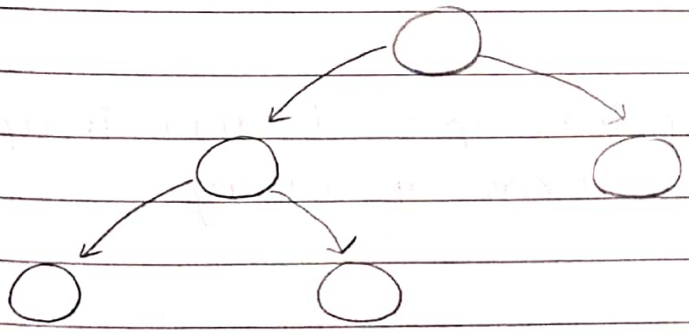
20/05/2023

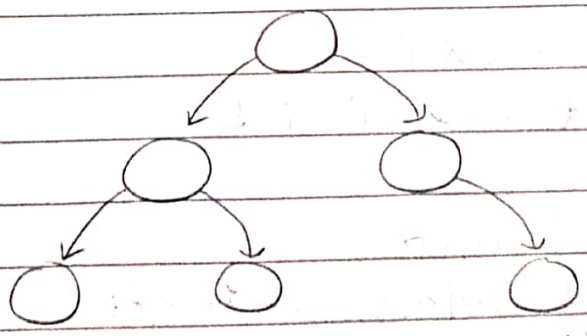
Heaps

Heap is a data structure which exists in form of complete binary tree & follows a property which is known as heap property.

Complete binary tree has all the levels completely filled (except last one) & the filling is done from left to right.



The above is an example of complete binary tree.

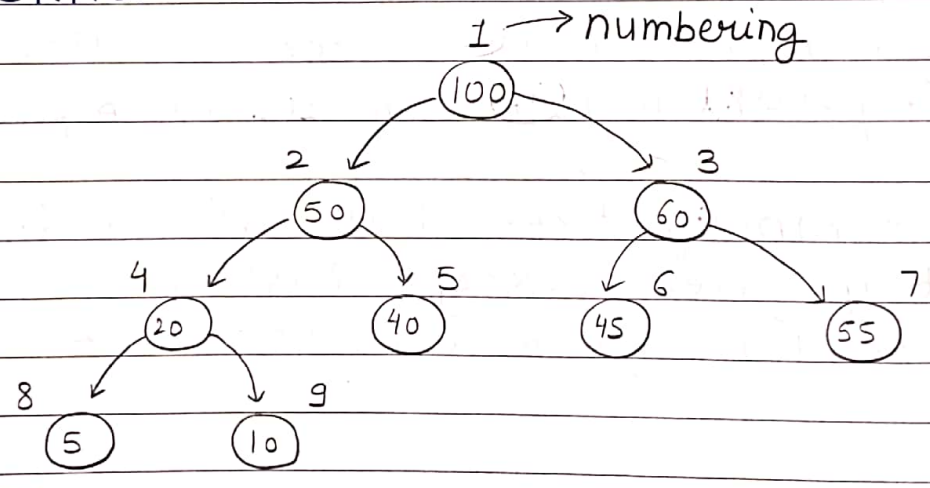


The above is not a complete binary tree as filling is not done from left to right.

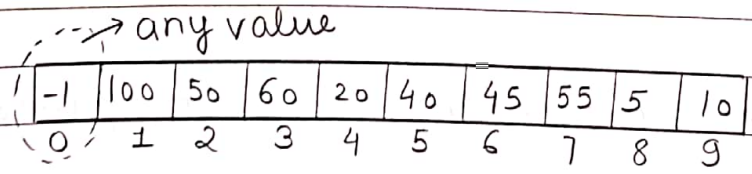
Heaps are created in form of array but visualization is done in form of trees.

Heap property

1) Max heap → Parent is bigger than the child.
2) Min heap → Parent is lesser than the child.



