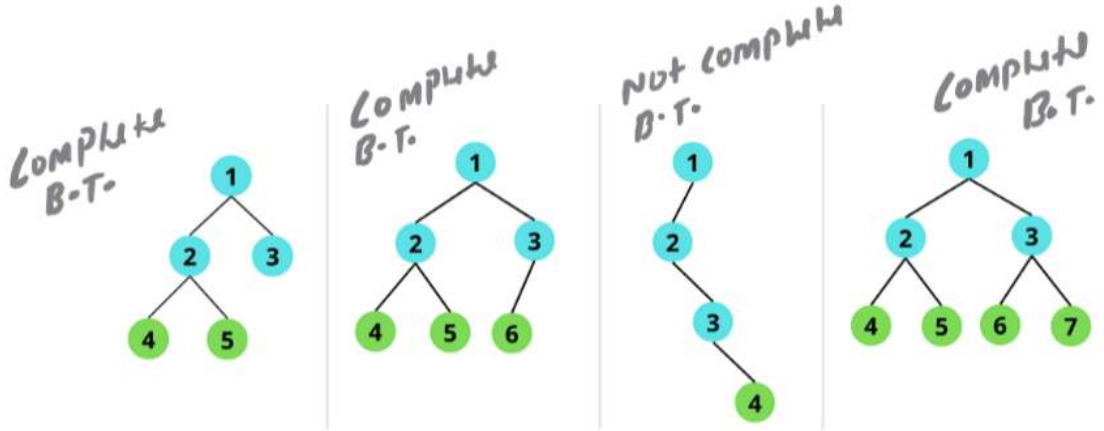
15/12/2023

HEAP CLASS - 1

📂 1. What is Heap Data Structure?

*A heap is a non-linear tree-based data structure where the tree is a **complete binary tree** and follows heap properties.*

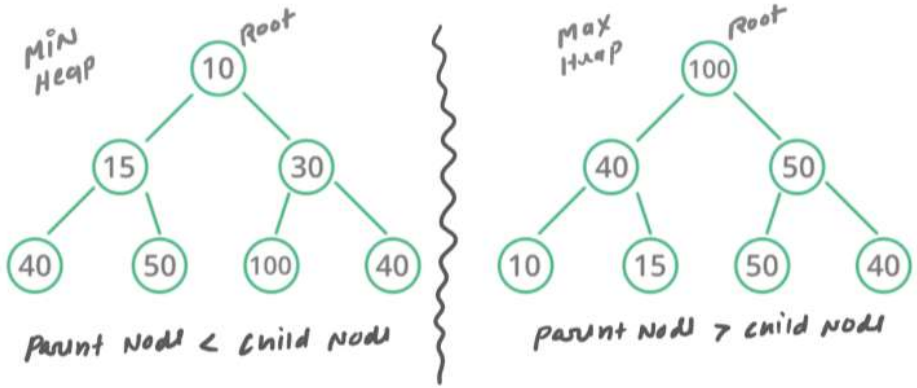


📂 2. Types of Heap Data Structure

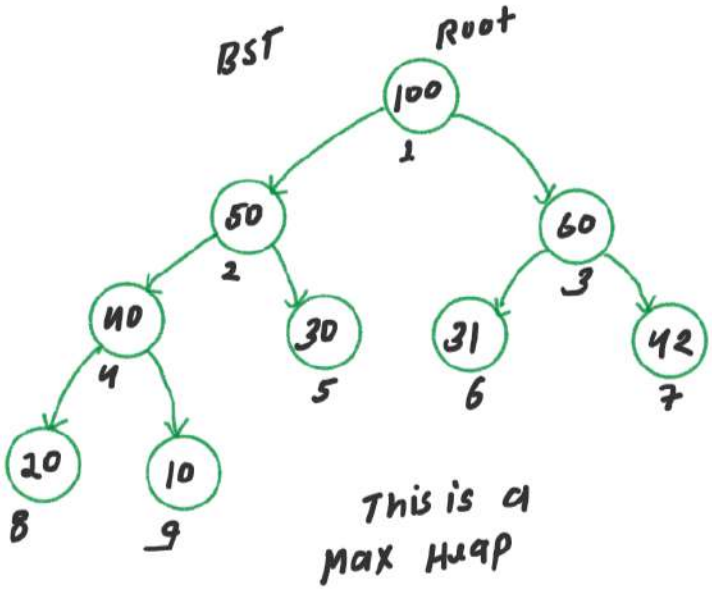
A heap can be either a min heap or a max heap.

