BRAC University

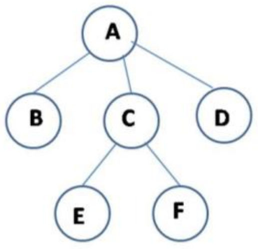
Algorithms

Heap Data Structure

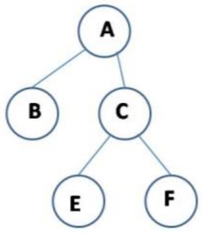
Rubayat Ahmed Khan

10/9/2015

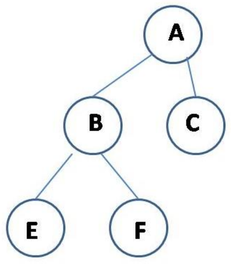
Tree: A tree is an Abstract Data Structure made up of nodes and edges. The following diagram is a tree.



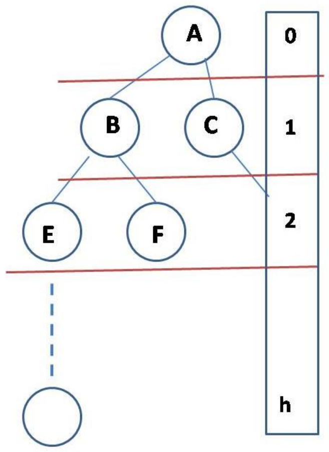
The round objects are called nodes and the lines connecting them are called edges. The node at the top named A is called the root. B, C and D are the children of A and they are siblings as they have the same parent. C is the parent of E and F and this is a sub tree of the entire tree. E and F are called leaf nodes. In a tree a node can have arbitrary number of nodes and a specified number too. In a binary tree can a parent can have a maximum of 2 nodes and a minimum of 0 nodes.



This lecture will be restricted to binary tree only as it is the most widely used. There are versions of binary trees which we will see later. This is an example of a binary tree. There are a few tree terminologies that you must know.
Full binary tree: A tree where every node will have 2 children nodes or no children at all. The above one is a full binary tree.
Complete binary tree: A complete binary tree is the one where nodes are added in the tree in a left to right manner not skipping any position. The above one is not a complete tree because E and F have been added to C keeping the child positions of B empty. A complete tree can have a maximum of 2 children and a minimum of 0.