The above is an example of max heap.
Lets store it in form of array



Parent = i
Left - child = $2 \times i$
Right - child = $2 \times i + 1$

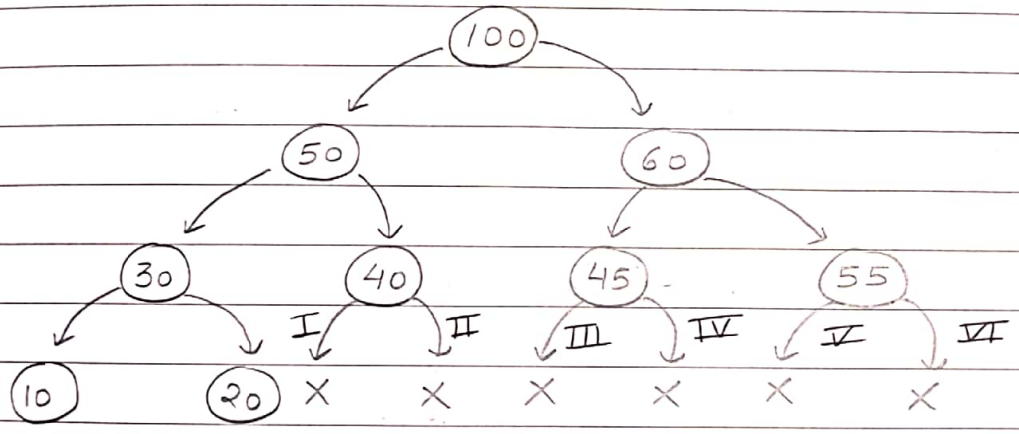
100 at 1st index
50 at 2nd index $(2 \times 1 = 2)$
60 at 3rd index $(2 \times 1 + 1 = 3)$

The above formulae is based on 1-based

indexing

index = i , then its parent is $\frac{i}{2}$.

Insertion in heap



Suppose we want to insert 57 in the max heap given.

1) As filling is done from left to right so at I place the element is inserted i.e. at the end of the array. But 57 > 40 & it violates the property of max-heap.
2) Now place 57 at right position. Compare 57 with the parent.

57 > 40 (True) & hence replace 57 & 40.

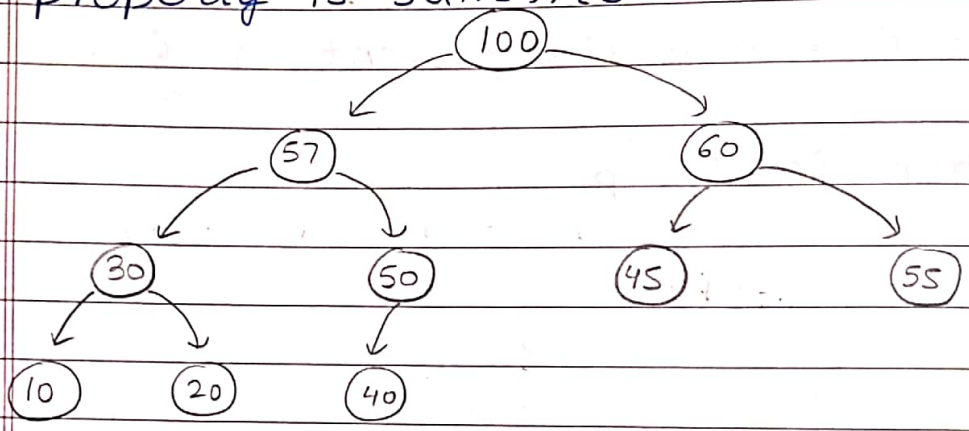
Now compare 57 with the parent 50 & again

57 > 50 (True) & hence replace 57 & 50.

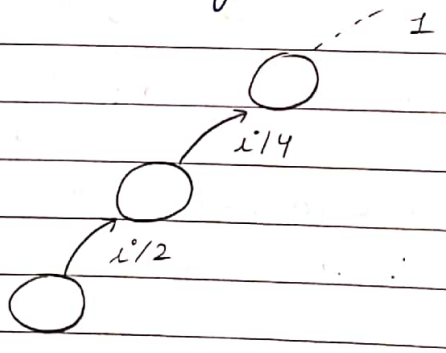
Now compare 57 with parent i.e 100 but here

57 > 100 (False).