In a max heap, *the key of a parent node is greater than or equal to the key of its child node.*

In a min heap, *the key of a parent node is less than or equal to the key of its child node.*



Heap Visualization through BST



Heap Implementation through Array

	-1	100	50	60	40	30	31	42	20	10
Array	0	1	2	3	4	5	6	7	8	9

When we are using 1 Based Indexing of Array

Child to Parent

index → index / 2

Child Node → goto → Parent Node

Parent to child

index → goto → Left Child = 2*index

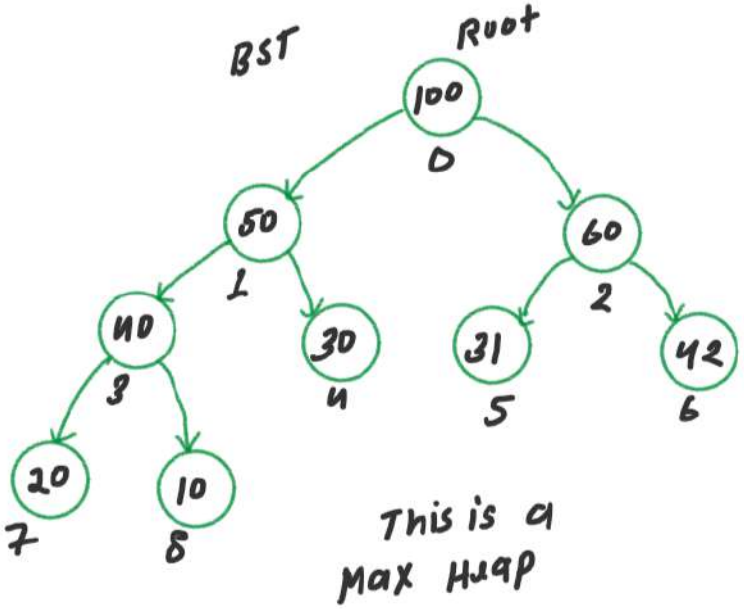
index → Parent Node goto → Right Child = 2*index + 1

Child Node Index = 2 → Parent Node Index = 2/2 = 1

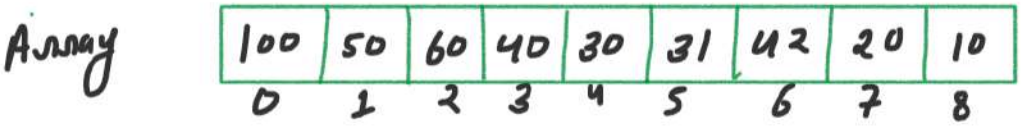
Parent Node Index = 4 → Left Child = 2*4 = 8

Right Child = 2*4 + 1 = 9

Heap Visualization through BST



Heap Implementation through Array



When we are using 1 Based Indexing of Array

Child to Parent

Child = i → Parent = $\lfloor i/2 \rfloor$

Parent to child

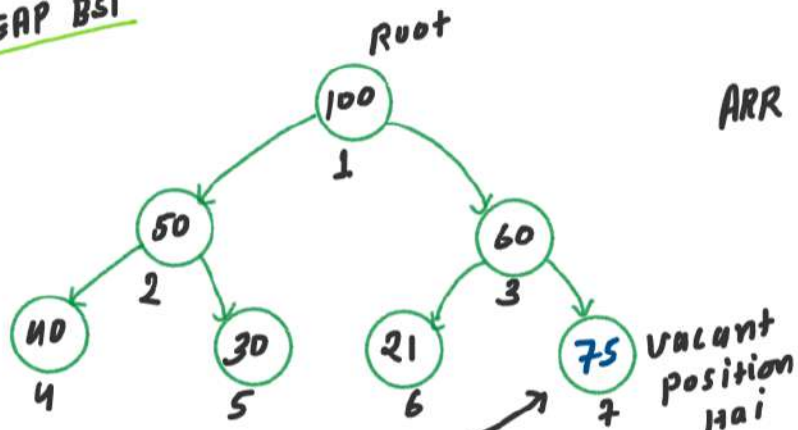
Parent = i → Left = 2*i + 1

Right = 2*i + 2

3. Insertion Operation of Heap Data Structure

{Always, insertion of new element from left to right}

MAX HEAP BST



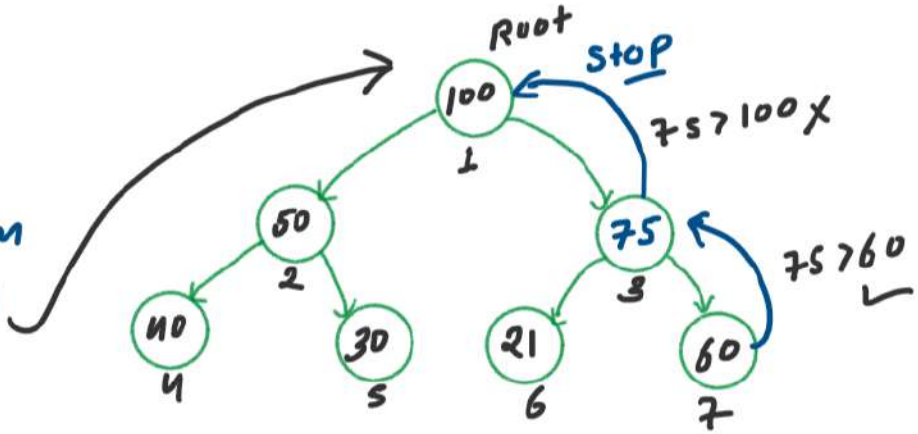
ARR

-1	100	50	60	40	30	21
0	1	2	3	4	5	6

We want to insert 75

1) insert element into vacant position

2) construct the position [Heapify]



What is Heapification ?

