Heap and Priority queue

Types of Binary Tree

- ***Strictly binary tree***: If every non-leaf node in a binary tree has non-empty left and right subtree, the tree is called strictly binary tree.
- A strictly binary tree with n leaves has 2n-1 nodes

- ***Complete binary tree:*** A complete binary tree of depth d or height d is the strictly binary tree, all of whose leaves are at level d***.***
- ***No of nodes in Complete binary tree =*** $2^0 + 2^1 + \ldots\ldots 2^d$

Almost Complete binary tree: In an almost complete binary tree, all the levels of a tree are filled entirely completely except the last level that might be partially filled and it must be filled from left to right.

Binary tree representations and operations on binary trees

- We can perceive binary tree as a data structure in which:
- Data is stored in nodes and these nodes are connected to each other like a binary tree or there is some relation ship among the nodes which can be defined as a binary tree
- Address of root node is stored to recognize the tree
- Given address of root node, following operations may be required:
- Address of a particular node
- Contents of a node
- Children of a node (left or right child)
- Father of a node , etc.

Binary tree structure

```
Struct treenode  {int info;
                  struct treenode *left;
                   struct treenode *right;
                  }
OR
Struct treenode  {int info;
                   Struct treenode  *father;
                  struct treenode *left;
                   struct treenode *right;
                  }
```

Can we store binary tree in array?? Or can we implement binary tree using arrays??

Implicit array implementation of binary tree

- If we have **almost complete binary tree** or **complete binary tree**, then n nodes of the tree can be stored in an array of size n.
- Array index has values from 0 to n-1
- If we number the nodes from 0 to n-1, then two children of **node p** will be **2p+1 and 2p+2**
- Given a left child at **position p**, its right child will be **at p+1 position**
- Given a right child at position **p**, its left child will be at **p-1** position
- Father(p) = **p-1/2**
- Note: Here we have not stored any explicit link to children, all the links are implicit

Implicit array implementation of binary tree(Not possible for every tree)

- If we want to implement a binary tree using arrays and if tree is not an almost complete binary tree or a complete binary tree, then
- we have to keep many blank array locations with some flag indicating that this node does not belong to the original tree.
- So, the number of array locations will be more than no. of actual nodes in the tree and many of them may be empty.
- Hence, if the tree is almost complete binary tree or complete binary tree, it is stored in an array otherwise linked representation is used and each node is created and linked dynamically as done in a linked list

Why?????.. Tree structure

- What is the need to have a tree structure?
- What are the applications where this structure can be useful?
- Will this structure help us in thinking more logical and simple algorithms while solving problems?
- Does it reduce complexity?
- Is it easy to implement?
- What are the overall advantages and disadvantages?
- Let us discuss some applications…

What is heap??