Final heap becomes in which max heap property is satisfied.



Time complexity = O(logn)



$\frac{i}{2^k} = 1$

$2^k = i$

$k = \log_2 i \quad (\log_2 n)$

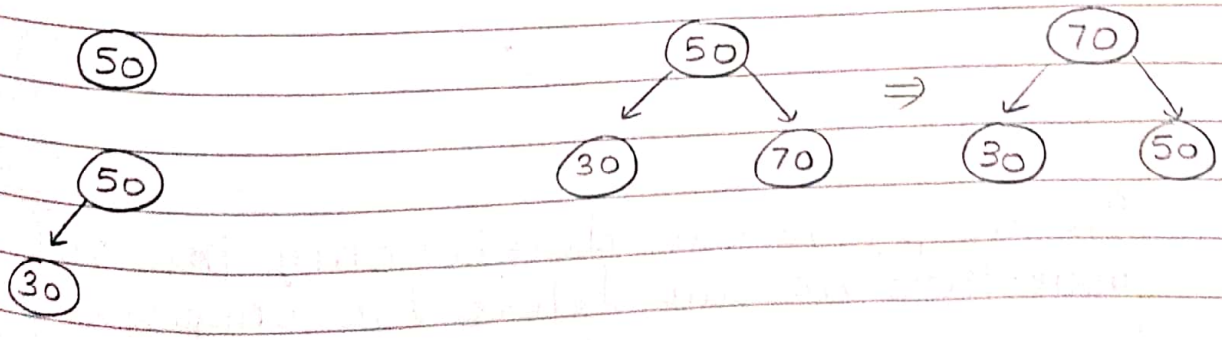
Code

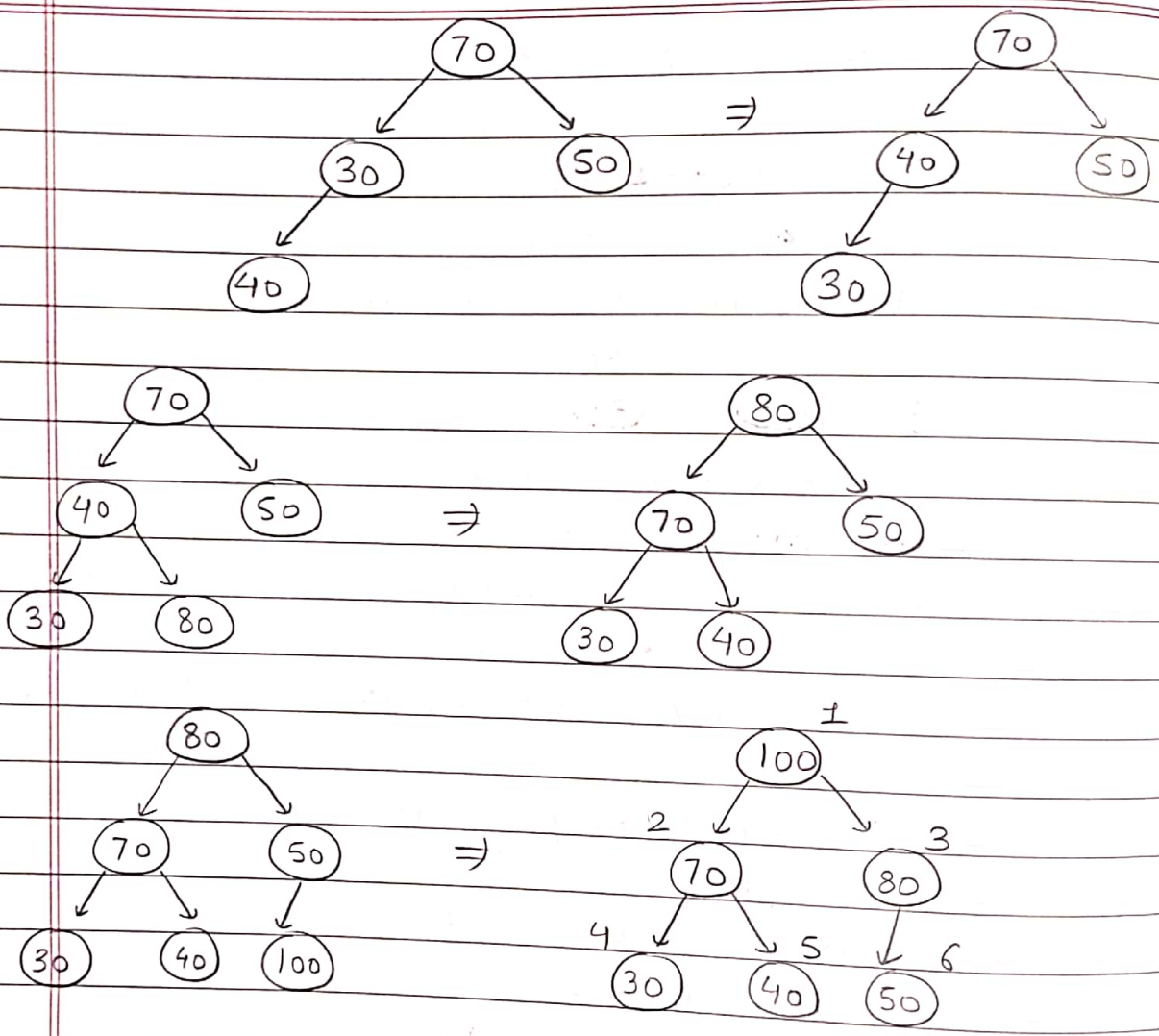
```
Class Heap {

    public :
        int arr[101];
        int size;
        Heap() {
            size = 0;
        }
```

```
void insert (int value) {
    //Insert at index
    size = size + 1;
    int index = size;
    arr [index] = value;
    //Place value so that heap property
    is satisfied
    while (index > 1) {
