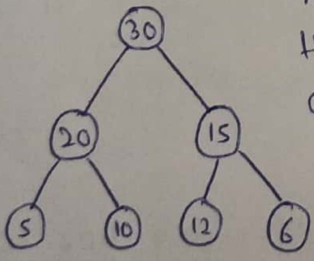
HEAP

★ The very first Condition of a heap is that it is a Complete Binary tree, it means if we fill elements in an array, then there will be no blank spaces left

Reason :- Bcz Binary Heaps are implemented using array, we can also use linked list representation but most of the time algorithms are made using array, so thats why it is a Complete Binary tree and we dont want vacant spaces.

★ Second Condition Contains two things, Maximum Heap and Minimum Heap

Max. Heap :- Every node should have an element greater than or equal to all of its descendents and duplicates are also allowed as this is not a BST.

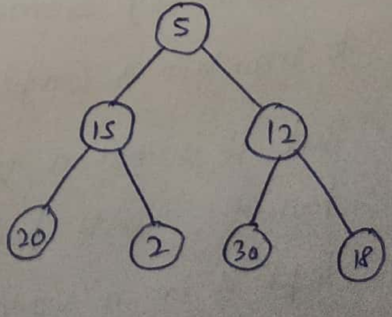


30	20	15	5	10	12	6
1	2	3	4	5	6	7

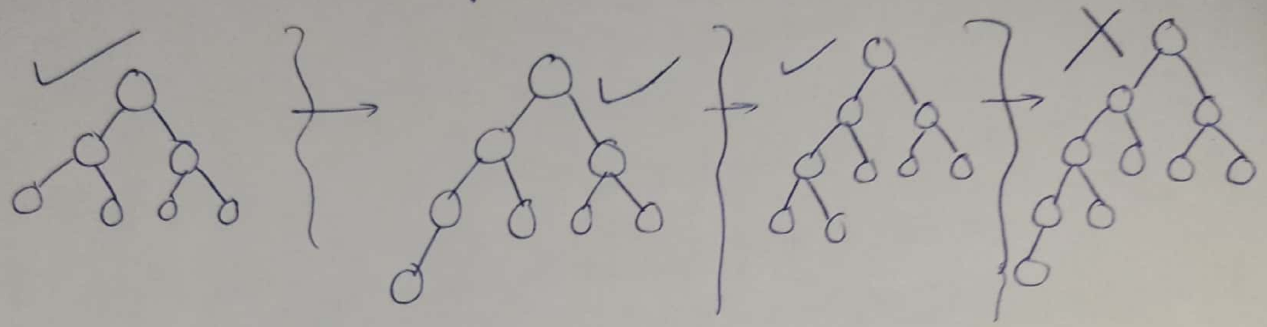
Node at index i
Left child at 2*i
Right child at 2*i +1

Min. Heap

Every node should have an element smaller than or equal to all of its descendants

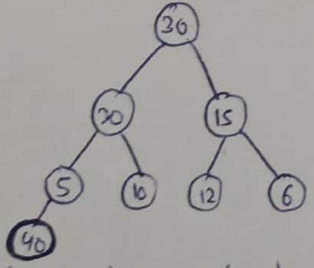


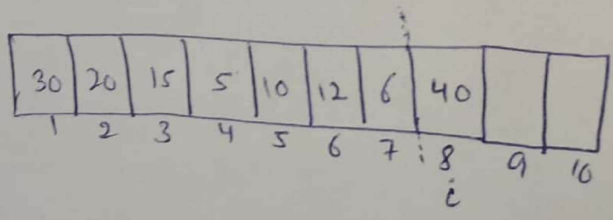
Important Note :- ① As Heap is a Complete Binary tree, so there will be always O(log n) Height of a tree, the Height of a tree will not increase unneccesarily



② Heap is not Using for Searching Purpose.