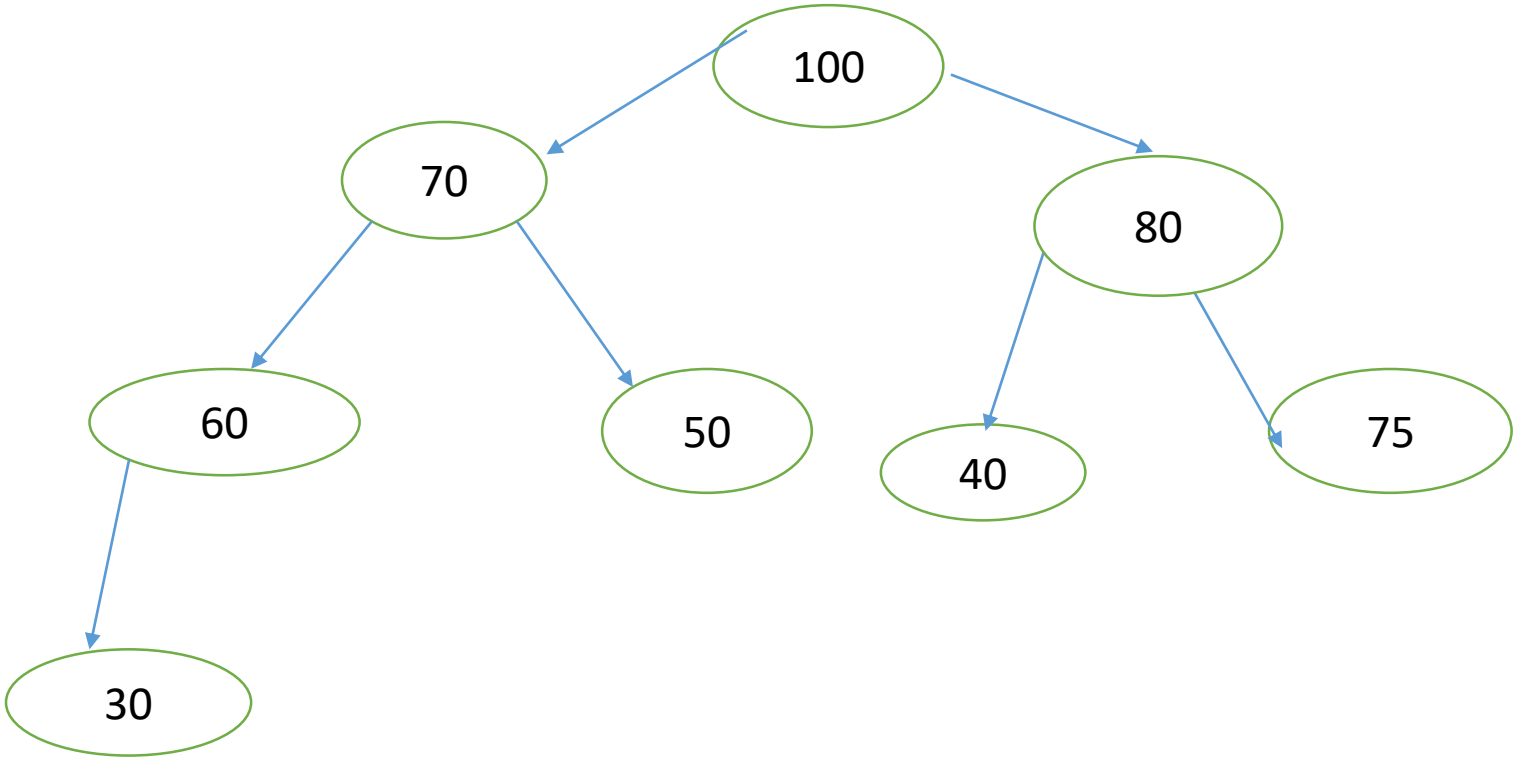
- ***Descending heap or max heap***: Max heap of size n is an almost complete binary tree of n nodes such that the contents of each node are less than equal to its father
- So, root contains the maximum or largest element in the heap
- ***Ascending heap or min heap***: Min heap of size n is an almost complete binary tree of n nodes such that contents of each node are greater than equal to its father
- So, root contains the minimum or smallest element in the heap

Steps to build heap…..

- Insert 45, 67, 23
- Delete
- Insert 34, 78
- Delete
- Insert 5, 100
- Delete
- delete