        int parent = index / 2;
        if (arr [parent] < arr [index]) {
            //Swap
            swap (arr [parent], arr [index]);
            //update index
            index = parent;
        }
        else { //Do nothing
            break;
        }
    }
}
};
```

Creating heap with given numbers

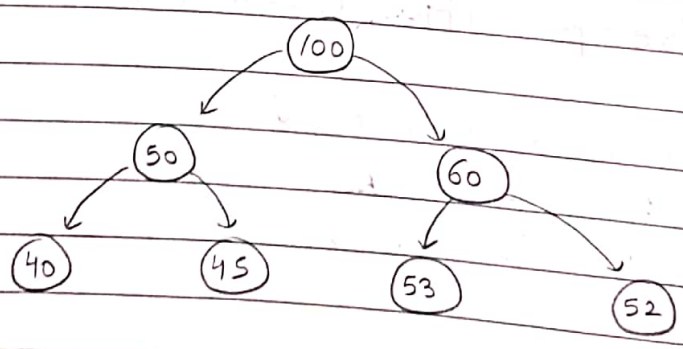
Ex→ 50, 30, 70, 40, 80, 100





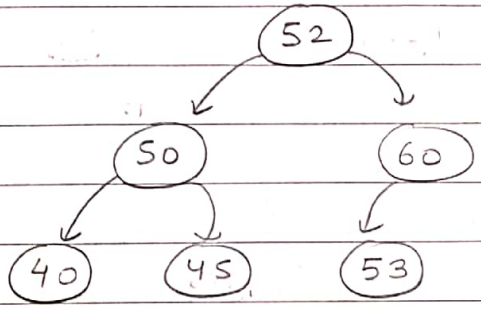
Ans ⇒ -1 , 100 , 70 , 80 , 30 , 40 , 50
0 1 2 3 4 5 6

Deletion in heap



In heap, we can delete only the root node. Here we can delete 100 only.