☆ Inserting in a Heap :- Learn how to insert in a max. Heap and for min. Heap procedure will be same



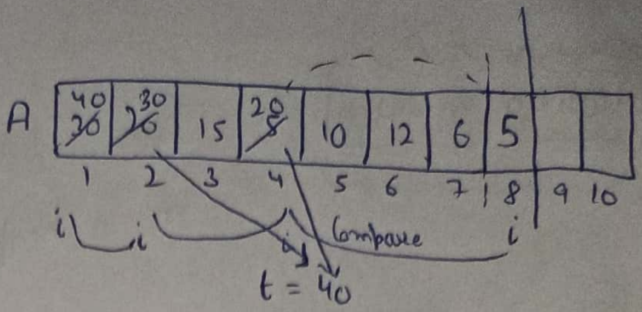


30	20	15	5	10	12	6	40		
1	2	3	4	5	6	7	8	9	10

Let Us insert (40)

- Insert a key element in an array in a next free space bcz we want to maintain a Complete Binary tree, so 40 will be left child of 5 (i=4)
- But it is not a max. heap now, So rearrange the elements with its parent, Now its parent is (5), so it is Smaller so Copy it in below and kept it in a variable and then again Compare Until parent becomes larger or it will end of array (i.e. i=1), else when it is less than parent Copy it on that place.

Now Compare it in Array,



⇒ Program :-

A	40	35	15	30	10	12	6	5	20	
	1	2	3	4	5	6	7	8	9	10

i↑n

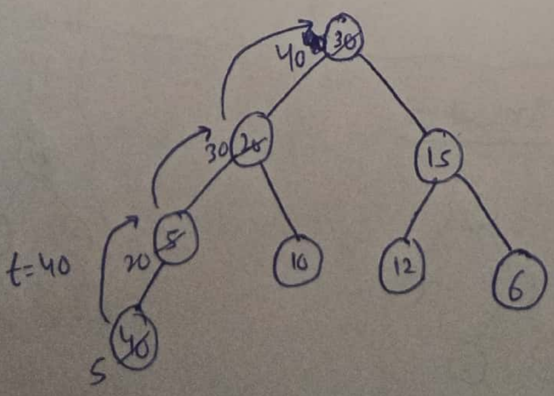
temp = 50

→ Array for heap

[n]→ is the index of recently stored element

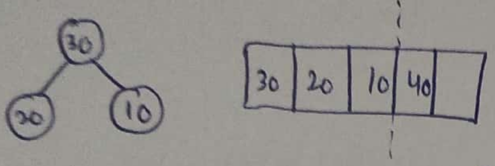
```
Void Insert (int A[], int n)
{
     int temp, i=n;
        temp = A[n];
     While (i>1 && temp > A[i/2])
       {
           A[i] = A[i/2]
             i = i/2;
       }
       A[i] = temp;
}
```

Analysis :- Actually while inserting we are also rearranging the element and it depends upon the height of tree and it is a Complete binary tree, So $\boxed{\text{time Complexity} = O(\log n)}$ as we can also see from the Code as in while loop (condition gets half each and every time)

