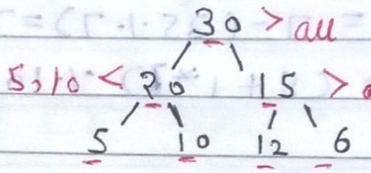
Heap

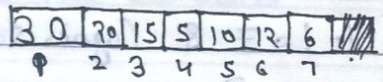
1. It is a complete Binary Tree, there should be any place vacant in between
2. Duplicates are also allowed
3. Every node should have element greater than or equal to its descendents if this is condn this is called **Max Heap**



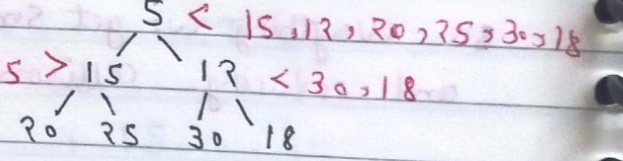
Node at index i

Left child at 2*i

Right child at 2*i+1



4. Every node can have element smaller than equal to its all descendents. This is called **Min Heap**.
5. Heap is represented generally using array. A Heap will be either Max Heap or Min Heap.



6. Height of complete Binary tree will always be $\log n$ and will not T un-necessarily.
7. Heap is not used for searching purpose.

Insertion in Heap

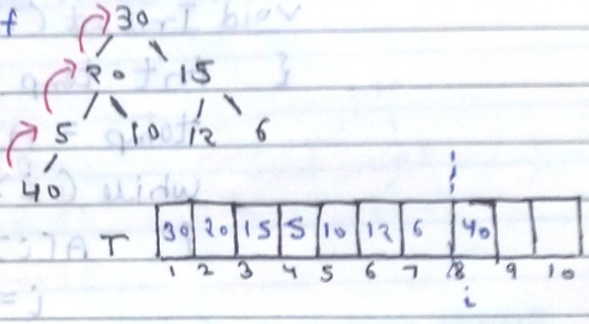
We are taking example of max Heap and someway is applied on Min Heap also

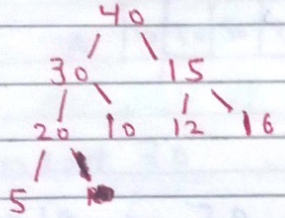
In Heap we always insert in next free space so as to maintain a complete Binary Tree

In complete Binary Tree (CBT) we fill element from Left to Right

if 40 is to be inserted we insert it at Left of 5

But it does not follow property of max Heap so move it up till it become Max Heap, Rearrange it to make Max Heap.





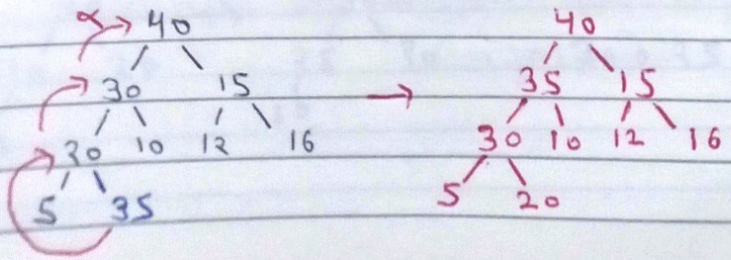
In array we know Left child is at 2i so

i=8, parent of 40 will be at i=4 i.e 5

Compare & Replace if 5 < 40 now Parent of i=4 will i=2 i.e 20 so check & replace now again check for parent of i=2 i.e i=1 check & compare so new Array is 40, 30, 20, 15, 5, 10, 12, 6,

Heap size is now changed to 8

Insert 35



T

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

(comparing i/2 and replacing)

Heap size = 9 now

Program to insert in Heap

n = index of recently inserted element

40, 35, 15, 30, 10, 12, 6, 5, 20

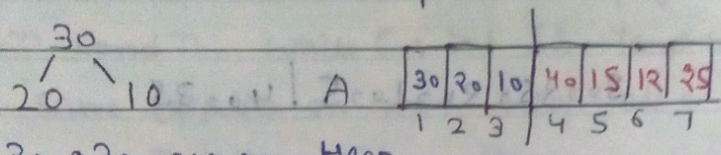
↑ n

So after passing value n in Heap we call

```
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;
}
```

Time = O(log n) because while rearranging the inserted element can max go upto root. so depend on Height of tree.. From loop also we can see i is divided by 2 so $\log_2 n$ time.