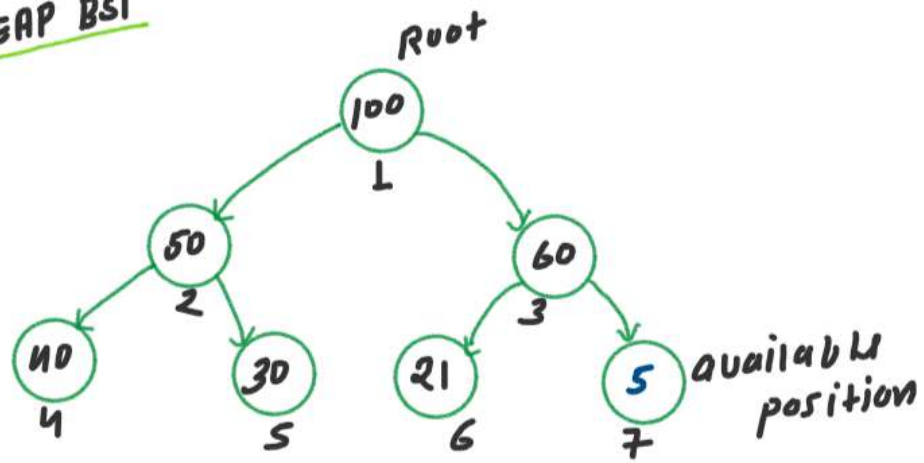
New Element > parent Node

↳ swap

New Element < parent Node

↳ ignore / stop

MAX HEAP BST

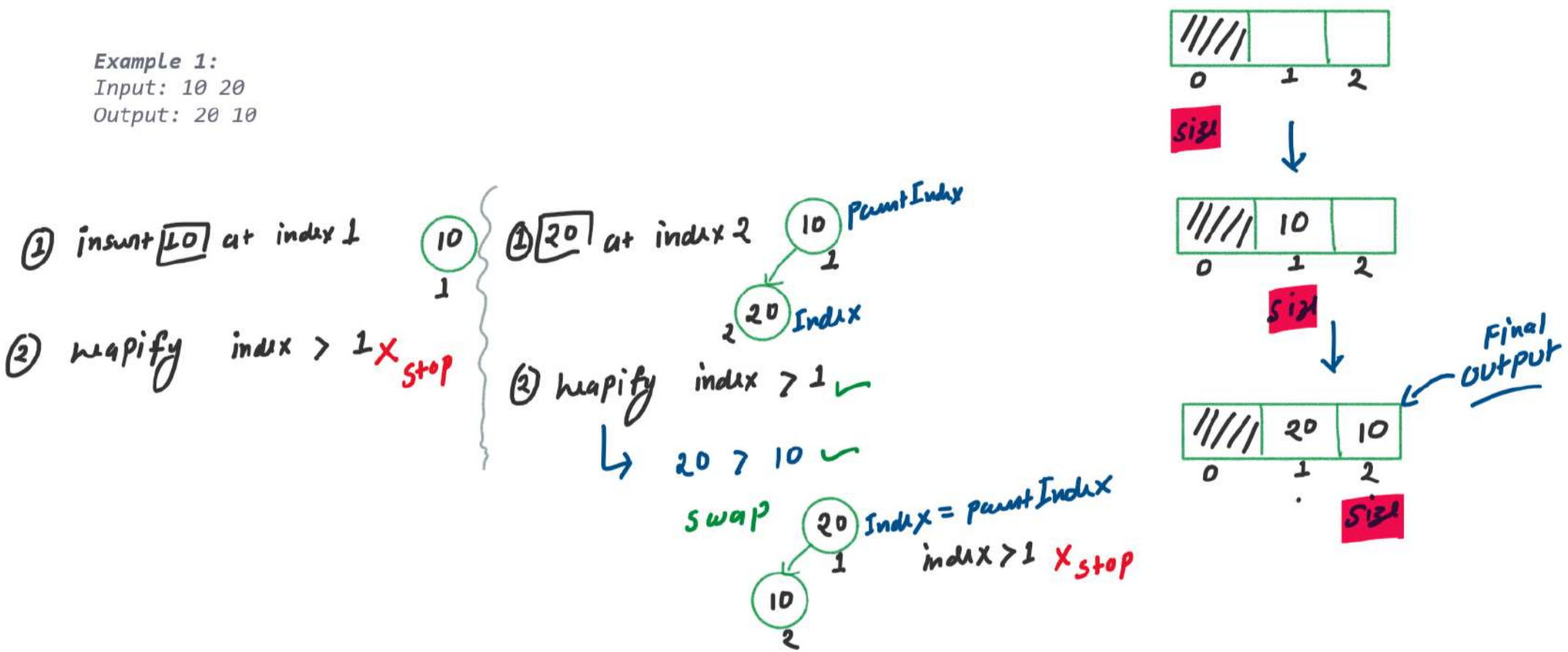


1. insert 5
2. 5 > 60 False

↳ Stop

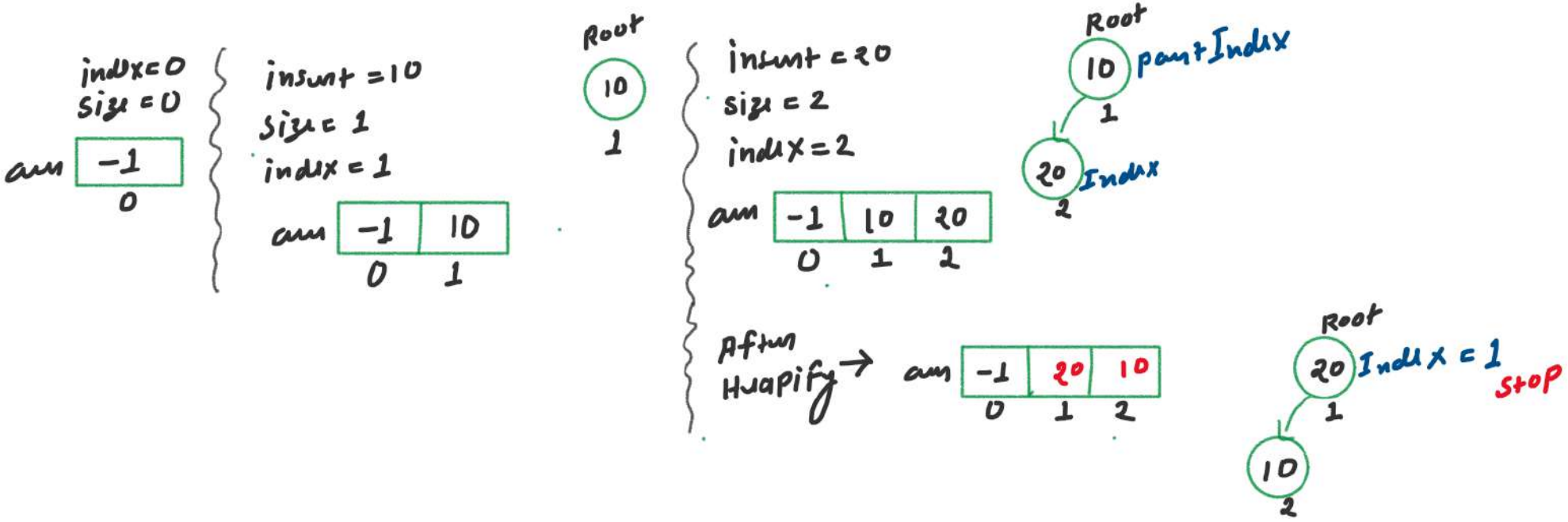
DRY RUN

```
Example 1:
Input: 10 20
Output: 20 10
```



DRY RUN

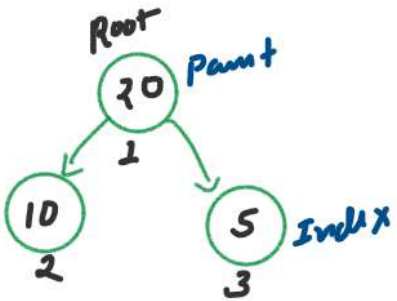
Example 2:
Input: 10 20 5 11 6
Output: 20 11 5 10 6