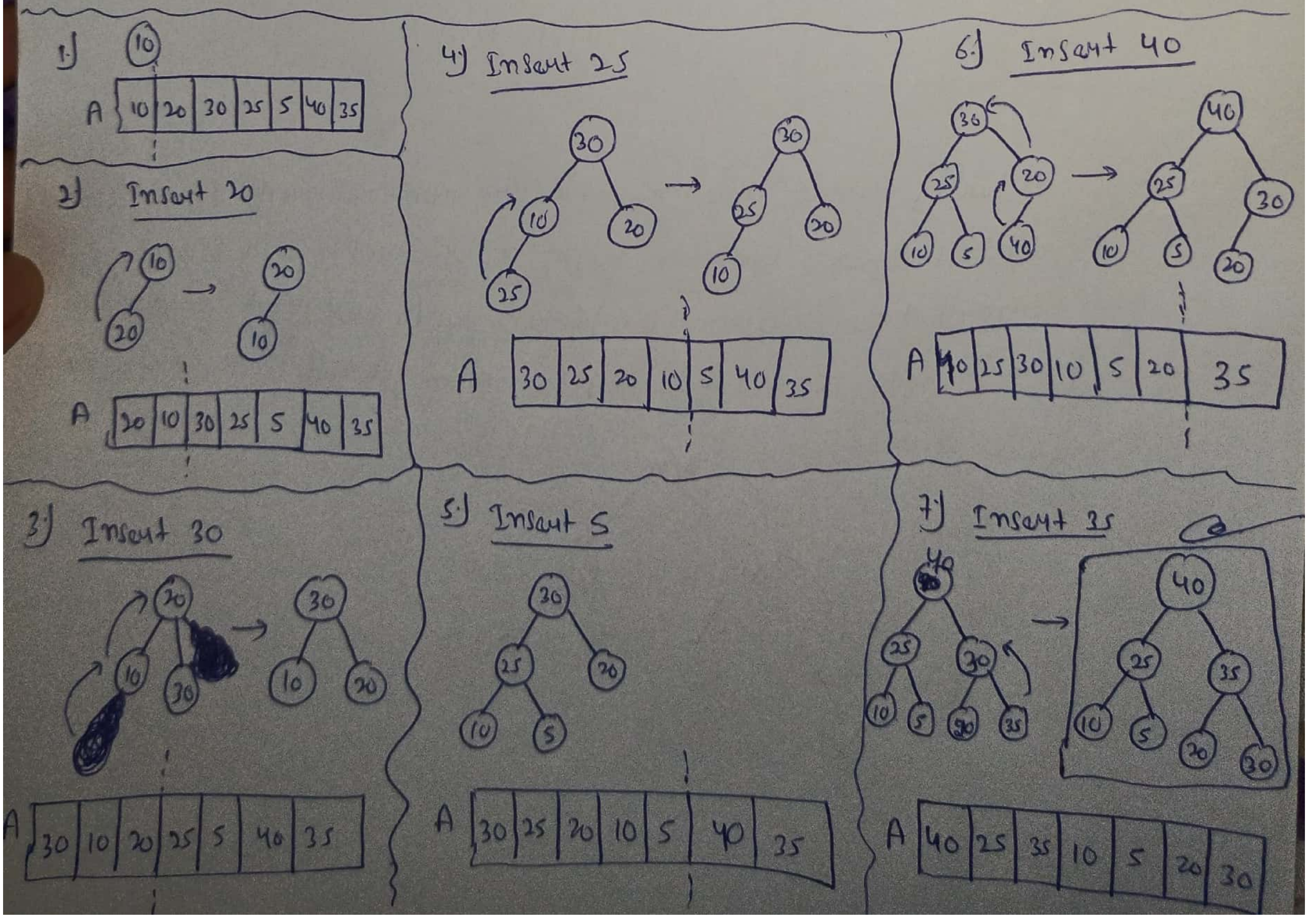


bcz, for creating n el
are inserted and for 1 el
Time Complexity is O(logn)

☆ **Creating a Heap** :- We already have learnt how to insert in a heap [Time Complexity :- O(nlogn)]



Now we know that it is Complete binary tree of 3 elements or it is a heap of 3 elements, so Now we will insert 40 and it will arranged in it to form as a heap, so (40) is not in a heap before but it is present in an array, so we will insert 40 and more elements one by one, so it means the 3 elements (30,20,10) are also inserted like this, So we don't need an extra array to form a heap, within the array elements will be adjusted to form a heap, so thats why it is called "inplace heap Creation".



1.) 10
A 10 | 20 | 30 | 25 | 5 | 40 | 35
2.) Insert 20
A 20 | 10 | 30 | 25 | 5 | 40 | 35
3.) Insert 30
A 30 | 10 | 20 | 25 | 5 | 40 | 35
4.) Insert 25
A 30 | 25 | 20 | 10 | 5 | 40 | 35
5.) Insert 5
A 30 | 25 | 20 | 10 | 5 | 40 | 35
6.) Insert 40
A 40 | 25 | 30 | 10 | 5 | 20 | 35
7.) Insert 35
A 40 | 25 | 35 | 10 | 5 | 20 | 30

