Heaps

# Heaps

If you were here last session you will hopefully remember learning about priority queues. Today we will start out with diving a bit further into it by investigating Heaps.

(briefly go over priority queues for any people that weren’t there last week)

Heaps are a data structure that uses binary trees to implement a priority queue. So before we start on Heaps lets learn a bit about trees.

## Trees

So a tree, as you hopefully know already, is a collection of connected nodes which have no loops and with a designated root.

### Terminology

Degree of a node – The degree of a node is the number of children the node has.

Degree of a tree – The maximum degree of all the nodes in the tree.

Leaf – Nodes that have a degree of 0, if a node is not a leaf it is called nonterminal.

Sibling – Nodes with the same parent node.

Level – The level of a node is the number of generations you have to go up to get to the root

Height – The height of a tree is the maximum level of any node in the tree

Forest – Set of disjoint trees.

### Representing trees

Now we have the terminology out of the way how do we represent the tree in memory. There are two main ways of storing the tree:

Link lists – A linked list is a data structure which in this case store x + 1 things, the value of the node and the memory locations of x child nodes. (Draw diagram).

Linear Array – These can only be used to store binary trees. This is just a normal array; the root node would be stored at index 0 but is said to be at position 1. The child nodes location is calculated by using the equation:

For

For example finding the children to the root index.

Then if you want to find the children of those children just repeat this process with one of the children as the parent nodes.

An important point here is that all nodes must have less than or equal to children.

To do the reverse of this is very simple, all you do is divide the position by the 2 and take the floor of the value:

So, for the example above, just to prove it works:

Etc.

An important factor to consider when deciding what tree to use is what do you think the tree will look like. This is because if you had a binary tree which consisted of 8 nodes that each have 1 parent and 1 child (apart from the leaf and root) you would need an array of length just to store them. This is because the size of the array needed to represent a tree is equal to . So for sparsely populated trees linear array can be very space inefficient.

Just to emphasise this, if we look at the worst case space complexity for a tree with a degree of 2 is .

If you couldn’t tell, today we will be using the linear array to represent our heap.

## How it works

So finally, how does a heap work. A heap is a way of efficiently storing numbers in an order, I am just going to use smallest to biggest for now. A heap will always keep the smallest number at its root, and for all nodes that are nonterminal, all children will be larger than it. So every parent is smaller than its children. This is the core concept of the Heap.

## Using the heap

### Pushing a value

As we are using a linear array we can just push the value we want to add onto the end of the array, then we can start moving it into it correct position, the algorithm goes as follows:



This inserting of the value has a complexity of

This is a simple algorithm, so we are going to get you guys to actually implement it now.

### Popping from the heap

To pop a value is a little more complicated:



This is very rough pseudocode, so we want to replace the root node with the last node and shrink the array by 1, then keep swapping the node until its in a valid state.

## Heap Sort

Heaps can be used as a fast way to sort an array. This works by adding all the elements to the array, then popping each value and placing it at the end of the array. So just like when you normally pop an index you swap the biggest value with the value at the end of the tree and decrease the size of the tree. So for example if you had the heap:

[10, 4, 6, 2, 1, 5, 3]

Then you pop:

[3, 4, 6, 2, 1, 5 | 10] (The bar here is used to show where the heap ends)

Then turn it back into a heap:

[6, 4, 3, 2, 1, 5 | 10]

[6, 4, 5, 2, 1, 3 | 10]

Then keep doing this:

[3, 4, 5, 2, 1 | 6, 10] Popping value

[5, 4, 3, 2, 1 | 6, 10] Turning back into heap

[1, 4, 3, 2 | 5, 6, 10] Popping value

[4, 2, 3, 1 | 5, 6, 10] Turning back into heap

…

[1, 2, 3, 4, 5, 6, 10]

## Trying it out

Go to this link:

<https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/>

This another explanation of heaps, and at the bottom of the page there is a problem for you to solve using heaps.

This question give you a line of numbers to add to the queue, then a list of commands. The commands start with a number 1 or 2. If 1 then it wants you to add the number next to it to the heap, if 2 then pop the max number.

## Last weeks problem

Now we are going to go back to last weeks problem and using this new data structure maybe we can solve this now.

So here is the solution we wrote last week, which while solved the problem, it did not do it fast enough. So now copy this code but now rewrite the BasicQueue class to be a heap and try submitting your answer.