insert = 5
size = 3
index = 3

arr	-1	20	10	5
	0	1	2	3

Before and After Heapification

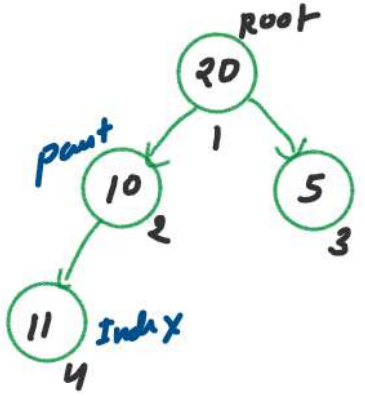


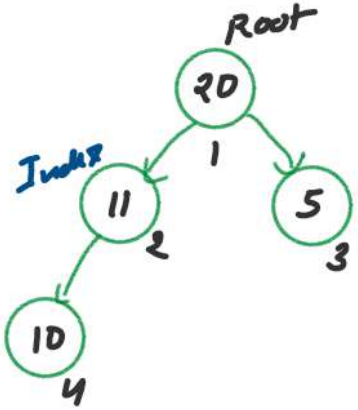
insert = 11
size = 4
index = 4

arr	-1	20	10	5	11
	0	1	2	3	4

After Heapify =>

arr	-1	20	11	5	10
	0	1	2	3	4





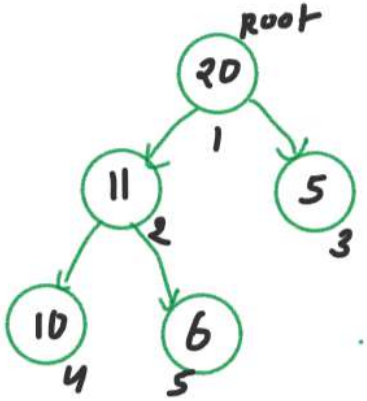
insert = 6

size = 5

index = 5

arr

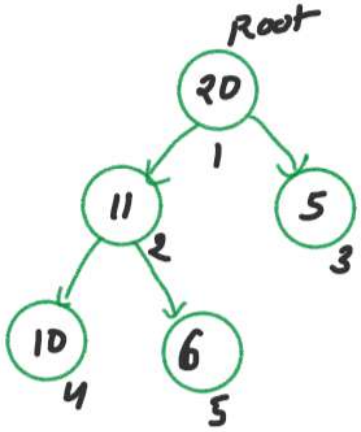
-1	20	11	5	10	6
0	1	2	3	4	5



After heapify =>

arr

-1	20	11	5	10	6
0	1	2	3	4	5



Output =

20	11	5	10	6

```cpp
#include<iostream>
using namespace std;

class Heap
{
    public:
        int *arr;
        int capacity;
        int size;

        Heap(int capacity){
            this->arr = new int[capacity];
            this->capacity = capacity;

            // Size = current number of elements in heap
            this->size = 0;
        }

        // 1. Insertion Operation of Heap Data Structure
        void insertion(int val){
            ...
        }

        // Printing Heap
        void printHeap(){
            ...
        }
};

int main(){
    int capacity = 5;
    Heap h(capacity);

    // Insertion of element
    h.insertion(10);
    h.insertion(20);
    h.insertion(5);
    h.insertion(11);
    h.insertion(6);

    cout<< "Printing Heap" << endl;
    h.printHeap();
}
```

```cpp
// 1. Insertion Operation of Heap Data Structure
void insertion(int val){
    if(size == capacity){
        cout<< "Heap Overflow" << endl;