Example

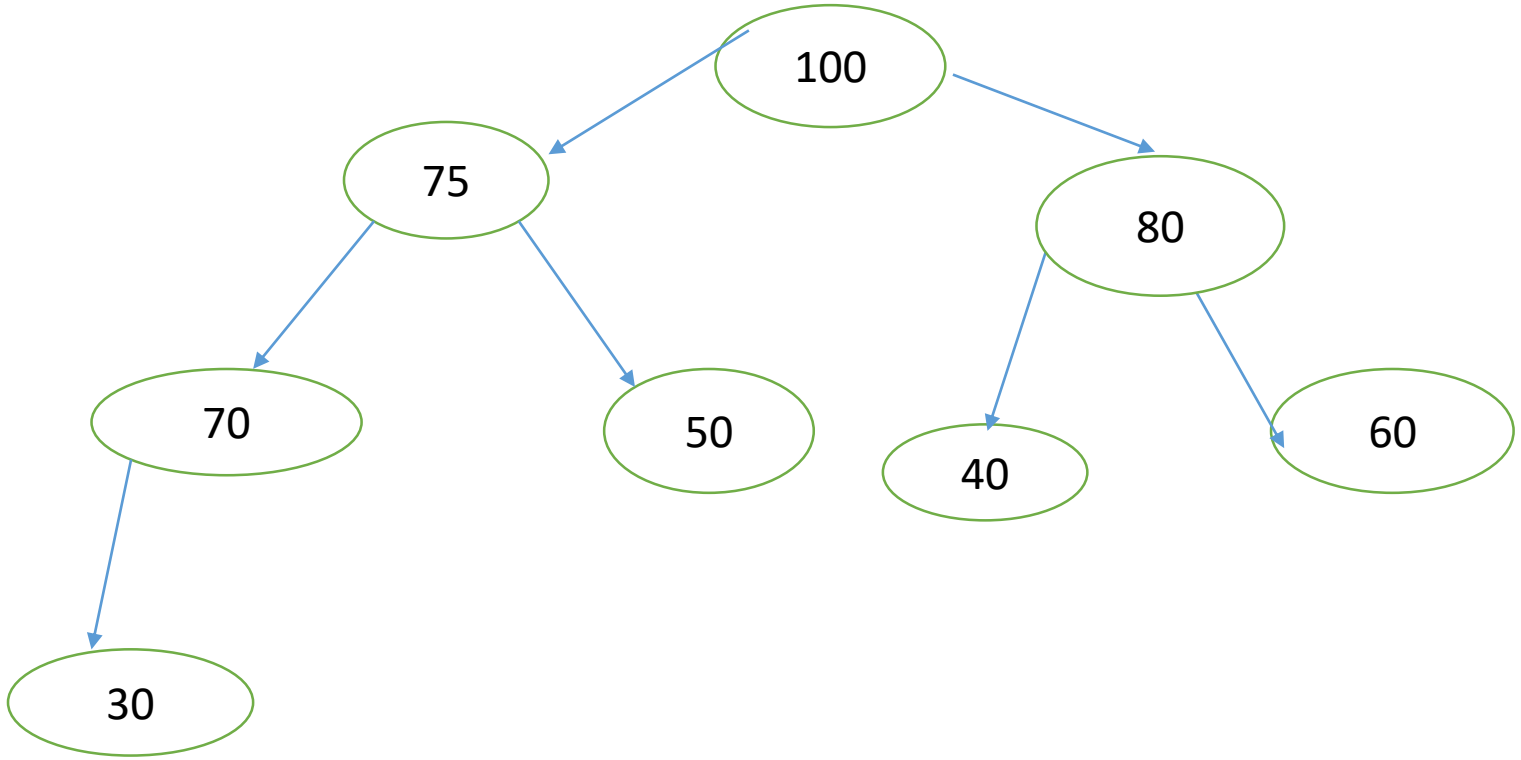
Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)



Many such structures i.e. max heaps may be constructed from this data..

Example

Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)



Why the concept of heap??

- For efficient implementation of priority queue (a queue which always deletes highest priority element from the queue)
- Possible implementations of priority queue are
- **Case1**: Store as an unsorted array or linked list

Complexities: insertion : **O(1)** and deletion **O(n)**

- **Case2**: Store as a sorted array or linked list

Complexities: deletion: **O(1)** and insertion **O(n)**

- **Case3**: Store as an descending heap
- Complexities: deletion: **O(logn)** and insertion **O(logn)**

How to construct heap??

- For each element..

1. Insert the element at the last empty position
2. Compare with its father
3. If father is less than child, swap father and child
4. Continue till father becomes greater than child or we have reached the root

Heap Insert function

```
Heap_insert(int dpq[],int k, int addval)  k = no. of elements already entered in the array
{int i, father;
  i = k;
 father = (i-1)/2;
while(i>0 && DPQ[father] <addvalue)
{ DPQ[i] = DPQ[father];
   I = father;
father = (i-1)/2;
}
DPQ[i] = addval;
}
```

Heap delete function

```
int Heap_delete(int dpq[],int ksize)
/* ksize = no. of elements entered in the array */
{int pos;
pos = DPQ[0];
Heap_adjust(0,ksize-1);
Return(pos);
};
```

