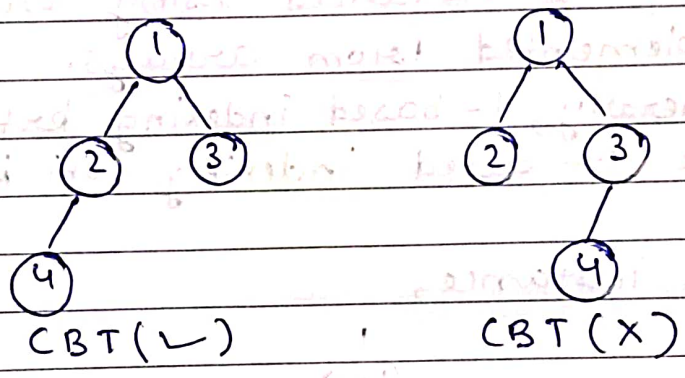
15/12/2023
Friday

* **Heap Data Structure -**

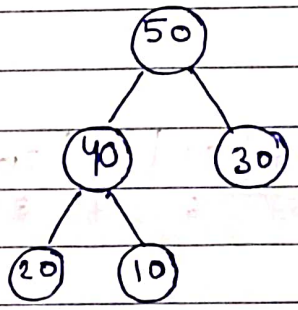
Complete Binary Tree + follows heap property (Min Heap, Max Heap)

- **Complete Binary Tree -** A complete binary tree is a tree in which nodes at every level is completely filled except the last level and nodes in the last level are filled from left to right.

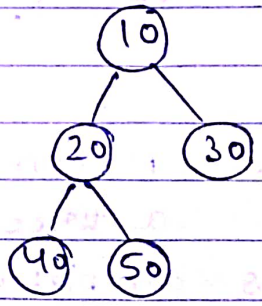


CBT(L) CBT(X)

- **Max heap -** Parent node is always greater than child nodes. This must be true for every node in CBT.

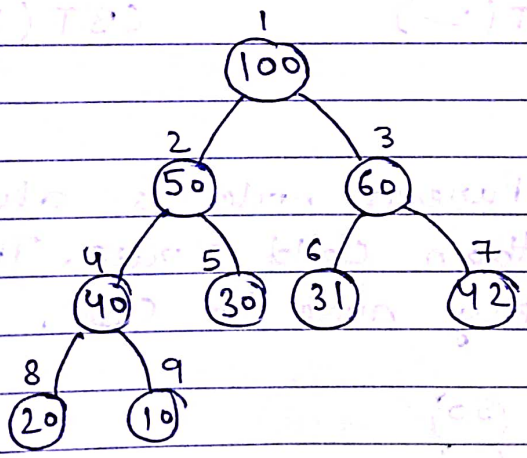


- **Min Heap** - Parent node is always less than child nodes. This must be true for every node in CBT.



Note- Heap is visualized using trees but it is implemented from arrays.
Generally, 1-based indexing krte hai array ki but 0-based indexing bhi kr skte hai.

For instance,



100	50	60	40	30	31	42	20	10
1	2	3	4	5	6	7	8	9

Heap
- → visualize - tree
- → implement - array

For 1-based indexing, -

- If particular child ke parent ko access krna ho, then -

$$\text{Parent index} = \text{child index} / 2$$

- If particular parent ke children ko access krna ho, then -

$$\text{left child index} = 2 * \text{parent index}$$
$$\text{Right child index} = 2 * \text{parent index} + 1$$

For 0-based indexing -

- Child to parent -

$$\text{Parent index} = (\text{child index} / 2) - 1 / 2$$
$$= \frac{(\text{child index} - 1)}{2}$$

- Parent to children -

$$\text{left child index} = 2 * \text{parent index} + 1$$
$$\text{Right child index} = 2 * \text{parent index} + 2$$

* **Insertion in heap -**

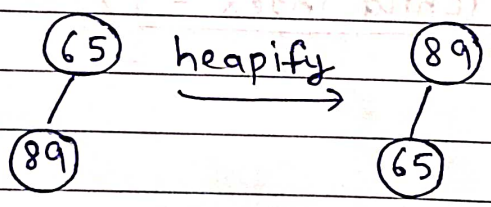
- Insert the element at the vacant or available position.
- Now, place it at its correct position. For that, compare it with its parent node.