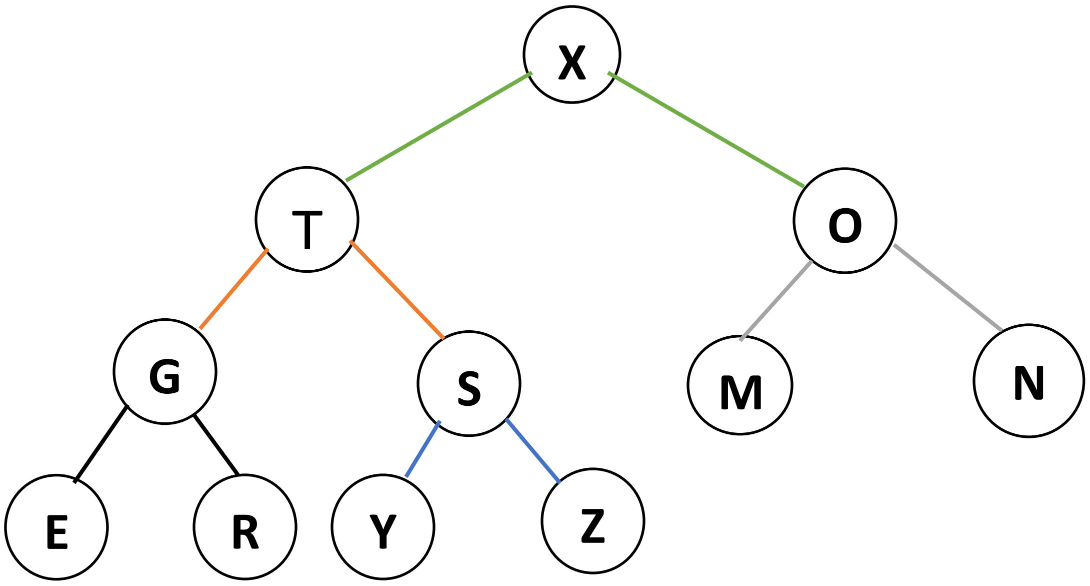


This is a complete binary tree. A complete binary tree does not have to be full. Each node can have 0, 1 or 2 children.
At each level, if we consider the root to be in level 0, the maximum number of nodes are 2^{level}.
The maximum number of nodes that a tree can have is $(2^{highestLevel + 1}) - 1$.
The depth of a node is the number of edges from the root.
The height of a node is the number of edges from that node to the deepest leaf.
The height of a tree is the number of edges from the root to the deepest leaf. This property is very important and we will see it as we progress.
How to find the height of a tree given the number of nodes ?



This is a binary tree of h levels. At level h the maximum number of nodes can be 2^h and the minimum node can be 1. Therefore there can be 1<= nodes<=$2^{h.}$ Let us say there are n nodes. $n=2^h$ when n is max. If we solve this equation to find h in terms of n we get $h = \log_2 n$. Therefore the height of the tree is ⌊log2n⌋.

Heap: Heap is an ADT for storing values. A heap is expressed as a special binary tree pictorially and as its underlying data structure it uses an array. The tree gives heap an advantage to manipulate using a pen and paper quite easily which we will see as we progress. Let me break down the "special tree" as mentioned. A heap has to be a complete binary tree and it must satisfy the heap property.
Heap property: The value of the parent must be greater than or equal to the values of the children. (Max heap). or the value of the parent must be smaller than or equal to the values of the children. (Min heap). There are two types of heaps. Max heap is mostly used. A heap can be either a max heap or a min heap but can't be both at the same time.



The above tree is a heap. Below is the array representation of the above heap. Please note that the tree is used for efficient tracing. While programming the data structure is a simple array. Below is the array representation of the above heap. We start the index from 1.

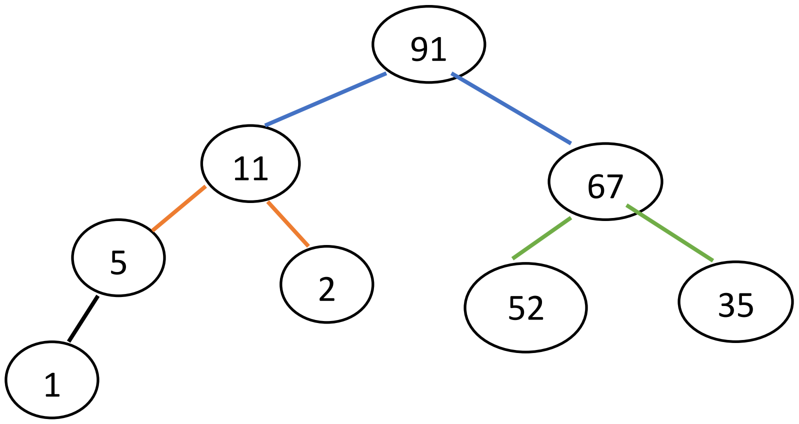
1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	E	R	Y	Z	C

For ith NODE, its parent NODE is i/2.
For ith NODE, its children are 2i and 2i+1.
Note: 2i is the LEFT child and 2i+1 is the RIGHT child.

Benefit of using ARRAY for Heap rather than Linked List
ARRAYS give you random access to its elements by indices. You can just pick any element from the ARRAY by just calling corresponding index. Finding parent and its children is trivial. Linked List is sequential. This means you need to keep visiting elements in the linked list unless you find the element you are looking for. Linked List does not allow random access as ARRAY does. On the other hand each linked list must have three (3) references to traverse the whole Tree (Parent, left, Right).