        return;
    }

    // Size increase ho jayega (1-Based Indexing)
    size++;
    int index = size;
    arr[index] = val;

    // Take the value to its correct position
    while (index > 1)
    {
        int parentIndex = index/2;
        if(arr[index] > arr[parentIndex]){
            // Swap kardo
            swap(arr[index], arr[parentIndex]);
            index = parentIndex;
        }
        else{
            // Loop ko stop kardo
            break;
        }
    }
}

// Printing Heap
void printHeap(){
    for (int i = 1; i <= size; i++)
    {
        cout<< arr[i] << " ";
    }
    cout<<endl;
}
```

Insertion

Heapification

Insertion Operation:

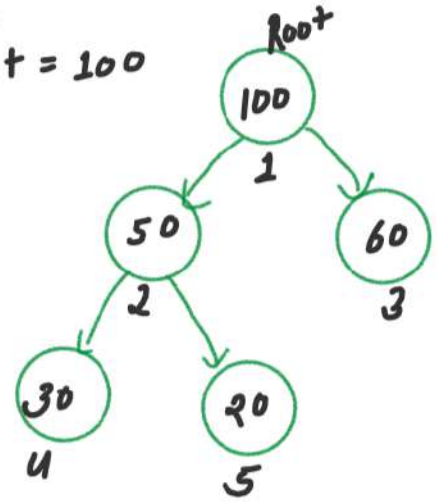
Time Complexity: *O(log N), Where N is number of elements in heap.*

Space Complexity: *O(capacity), Where capacity is constant.*

4. Deletion Operation of Heap Data Structure

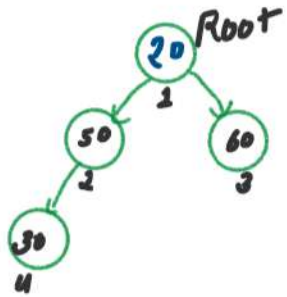
EX

Root Element = 100



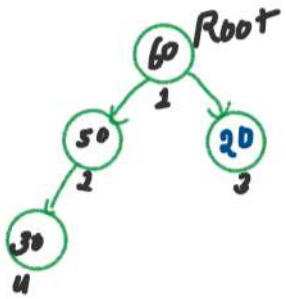
Always Remember, **Access Element** from Root & **Delete Element** from Root

STEP1 Root ko Replace kardo last element se.



STEP2 Ab Root ko uski correct position par lagadado

Output

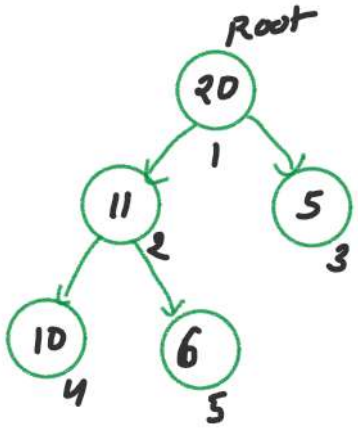


DRY RUN

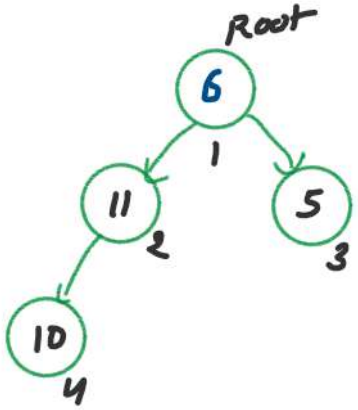
EX=2

Arr	-1	20	11	5	10	6
	0	1	2	3	4	5

size = 5



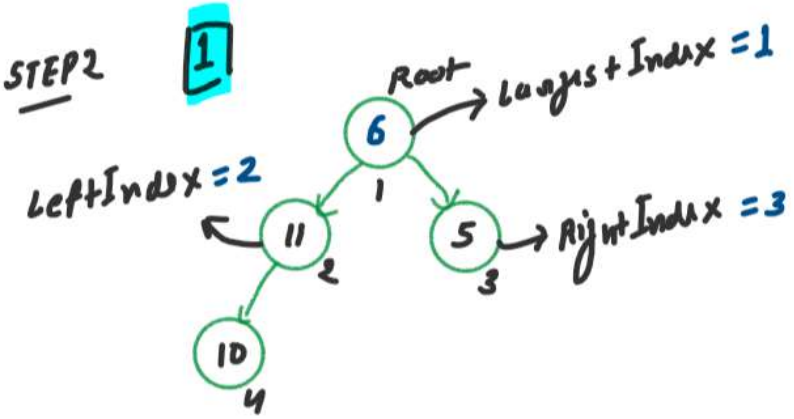
STEP 1



Arr	-1	6	11	5	10
	0	1	2	3	4

size = 4

STEP 2 [1]