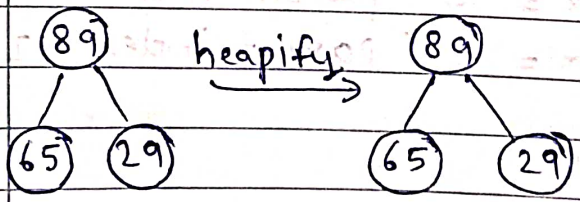
Heapification - If parent element is greater than the element to insert, then swap otherwise ignore. Heapification can be done from top to bottom or bottom to top in the tree:

= **Elements to insert in heap -**

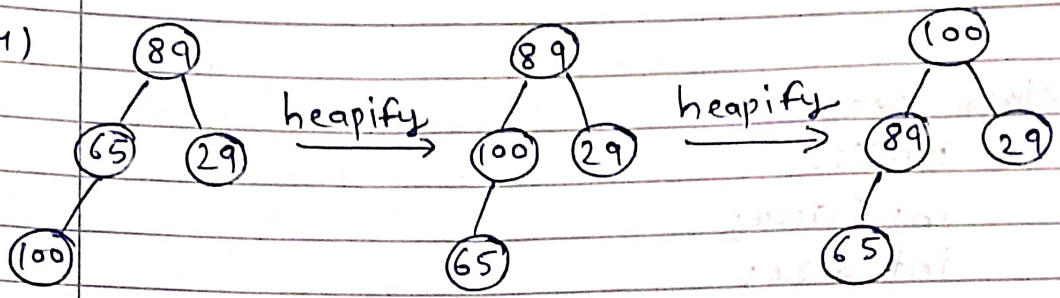
65 89 29 100 10 60

(1) 65 heapify 65

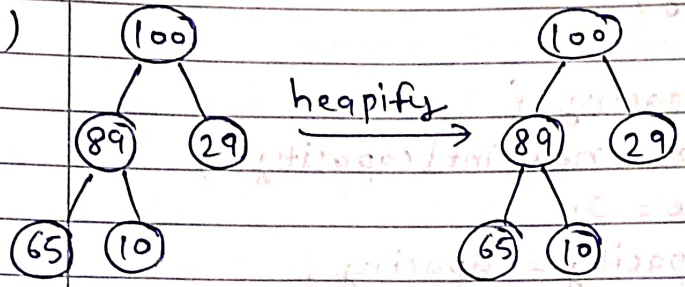
(2) 

(3) 

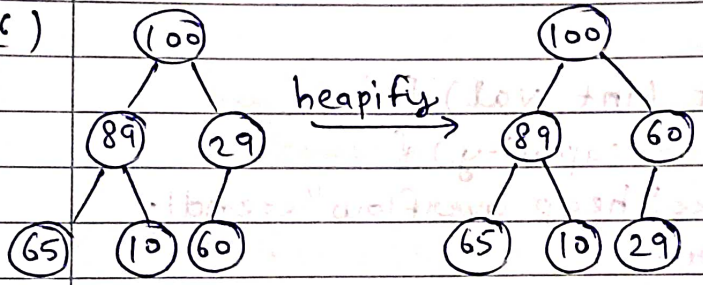
(4)



(5)



(6)



So, the final heap would be

100	89	60	65	10	29
1	2	3	4	5	6

Code-

```
class heap {
    public:
    int * arr;
    int size;
    int capacity;

    heap (int capacity){
        this ->arr = new int [capacity];
        this ->size = 0;
        this -> capacity = capacity;
    }

    void insert (int val) {
        if (size == capacity) {
            cout << " heap overflow " << endl;
            return;
        }

        size ++;
        int index = size;
        arr [index] = val;

        while (index > 1) {
            int parentIndex = index / 2;

            if (arr [index] > arr [parentIndex]) {
                swap (arr [index], arr [parentIndex]);
                index = parentIndex;
            }
            else break;
        }
    }
```

```
int getsize(){
    return size;
}

int getElement(int index){
    return arr[index];
}

};

int main(){

    heap h(10);
    h.insert(5);
    h.insert(10);
    h.insert(15);
    h.insert(6);
    h.insert(25);
    h.insert(50);

    for(int i=1; i<h.getSize(); i++){
        cout<<h.getElement(i)<<" ";
    }
}
```

O/P- 50 15 25 5 6 10

Time Complexity - O(1) insert krne ke liye.
O(log n) correct position par lekar jane ke liye.
Correct position par leke jane ke liye height traverse krni hoti hai and height of CBT is log n.
So, overall T.C. for insertion is O(height) → O(log n).

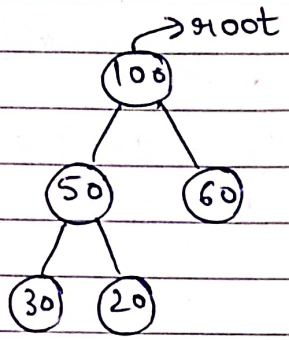
Note- Deletion always root node i.e. top se hota hai. Heap ko hum root node se access kr skte hai always.