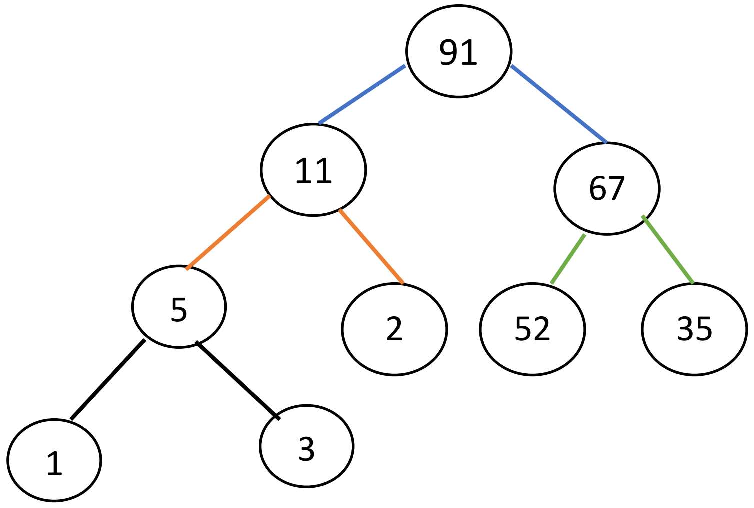
Operations on Heap

1. Insert (): Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.



Consider the above Heap. If we want to insert an element 3, we start left to right at the bottom level. Therefore 3 will be added as a child of 5.

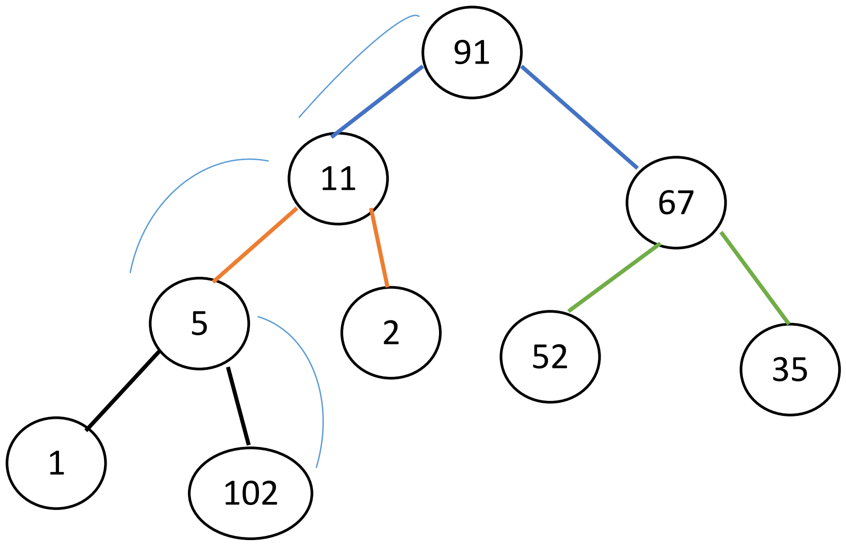
Then the new Heap will look like this:



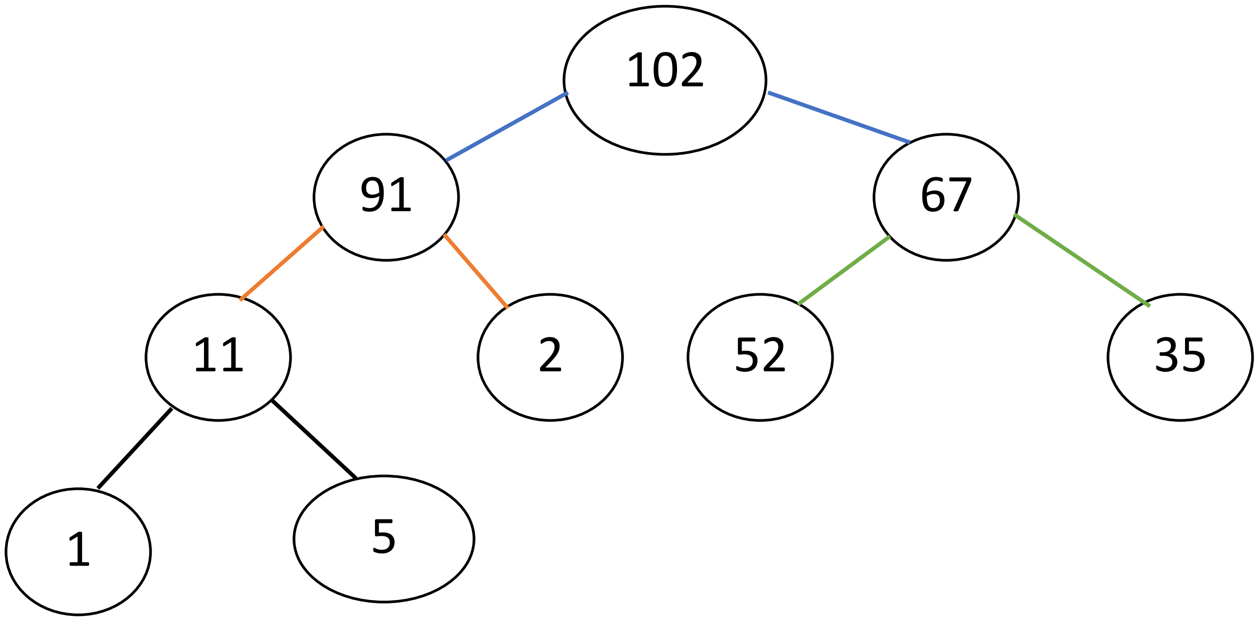
Look carefully five (5) is 3's parent and it is larger. Hence Heap property is kept intact.

What if we want to insert 102 instead of 3?
Let's say we want to insert 102 at the existing Heap. 102 will be added as a child of 5. Now is the Heap property hold intact? Therefore we need to put 102 in its correct position. How we going to do it? The methodology is called HeapIncreaseKey () or swim().

<u>1.1 HeapIncreaseKey ()/swim()</u> Let the new NODE be 'n' (in this case it is the node that contains 102). Check 'n' with its parent. If the parent is smaller (n > parent) than the node 'n', replace 'n' with the parent. Continue this process until n is its correct position.



After the swim() operation the Heap will look like this:



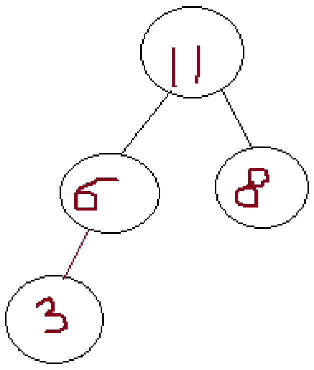
Time Complexity: Best case O(1) when a key is inserted in the correct position at the first go. Worst case is when the newest node needs to climb up to the root. We have learnt that the distance from the leaf node to the root is lg(n) (height of the tree). Hence this is the worst case complexity. O(1) [insertion] + O(lgn) [swim]

Thus insert is a combination of two functions: insert() and swim().

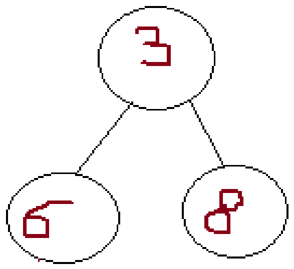
Pseudocode:

```
insert (H, key){
        size(H) = size(H) + 1;
        H[size] = key;
        swim (H, size);
}

swim(H, index){
        if (index < = 1){
                return;
        }else{
                parent = H[index/2];
                if (parent > H[index]){
                        return;
                }else{
                        exchange parent with H[index]
                        swim(H, parent);
                }
}
```

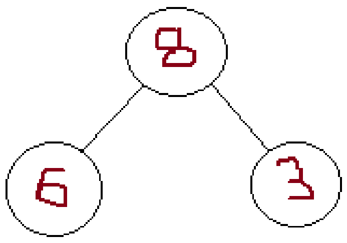
2. Delete (): In heap you cannot just cannot randomly delete an item. Deletion is done by replacing the root with the last element. The Heap property will be broken 100%. Small value will be at the top (root) of the Heap. Therefore we must put it in a right place which is definitely somewhere down the Tree.