Heap Adjust steps

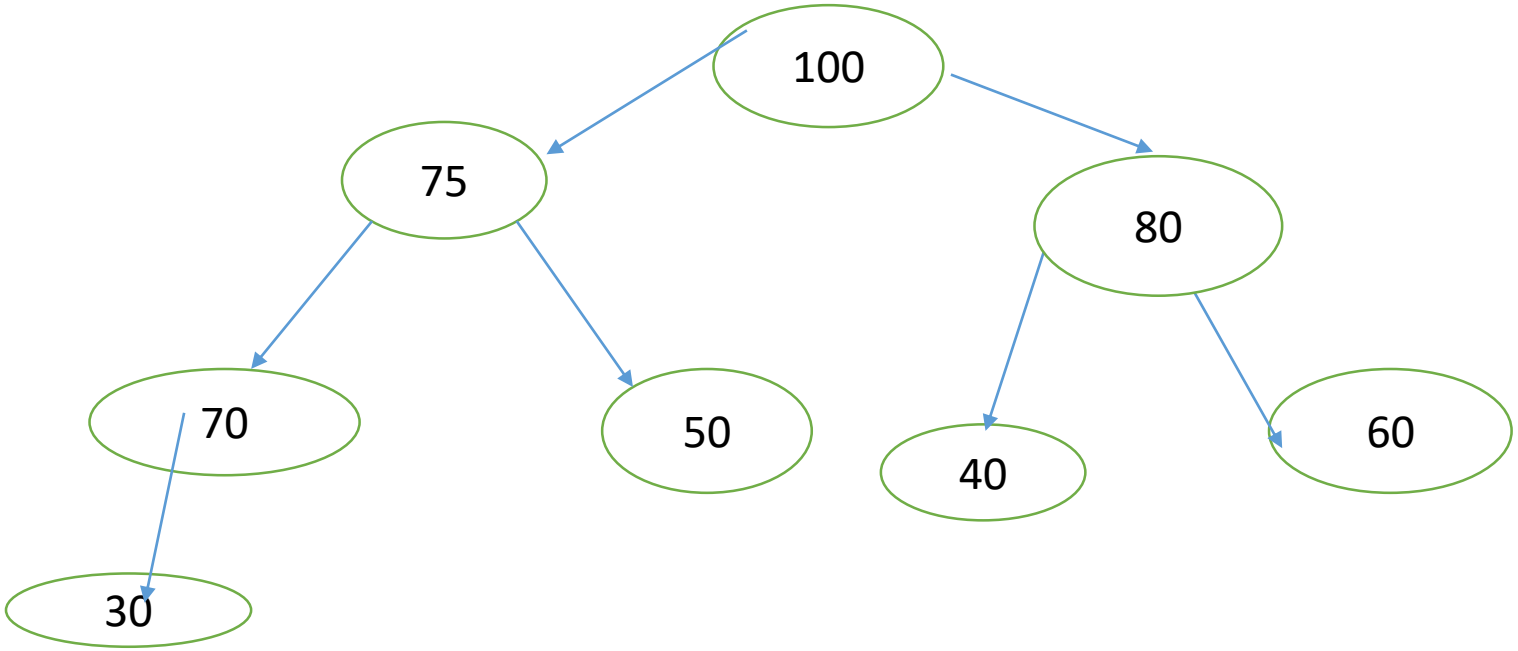
Goal: After deletion, root location is empty, and we need to rebuild a max heap for remaining elements.

Steps:

1. Store the value of last element of array in a temporary variable(***temp***) and reduce the size of array by 1
2. Starting from root, find the index of larger child(***child***)
3. Compare the ***temp*** and ***child*** and put the bigger value in root
4. ***Continue the process till temp gets proper place in the tree***

