap can also be stored in binary trees of other shapes. The same main heap conditions hold: the root contains the smallest element and each subtree is a heap.
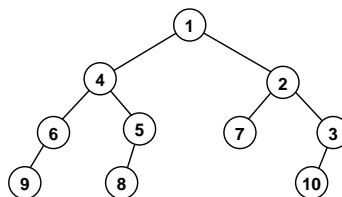
We can define a *skew heap* using our `BinaryTree` implementation. The skew heap itself maintains a reference to the root of the tree and a count (for convenience).

Unlike with the `Vector`-based, complete heaps, we do not have quick access to the "last" element that we used for our `add` and `remove` methods there. So those methods are approached differently.

`add` is accomplished by taking the existing skew heap and a new skew heap consisting of the new element, and "merging" them.

`remove` is done by taking away the root, and merging the two children (which are each skew heaps) into a new skew heap.

So we just need an efficient way to merge two skew heaps into a new one, and we'll be able to write `add` and `remove` easily.

The more general case comes up with the `remove` method, when we need to merge two arbitrary skew heaps. Consider removing from the example above. This leaves us with two skew heaps:
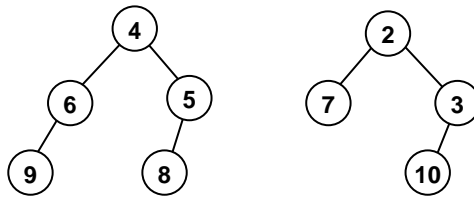
If either of our skew heaps has an empty child, we can easily deal with the merge.

We will not look in detail at skew heaps in class, but you can read further about them in the text and see the implementation:

See Structure Source:
`/home/jteresco/shared/cs211/src/structure5/SkewHeap.java`