1) Replace last value with root node



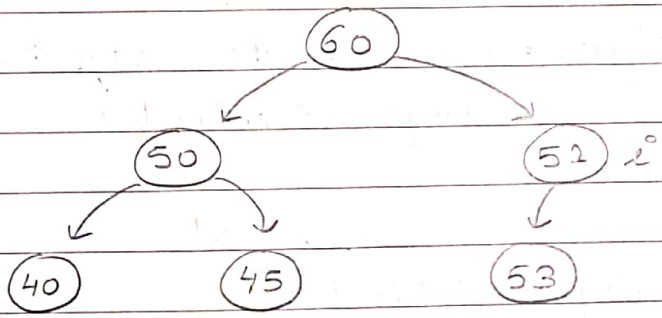
2) Now place 52 at right position.

i 2i

max (52, 50, 60) = 60

2i+1

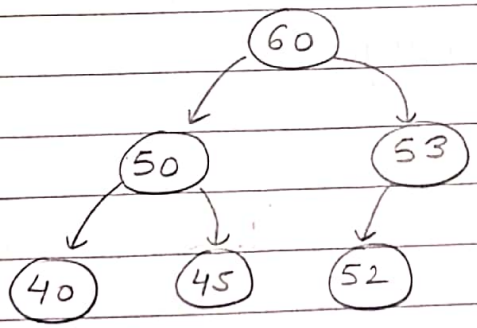
Replace 52 and 60.



Now i is at 52.

max (52, 53) = 53

Replace 52 and 53.



TC = O(logn)

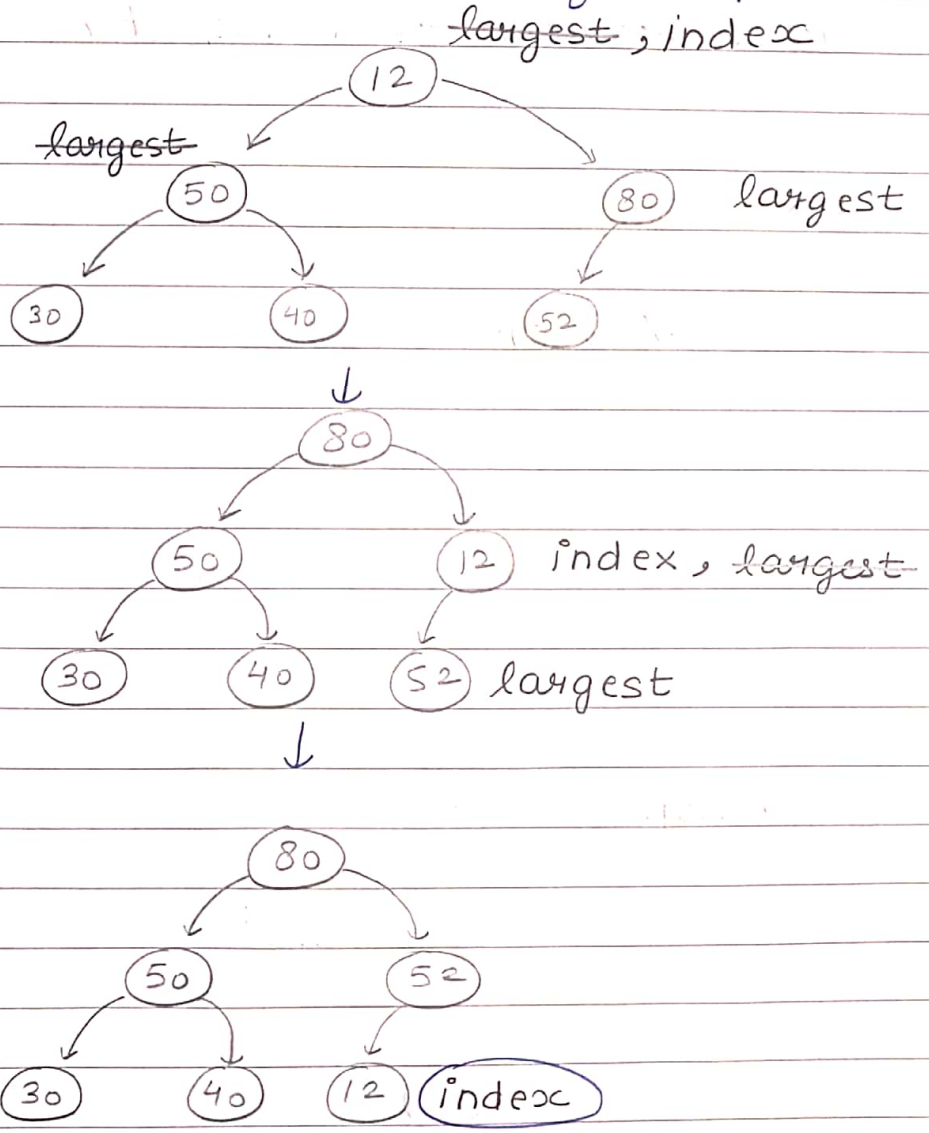
Code

```
//Will delete the root node only
int deleteV() {
        int ans = arr[1];
```

```
//Replace root node & last node
arr[1] = arr[size];
size--; //Delete last value
//Heap property to be satisfied
int index = 1;
while(index < size) {
    int left = 2 * index;
    int right = 2* index +1;
    //Find max. value among parent & child
    int largest = index;
    if(left <= size && arr[largest] < arr[left]){
        largest = left;
    }
    if(right <= size && arr[largest] < arr[right]){
        largest = right;
    }
    //Largest not updated
    if(largest == index)
        break; //Property satisfied
    else{
        swap(arr[index], arr[largest]);
        //Update looping variable
        index = largest;
    }
}
return ans; //Value that got deleted
}
```