* **Deletion in heap -**

Step1 • Replace the element of root node with the last node or the rightmost node of tree.

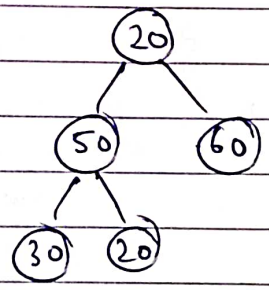
Step2 • Delete the last node.

Step3 • Now, comes heapification

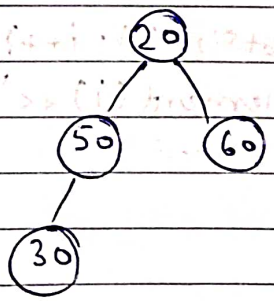


Note- Replacement Last node se isliye ki because last node pr koi other node dependent nhi h or is se heap ki properties bhi disturb ni hori i.e. safest element hai.

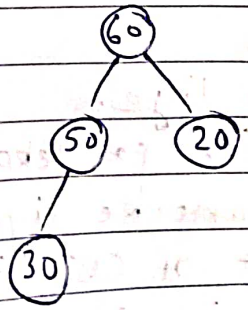
Step 1 -



Step 2 -



Step 3 -



Code -

```
class heap {
    public:
    int * arr;
    int size;
    int capacity;

    heap (int capacity) {
      this -> arr = new int [capacity];
      this -> size = 0;
      this -> capacity = capacity;
    }

int deletefromHeap ( ) {

    int answer = arr[1];
    arr[1] = arr[size];
    size--;

    int index = 1;
    while (index < size) {
        int leftChildIndex = 2 * index;
        int rightChildIndex = 2 * index + 1;

        int largestIndex = index;
        if (leftChildIndex <= size &&
            arr[largestIndex] < arr[leftChildIndex])
        {
            largestIndex = leftChildIndex;
        }
      if (rightChildIndex <= size &&
          arr[largestIndex] < arr[rightChildIndex]
      {
          largestIndex = rightChildIndex;
      }
```

~~if (arr[index] == largestIndex~~

```
if (arr[index] == arr[largestIndex]) {
            break;
        }
else {
    swap (arr[index], arr[largestIndex]);
    index = largestIndex;
  }
}
return answer;
}
};

int main ( ){
      heap h (20);
      h.insert (20);
      h.insert (11);
      h.insert (5);
      h.insert (10);

    cout<<"deleted element:" << h.deletefromHeap()<<endl;
  }
```

O/P - deleted element : 20

Time Complexity - O(1) - for replacement
O(height) = O(log n) for heapifying.

Overall, O(height).

__/__/__

* **Heapifying recursively -**

- Given is array, its size and index to be heapified.

Approach -

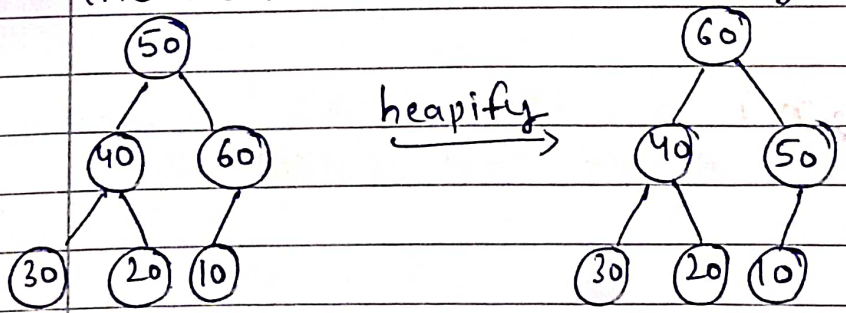
= Find leftChildIndex and rightChildIndex for the given index.

= Find the largest of index, leftchildIndex and rightChildIndex.

= largestIndex is the index of the largest element of the three.

= If index is not equal to largestIndex, then swap arr[index] with arr[largestIndex].
This all is for one case. Rest all is heapified by recursive call.
This will also work as the base case i.e. when index = largestIndex. Here, the condition gets false and base case hits.

Note - The next index to be processed is largestIndex. So, the next recursive call will go for that index.