Arr	-1	6	11	5	10
	0	1	2	3	4

index = 1

size = 4

Check left

6 < 11 TRUE

↳ largestIndex = 2

Check Right

11 < 5 FALSE

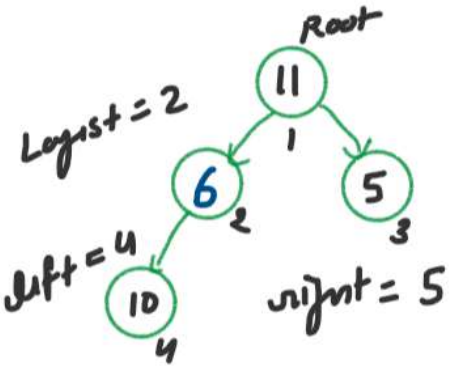
↳ largestIndex = 2

(Index != largestIndex)

↳ Swap

↳ Index = largestIndex

= 2

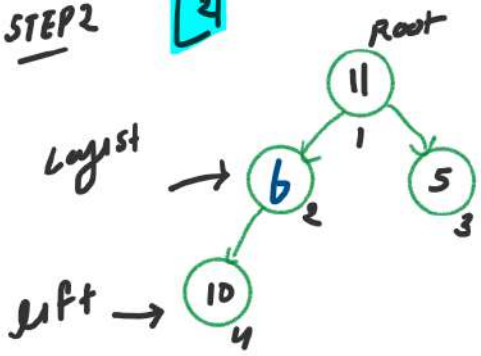


Arr	-1	11	6	5	10
	0	1	2	3	4

index = 2

size = 4

STEP 2 [2]



Arr	-1	11	6	5	10
	0	1	2	3	4

index = 2
size = 4

- LargestIndex = 2
- leftIndex = 4
- RightIndex = 5

Check left

6 < 10 TRUE

↳ largestIndex = 4

Check Right

↳ Out of Bound Condition

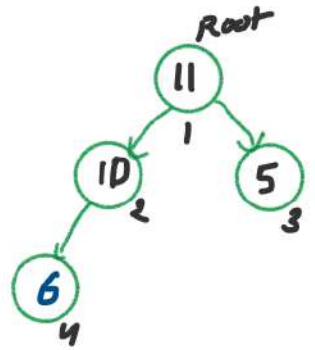
Right < size

5 < 4 False

(Index $_{0}^{!}$ = largestIndex)