Replace the root (11) with the last node (3)



The root is now smaller than its children. So replace 3 with the children with the greater key

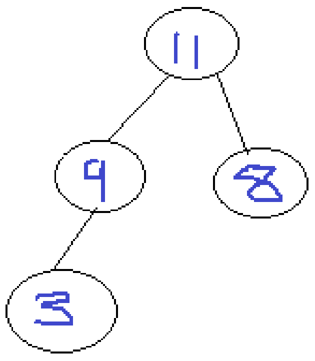


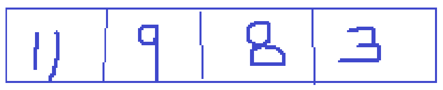
This process of putting a node in its correct place by traveling downward is called sink() or MaxHeapify(). The time complexity of sink is lgn as it might have to sink down to the end of the tree.
Delete is a combination of delete() and sink() hence the worst time complexity is lgn too.

Pseudocode:

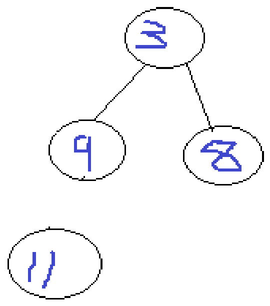
```
delete(H){
        if (size(H)==0){
                return;
        }else{
                exchange H[1] with H[size]
                size --;
                maxHeapify(H, 1)
              }
}

maxHeapify(H, index){
        if (size(H) ==0){
                return;
        }else{
                left = 2*index;
                right=2*index+1;
                if (left <= size && right<=size){
                        exchange H[1] with Max (H[left], H[right]);
                        maxHeapify(Max (left, right));
                }else{
                        if (left<= size && right>size){
                        exchange H[1] with (H[left]);
                     }

            }
}
```

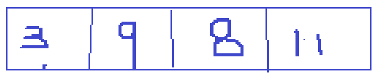
<u>Heap Sort:</u> Delete all the nodes of the heap.

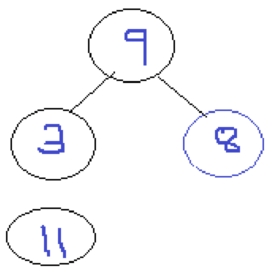


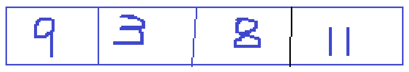


Steps:
Replace the root with the last node, which is basically delete and then sink.



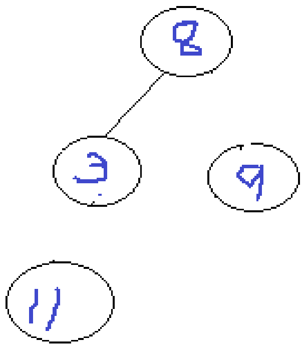


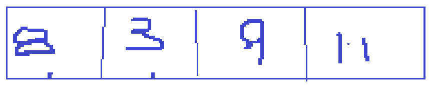




After heapify down

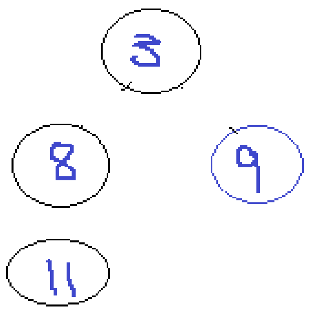
Continue deletion until no node is left. Now replace 9 with 8, which is basically delete again.

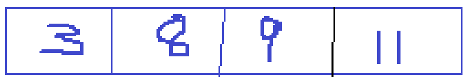




Heapy down not required because the root is bigger than child

Now replace 8 with 3.





After heapify down

This is heap sort. Time complexity O(nlgn). Heap sort is in place, that is, no extra array is required and not stable.

Pseudocode:

```
for all nodes i = 1 to n{
    delete (H);
}
```

Build Max Heap: You are given an arbitrary array and you have been asked to built it a heap. This will take O(nlgn).

Pseudocode:

```
for all nodes i = 1 to n{
    swim (H, i);
}
```