★ Code for inserting | creating in heap :-

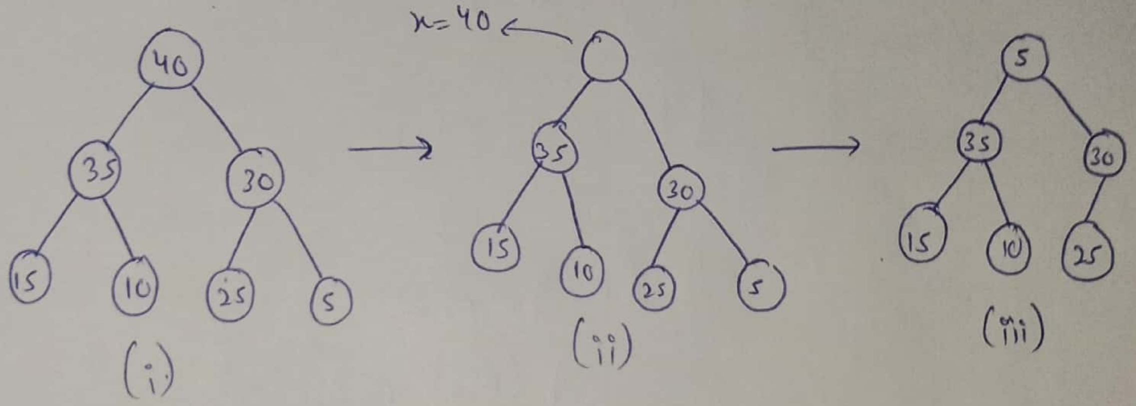
```
int main ( )
{
    int n, i ;
    Scanf ("%d" , &n);
    int A[n];
    printf (" Enter the element in Array");
    for (i=1 ; i<=n ;i++)
     {
        Scanf ("%d", &A[i]);
     }
    Void insert (int A[], int n); || we can insert upto any
                                     index depending upon our
                                     wish and make a heap
    || insert (a,1);
    || insert (a,2);
    || insert (a,3);
    || insert (a,4);
    || insert (a,5);
    || insert (a,6);
    || insert (a,7);
    for (i=1 ; i<=n ; i++)
     {
        insert (a,i) ;
     }
```

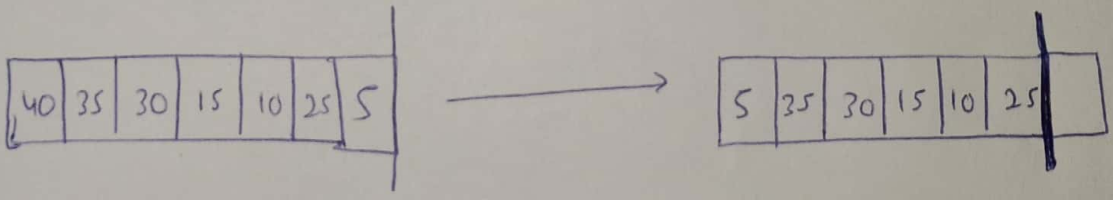
Scanned with CamScanner

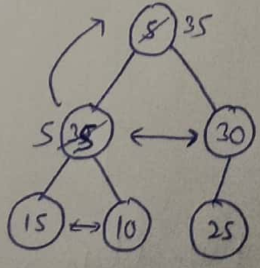
```
for (i=1; i<=n; i++)  // for printing heap
  {
     printf ("%d", A[i]);
  }
}

Void insert (int A[], int n)  // for Max. Heap
 {
     int temp , i=n;
      temp = A[n];
     while (i> 1 && temp > A[i/2])
       {
            A[i] = A[i/2];
              i= i/2
        }
       A[i] = temp;
 }
```

Deleting from Heap and Heap Sort :-

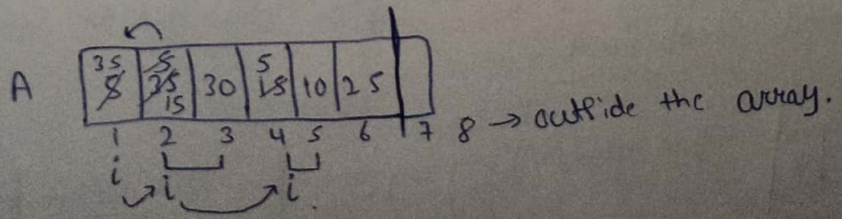
When we want to delete any element from a Heap we can delete only root value. and in the place of root, Copy the last element of the Complete Binary tree