Code -

```
void heapify (int* arr, int n, int index) {

int leftIndex = 2*index;
int rightIndex = 2*index + 1;
int largestIndex = index;

if (leftIndex <= n && arr[largestIndex] < arr[leftIndex])
 {
    largestIndex = leftIndex;
 }
if (rightIndex <= n && arr[largestIndex] < arr[rightIndex])
 {
    largestIndex = rightIndex;
 }

if (index != largestIndex) {
   swap (arr[index], arr[largestIndex]);
   index = largestIndex;
   heapify (arr, n, index);
 }
}
```

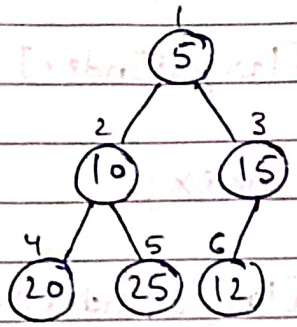
Time complexity - $O(\log n)$

Imp

* **Create heap from array -**

Array =

5	10	15	20	25	12
1	2	3	4	5	6



Here, we are heapifying from top to bottom. That's why leaf nodes valid heap hongi because leaf nodes me aage children honge nhi comparison ke liye.

for a complete Binary Tree, every level is filled except the last level.

So, the leaf nodes will be from indices -

$$\frac{n}{2} + 1 \longrightarrow n$$

where n is the no. of nodes in tree.

This will satisfy for every valid complete Binary tree.

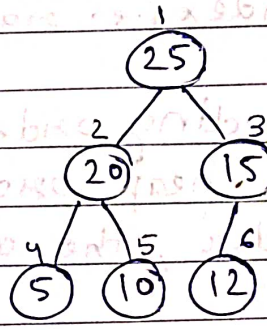
That's why loop whole array pr chalane ki need ni h, bs non-leafy nodes ke liye hi chalani hai. Yha hum hr non-leafy nodes ke indices ko heapify kr rhe honge array me.

Code -

```
void heapify (int * arr, int n, int index){
    int leftIndex = 2* index;
    int rightIndex = 2*index +1;
    int largestIndex = index;

if (leftIndex <= n && arr[largestIndex] < arr[leftIndex])
  {
      largestIndex = leftIndex;
  }
if (rightIndex<=n && arr[largestIndex] < arr[rightIndex])
  {
      largestIndex = rightIndex;
  }

if (index != largestIndex) {
   swap (arr[index], arr[largestIndex]);
   heapify (arr, n, largestIndex);
  }
}

void buildHeap (int arr[], int size) {

   for (int index = size/2; index > 0; index--){
        heapify (arr, size, index);
       }
   }
```

```
int main(){
    int arr[] = {-1, 5, 10, 15, 20, 25, 12};
    int size = 6;
    buildHeap(arr, size);
    for(int i=1; i<=size; i++){
        cout << arr[i] << " ";
    }
}
```

O/P- 25 20 15 5 10 12



Time Complexity - Build Heap function will take $O(n/2) \simeq O(n)$ time and heapifying takes $O(\log n)$ time but it will not take $O(\log n)$ time to heapify for every node in tree using ~~bottom-to-top~~ top-to-bottom approach.

The time complexity will be the summation of heights of the current nodes where limits goes from 1 to $n/2$. This will sum up to $O(n)$.

$$\sum_{i=1}^{n/2} \log_2 i \simeq n$$

Height of a particular node is the distance of the node from ~~root~~ leaf node.

* Heap Sort -

Algorithm -

while size is not equal to 1 i.e. until the array comprises single element, do the following steps -

- swap arr[1], arr[size].
- Reduce the array size by 1.
- Heapify the Ist index i.e. root element.

Note - For heap sort (ascending order), the array should follow max heap property.
For descending order, the array should be min heap.

Code -

```
void heapify (int arr[], int n, int index) {
    int leftIndex = 2*index;
    int rightIndex = 2*index +1;
    int largestIndex = index;

if (leftIndex <= n && arr[largestIndex] < arr[leftIndex])
  {
      largestIndex = leftIndex;
  }
if (rightIndex <= n && arr[largestIndex] < arr[rightIndex])
  {
      largestIndex = rightIndex;
  }
```

```
if (index != largestIndex){
    swap(arr[index], arr[largestIndex]);
    index = largestIndex;
    heapify (arr, n, index);
  }
}

void heapSort (int arr[], int size){

    while (size != 1){
      swap (arr[1], arr[size]);
      size--;
      heapify (arr, size, 1);
    }
  }

int main ( ) {
  int arr [] = {-1, 5, 10, 15, 20, 25, 12};
  int  n = 6;
  buildHeap (arr, n) ; //this function makes the heap
  heapSort (arr, n);
  for (int i= 1; i<= n; i++){
      cout << arr[i] << " ";
    }
  }
```

O/P - 5 10 12 15 20 25

Time complexity - There are n elements in array, and every elements needs to be heapify after swapping. So, overall complexity is $O(n \log n)$.