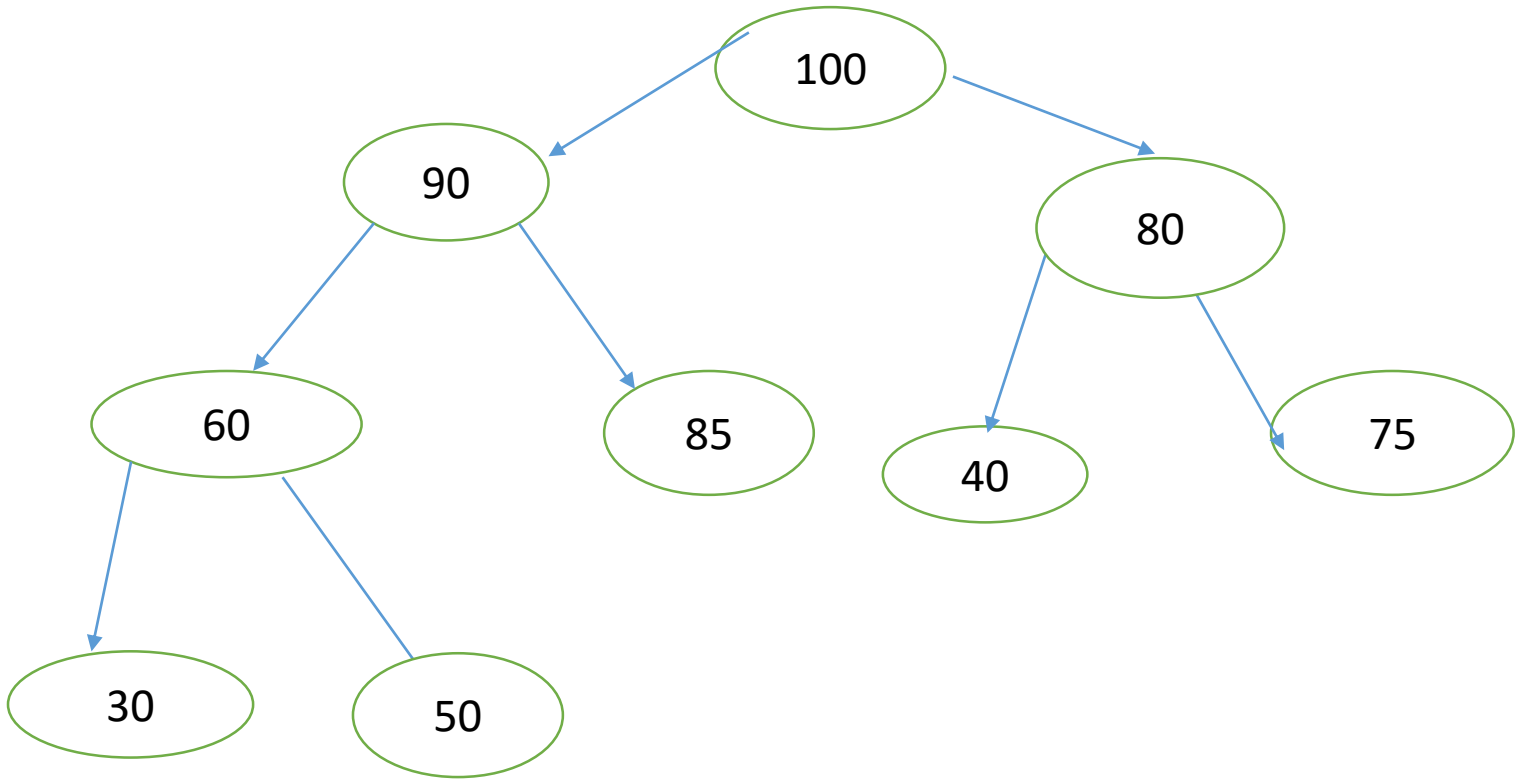
Example

Suppose data is : 50, 40, 60, 100, 70, 80,75, 30 (Max heap)



<u>After deleting 100:</u> 30 will be kept in temp, then 80 will moved to root, then 60 will be moved to position of 80, and 30 will be placed instead of 60

Exercise(deleting root node)



Heap delete function

```
Heap_adjust(int DPQ[], int root, int k)
{int father, child;
father = root; kvalue = DPQ[k];
child = larger_child(father, k-1)
while(child >=0 && kvalue <DPQ[child])
{ DPQ[father] = DPQ[child];
   father=child;
child = larger_child(father,k-1);/* index of larger child of father*, returns -1 if no child/
}
DPQ[father] = kvalue;
}
```

Efficient priority queue implementation

- Implement priority queue as a heap
- This means whenever a new data/client joins the queue, it will be added as if we are inserting the data in a heap
- So, the complexity of insertion is $O(\log_2 n)$
- whenever data is removed from priority queue, element of highest priority is removed
- If data is in a heap, root element is deleted and heap is adjusted
- So, the complexity of deletion is $O(\log_2 n)$

Heap sort

Steps:

1. Build descending heap reading one by one all the data elements of the array to be sorted (basically we are reorganizing the elements of the array)
2. Now delete data one by one and store the deleted data at the end of the array for that heap
3. We will have sorted array after 2n iteration
4. First n iterations of insertion and then n iterations od deletion
5. Total no. of insertions take n X ($\log_2 n$) steps and deletions take n X ($\log_2 n$) steps
6. Complexity of heap sort: O(2n $\log_2 n$)