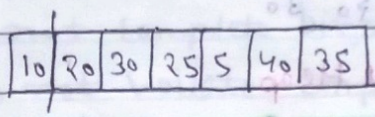
How to create a Heap



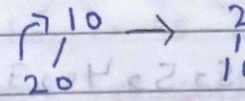
10, 20, 30 are on Heap

but 40, 15, 12, 25 are on Array but not on Heap so we insert & rearrange them one by one using Insert func^n so we don't need any extra array to insert in Heap as we insert Heap size T and elemt insert so when insertion is in same array we call is Inplace

Let create a Heap with first element as 10

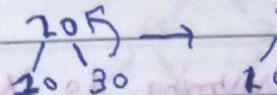


Insert 20



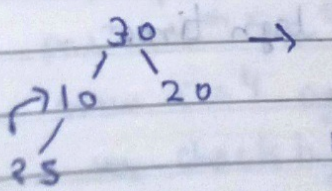
10, 20 | 30, 25, 5, 40, 35

Insert 30



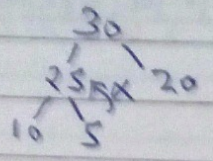
10, 20, 30 | 25, 5, 40, 35 = 30, 20, 10 | same

Insert 25



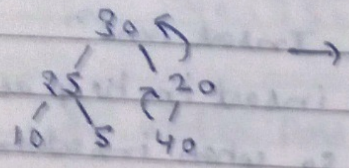
30, 25, 20, 10 | same

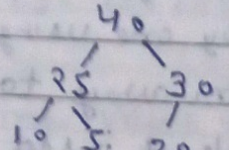
Insert 5



30, 25, 20, 10, 5 | 40, 35

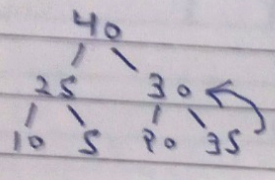
Insert 40

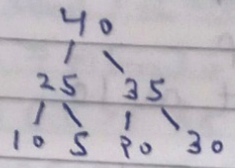


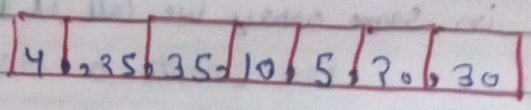


40, 25, 30, 10, 5, 20 | 35

Insert 35







Final Heap

Whole procedure is done on same Array so Heap is created Inplace.

```
Void CreateHeap()
{ int A[] = {0, 10, 20, 30, 25, 5, 40, 35};
  int i;                    0   1   2   3   4   5   6   7
  for(i=2; i<=7; i++)
    { Insert(A, i);
    } // Start from index 2 assuming 10 is
        already in Heap.
}
```

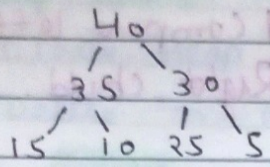
Time = n element X each insertion take log n time

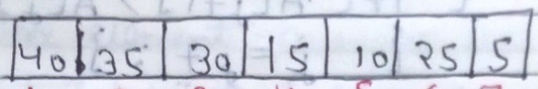
Time = n log n = O(n log n)

If we want to check for min Heap just put insert func^n cond^n while (i > 1 && temp < A[i/2])
rest same applies on min Heap also

Deletion from Heap

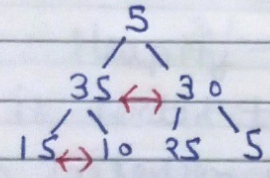
From Heap we can Delete only Root (largest elemnt in case of Max Heap) i.e we can Delete only max Priority element



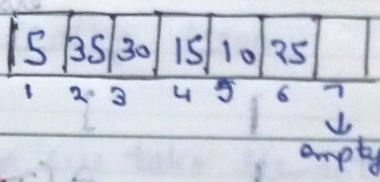


Delete 40

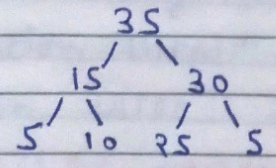
In place of 40 if we place 35 then 15 take pos^n of 35 and it is not complete binary tree but insert 5 in pos^n of 40 does not voilate Complete Binary tree cond^n.