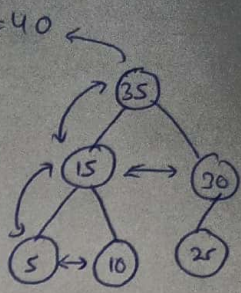


Now, To rearrange the elements,
first Compare the Children of new root,
than which will be larger, Compare it with the root and if greater than interchange, then again Compare its children, which will be larger, Simply interchange and stop when a index has not L R child.

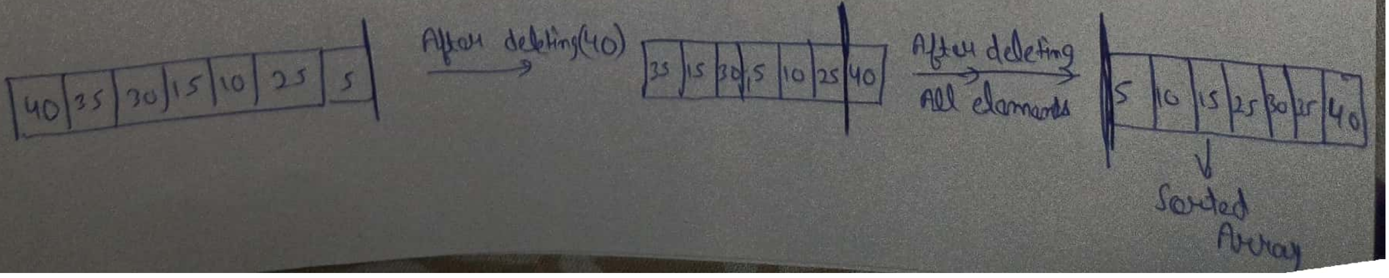


Heap

Program :-
```
Void Delete (int A[], int n)
{
        int x, i, j;
        x = A[1];
        A[1] = A[n];
        i=1 ; j = 2*i;
        while ( j < n-1)
        {
              if (A[j+1] > A[j])
                    j = j+1 ;
              if (A[i] < A[j])
              {
                  swap (A[i], A[j]);
                      i= j;
                      j = 2*j;
              }
              else
                  break;
        }
        A[n] = x;    ———— ⊛ [Copy element at deleted Space]
}
```



Now, if we delete an element from Heap, we will left will left with the Space at the end of the array, So there we will kept our deleted element and if we going on deleted the elements, we will get an Sorted Array and this is known as Heap Sort.



Heap Sort :-

① Create a Heap of 'n' elements
② Delete 'n' elements 1-by-1

So for Heap Sort, we have to know about how to insert and delete in a Heap

⇒ **Analysis** ⇒ Creating a Heap → Time Complexity :- n logn
Deleting from Heap all 'n' → Time Complexity :- n logn

$$\boxed{\text{So, Total Time of Heap Sort :- } 2n\log n = O(n\log n)}$$

☆ Code for Heap Sort :-

```
     int main ( )
    {
       int n, i ;
       Scanf (" %d", &n) ;
       int A[n] ;
       for (i=1 ; i<=n ; i++)
         {
           Scanf (" %d" , &A[i]) ;
         }
       Void insert (int A[] , int n) ;

        for (i=1  ; i<=n ; i++)
          {
            insert (A, i) ;
          }
```