Heapify (Asked a lot in D.E. Shaw)
Convert any array to the heap with the help of heapify function.

Note → Using max-heap, largest value in an array can be retrieved via the 1st index in O(1) time. This is the use case of heap.



Code

```
void heapify (int arr [], int n, int i) {
    int index = i;
    int left = 2 * i;
    int right = 2 * i + 1;
    int largest = index;
```

valid index

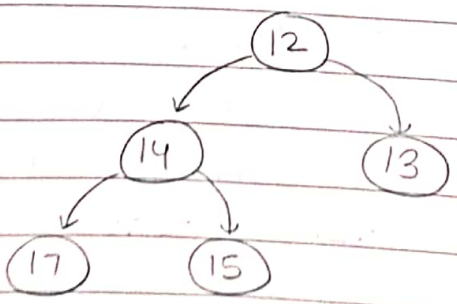
```
if (left <=n && arr[largest] < arr[left]){
        largest = left;
}
```

valid index

```
if (right <=n && arr[largest] < arr[right]){
        largest = right;
}
// Check largest updated or not
    if (largest != index) {
        //swap
        swap(arr[largest], arr[index]);
        index = largest; //update
        //recursive call
        heapify(arr, n, index);
    }
}
```

Build heap

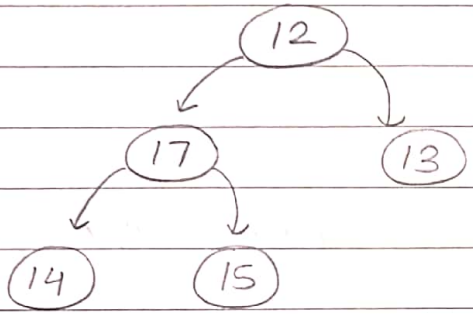
-1	12	14	13	17	15
0	1	2	3	4	5



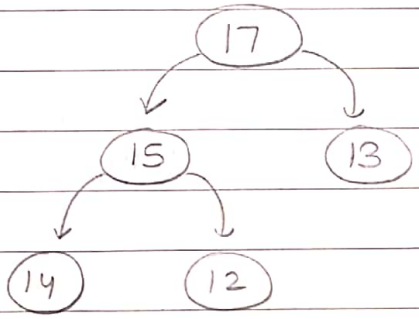
Lets start from last index i.e 5. We see 15, 17, and 13 is max heap. From here we can make a conclusion that we don't have to heapify the leaf nodes.