now rearrange it to make it max Heap



Compare both children of 5 and interchange with greater child



Someway in Array compare i with 2i (Left) & 2i+1 (Right child of i) and then compare bigger child (as max Heap) with parent i and replace & now check 2i & 2i+1 again for new pos^n of i i.e 2nd pos^n of 5 now check 4, 5 pos^n 4th > 5th so bring 15 to 2nd pos^n & 5 to 4th pos^n now i = 4 again check if valid index. We take i for checking & replacing.

```
void Delete(int A[], int n)
{   int x, i, j;
    x = A[n];
    A[1] = A[n];
    i=1; j=2*i;
    while (j < n-1)
    { if(A[j+1] > A[j])  // Compare left & Right child
      { j = j+1; }
      if(A[i] < A[j])
      { swap(A[i], A[j]);
        i = j;
        j = 2*j;
      }
      else { break;
           }
    }
    A[n] = x;  // Heap sort cond n
}
```

As we are deleting element from Heap, Heap size ↓ing and blocks in Array becoming vacant. We can use those blocks to store deleted elements as they will be part of Array only not a part of Heap. So this way we get A sorted Array and this method of sorting is known as Heap Sort. (in ascending order)

Heap Sort

1. Create Heap of 'n' element
2. Delete 'n' element 1 by 1 and store in Array (same)

Time

Creating Heap take $n \log n$ & Delete take $n \log n$ as n element are being deleted so

Time = $n \log n + n \log n = 2n \log n = O(n \log n)$

Heapify - Faster Method of Creating Heap

It is related to creation of Heap

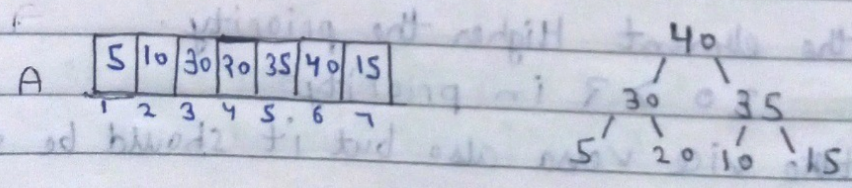
In Insertion in Heap During rearrange we take element from leaf to root

In Delete During rearrangement we take element from Root to leaf

Insertion Direcⁿ - leaf towards Root ($n \log n$ time)

Deletion Direcⁿ - Root towards leaf

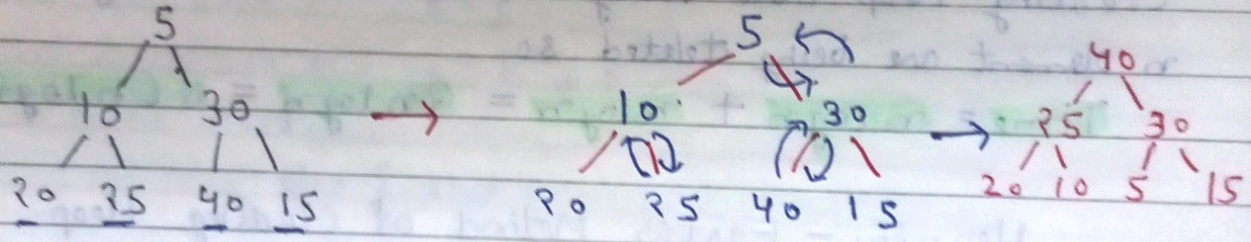
Direcⁿ of adjustment is different.



we assume during insertion that 5 i.e first element is already there and then we insert others but in Heapify we take last element i.e 15

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

but now we will not move from leaf to Root but we will move downward like in deletion element will not be compared with Parent but with children. and if children is bigger than element simply Replace.



Time = O(n). (reduced)
(Faster method).

Binary Heap as Priority Queues

Priority Queue means During insertion elements will have some priority and during deletion Highest priority element will be deleted first.

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

Larger the element Higher the priority.

20 > 2 in priority

we can take vice versa also but it should be decided beforehand.

Insertion

4	9	5	10	6	20	8				

Delete

First search then Delete then fill vacant space i.e shifting

Insert = O(1)

Delete = Searching O(n) + filling vacant O(n) = O(2n)
= O(n)

Q Can we somehow make Deletion O(1) ?

A= Yes, store them sorted order then delete but than insertion will take O(n) So anyone of operation will always take O(n)

We know we can delete Highest priority element only from Heap So Heap help us in implementing Priority queue
We know in Heap Insert = O(log n) Delete = O(log n) << O(n)
So Heap is best Data structure to implement Priority Queue otherwise we can use Array as above but it takes more time
If in our Priority Queue smallest element has Higher priority then use Min Heap