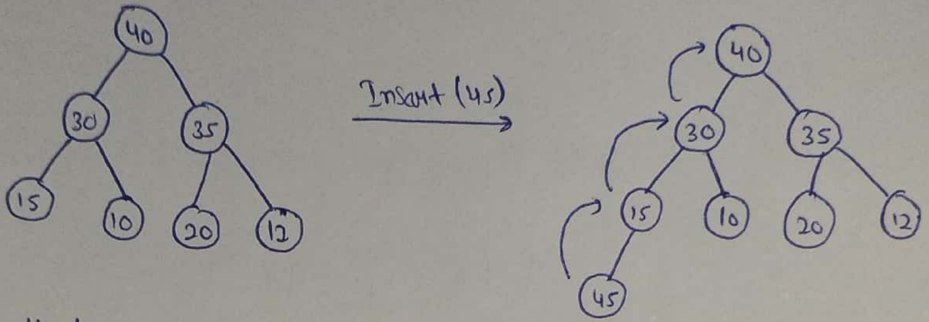
```
int delete (int A[], int n);
// printf (" Deleted value is %d \n", delete (A,n));
// printf (" Deleted value is %d \n", delete (A, n-1));
// printf (" Deleted value is %d \n", delete (A, n-2));
// printf (" Deleted value is %d \n", delete (A, n-3));
// printf (" Deleted value is %d \n", delete (A, n-4));
// printf (" Deleted value is %d \n", delete (A, n-5));
// printf (" Deleted value is %d \n", delete (A, n-6));
// printf (" Deleted value is %d \n",
 for (i = n ; i >= 1 ; i--)
  {
   delete (A, i);
  }
  printf (" Sorted heap we get is : \n");
  for (i = 1 ; i <= n ; i++)
   {
    printf (" %d ", A[i]);
   }
}
void insert (int A[], int n)
{
   int temp , i = n;
   temp = A[n];
   while (i > 1 && temp > A[i/2])
    {  A[i] = A [i/2];
         i = i/2;
    }
     A[i] = temp;
}
```

```
int delete (int A[], int n)
{
    int i, j, x;
     x = A[i];
     A[1] = A[n];
     A[n] = x;
      i = 1;
      j = i * 2;
     while (j < n-1)
      {
          if (A[j+1] > A[j])
               j = j+1;
          if (A[i] < A[j])
           {
               int temp = A[i];
                A[i] = A[j];
                A[j] = temp;
                 i = j;
                 j = 2 * j;
           }
          else
           break;
      }
     return x;
}
```

★ **Heapify** :- This Procedure is related to creation of Heap and we can say it is a faster method for creating Heap.

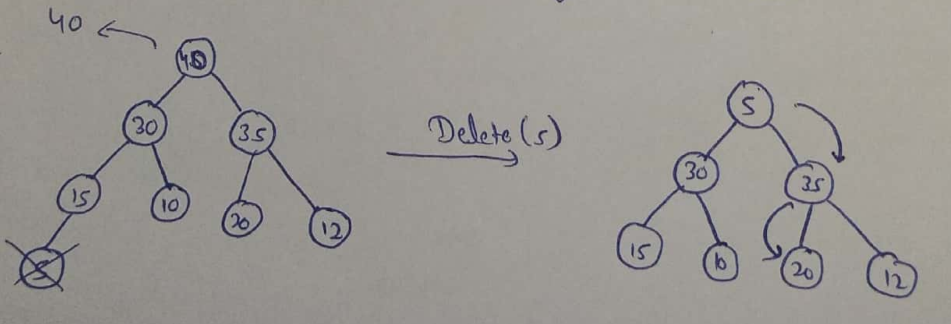
Now, To Understand Heapify lets Understand Something

⇒ **Insertion** :-



We can see that after inserting 45, we get a Complete binary tree but not a max heap, So make it as max heap, we will Compare the inserted element with its parent, So the imp. thing is "in insertion elements are adjusted by Sending from leaf to root node".

⇒ **Deletion** :-



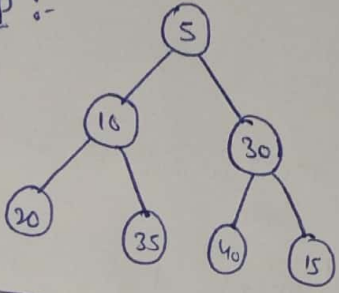
We can see that After Deletion we get a Complete binary tree but not max Heap, So make it as max Heap, we will Compare the root with its descendants, So the imp. thing is "in deletion elements are adjusted by Sending from root to leaf node".