↳ Swap

↳ Index = largestIndex = 4



Arr	-1	11	10	5	6
	0	1	2	3	4

index = 4
size = 4

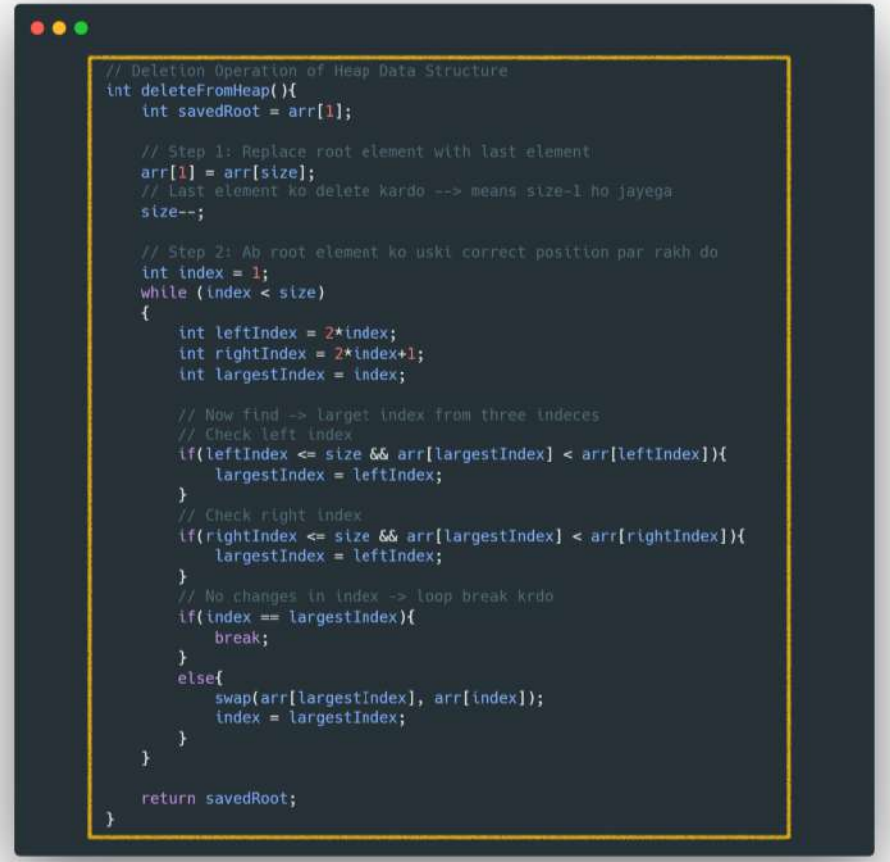
TRUE

4 == 4

STOP ⇒ Index == size

Output

11 10 5 6



```cpp
// Deletion Operation of Heap Data Structure
int deleteFromHeap(){
    int savedRoot = arr[1];

    // Step 1: Replace root element with last element
    arr[1] = arr[size];
    // Last element ko delete kardo --> means size-1 ho jayega
    size--;

    // Step 2: Ab root element ko uski correct position par rakh do
    int index = 1;
    while (index < size)
    {
        int leftIndex = 2*index;
        int rightIndex = 2*index+1;
        int largestIndex = index;

        // Now find -> larget index from three indeces
        // Check left index
        if(leftIndex <= size && arr[largestIndex] < arr[leftIndex]){
            largestIndex = leftIndex;
        }
        // Check right index
        if(rightIndex <= size && arr[largestIndex] < arr[rightIndex]){
            largestIndex = leftIndex;
        }
        // No changes in index -> loop break krdo
        if(index == largestIndex){
            break;
        }
        else{
            swap(arr[largestIndex], arr[index]);
            index = largestIndex;
        }
    }

    return savedRoot;
}
```

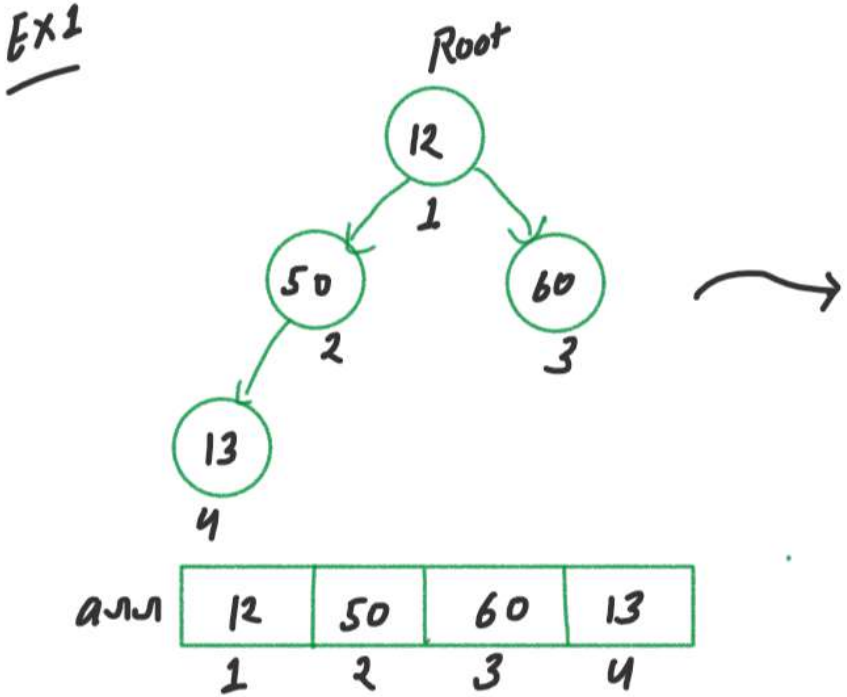
Deletion Operation:

Time Complexity: *O(log N), Where N is number of elements in heap.*

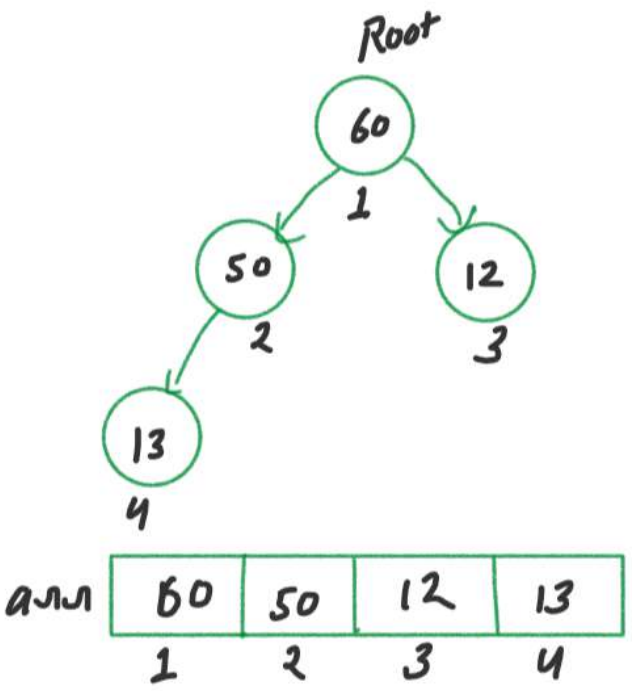
Space Complexity: *O(1), Where 1 is constant.*