Now we are at 2nd index i.e 14.

Now here not a max-heap. Hence ca ll heapify.



Now we are at 1st index & it is not max heap. Hence call heapify.



Hence max-heap has been constructed.

Note → It is important to note that nodes from $\frac{n}{2}+1$ to n will be leaf nodes.

$\frac{n}{2}+1$ to n → no heapify required

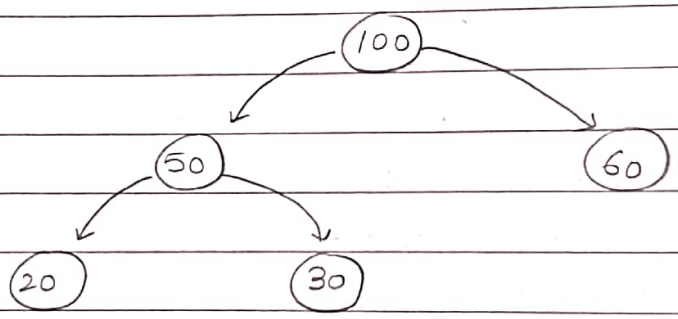
1 to $\frac{n}{2}$ → heapify required
→ start heapify from this node

Code

```
void buildHeap (int arr[], int n){

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

Time complexity = O(n)

Heap sort

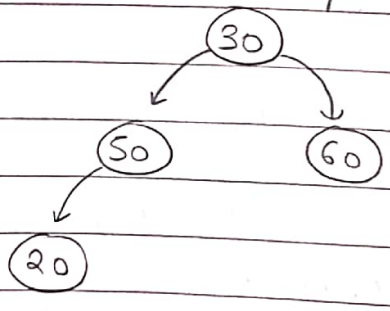


100, 50, 60, 20, 30

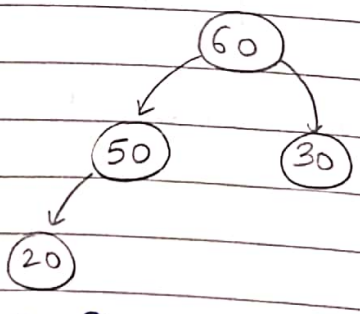
1) Swap first and last element. We get

30, 50, 60, 20 | 100

heap | sorted



2) Now call heapify for 30.

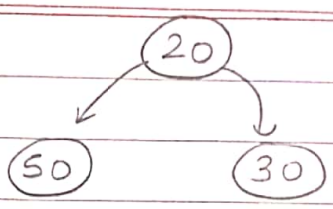


60, 50, 30, 20

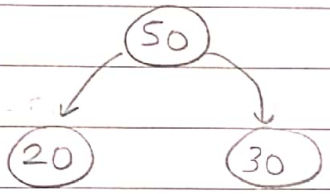
3) Swap 60 and 20

0, 0, 0 | 60, 100

heap | sorted



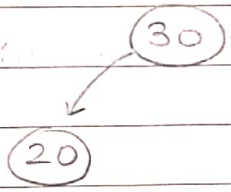
4) Call heapify for 20.



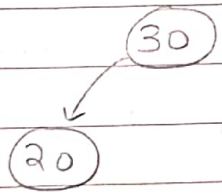
50, 20, 30, 60, 100

5) Swap 50 and 30

30, 20 ; 50, 60, 100
heap sorted



6) Call heapify for 30.



30, 20, 50, 60, 100

7) Swap 30 and 20.

20, 30, 50, 60, 100
sorted

20 is single element & is already sorted.
Hence sorted array is
20, 30, 50, 60, 100

Code

```
void heapSort (int arr[], int n) {
    // n is valid index in 1-based indexing
    int index = n;
    while (n != 1) {
        swap (arr[1], arr[index]);
        index--;
        n--;
        // Apply heapify on 1st index element
        heapify (arr, n, 1);
                          ↳ mistake occurs
    }
}
```