We already know, when we are creating a Heap simply it will take time, that is

Time for insertion :- O(n log n)

⇒ Simple Create Heap :-

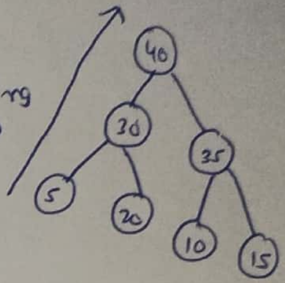
By sending it towards root.



A

5	10	30	20	35	40	15
1	2	3	4	5	6	7

$\xrightarrow[\text{in a Heap}]{\text{After inserting}}$

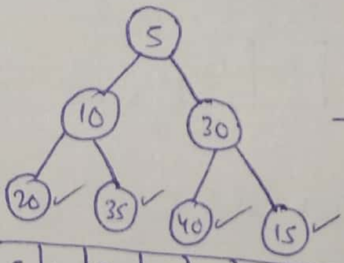


A

40	30	35	5	20	10	15
1	2	3	4	5	6	7

Max. Heap

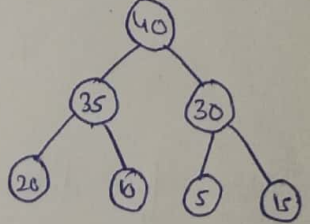
⇒ Heapify :-



A

5	10	30	20	35	40	15

$\longrightarrow$



A

40	35	30	20	10	5	15

We know, during Simple Creation of Heap, we say (5) is already in heap and we were moving from left to right in an Array, but in Heapify we will move from Right to left, so, now we will not compare with parent or move from leaf to root as in simple heap creation, we will do from root to leaf (compare with children). So, now for 15, 40, 35, 20 there are no children for them, so, they are already in heap. Now come on 30, compare with children and same for (5) then.

Note :- Max. Heap formed by simple creation & Heapify will be different but both will be the max. Heap [same for Min. Heap].

Time :- O(n)

As adjustments get half, bcz we leave leaf nodes.

☆ Heap as Priority Queue :-

It's related to insertion and deletion of elements, such that Elements are inserted acc. to their priorities and the highest priority element will be deleted first.

elements → 4, 9, 5, 10, 6, 20, 8, 15, 2, 18

"Larger the element, higher the priority"
or
"Smaller the element, higher the priority". → we can select anyone before Hand.

Let us select, Larger the element, Higher the priority.

- Now, Let us insert the elements in an Array

A	4	9	5	10	6	~~20~~	8	15	2	18
	1	2	3	4	5	6	7	8	9	

Now, for deletion, we have to search the max. element from an Array which takes $O(n)$ time and then we have to shift also (bcz we dont want vacant space in array)

So, Insert → $O(1)$
Delete → $O(n+n) \Rightarrow O(2n) = O(n)$

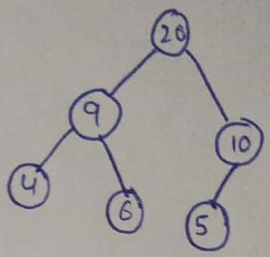
We can also do in another way, Such that Store the elements in Sorted form and then delete the last element, but here

Insert → $O(n)$
Delete → $O(1)$

So, Any of the process will take $O(n)$ time, But we can do in another way to reduce the time.

Solution :- Elements $\rightarrow \underbrace{4, 9, 5, 10, 6, 20, 8}$, 15, 2, 18

Make a max. heap of these elements :-



We know, for insert it will time = O(log n).

And Now, for deletion, We know, root will be deleted first and in max. Heap, root value is max. (i.e Having highest priority)

So, Deletion also takes time = O(log n).

So, We can See Insertion & deletion both takes O(log n) which is always less than O(n).

" So, Thats how, Heap is best to implementing a priority Queue"

Note :- We can also use min. Heap, if We are saying that Smaller the no., Higher the Priority.

[" Hence, Heaps are used to implement Priority Queues."]

Scanned with CamScanner