5. HEAPIFY Using Recursion

EX1

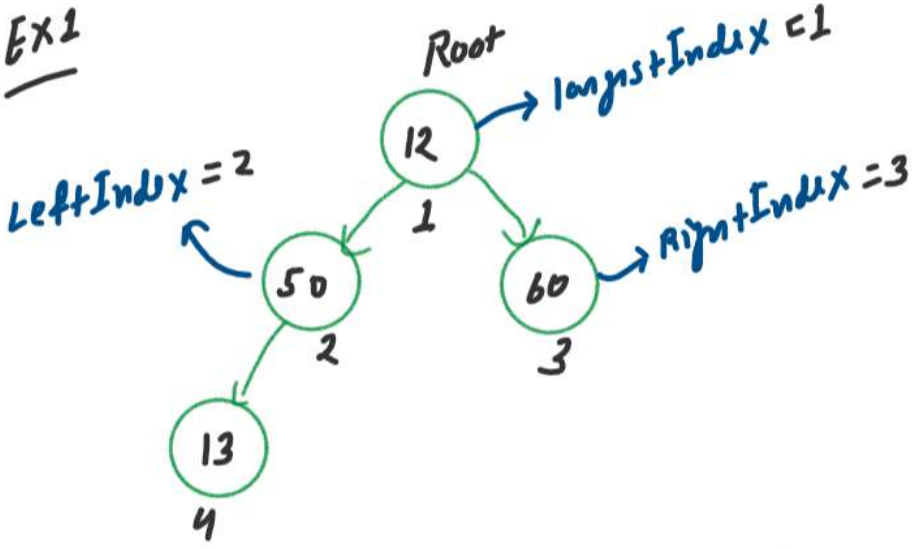


Output



DRY RUN

EX1



Index = 1

Size = 4

Find largest among these

Check left

12 < 50

↳ largestIndex = 2

Check Right

50 < 60

↳ largestIndex = 3

if(index != largestIndex)

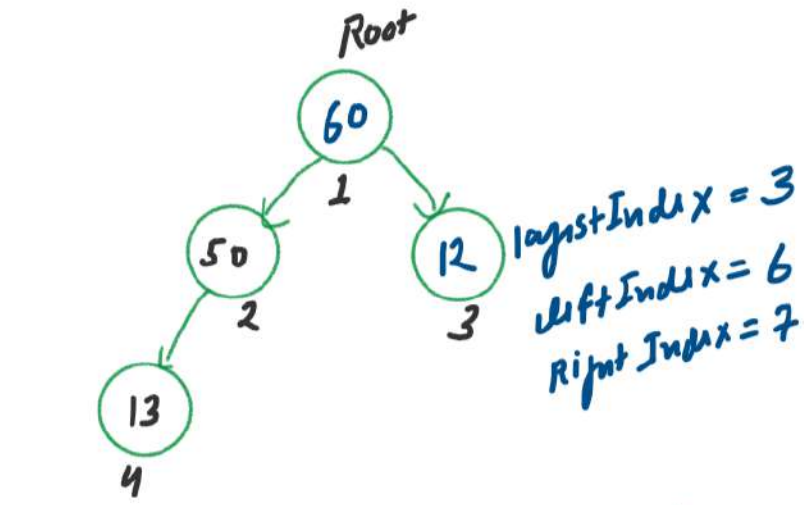
1-case hum solve kar lenge

swap(arr[index], arr[largestIndex]);

Ab Recursion solve kar lega

index = largestIndex;

f(arr, size, index);



Final Output

Right and left Index TO Available Iti Nani Hai

Means

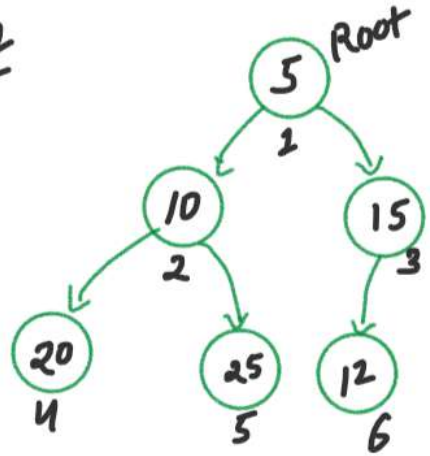
Out of Bound conditions Apply Ho jayjiji

left <= size

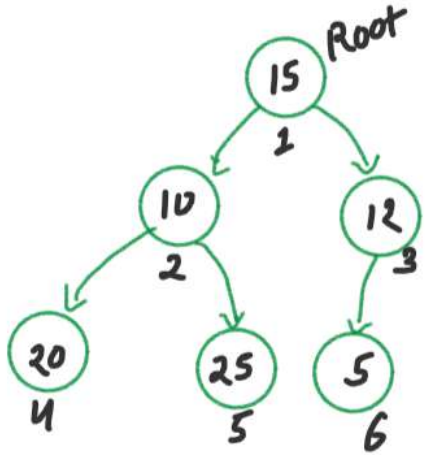
Right <= size

(TO Alag se Base conditions Apply Karne Ki Need Nahi Hai)

EX2



Output



arr

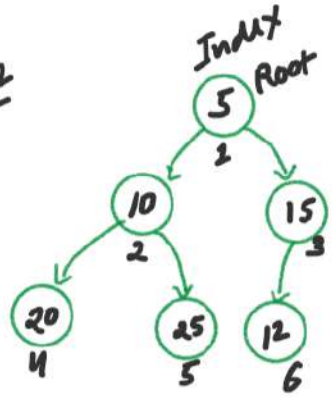
1	2	3	4	5	6
5	10	15	20	25	12

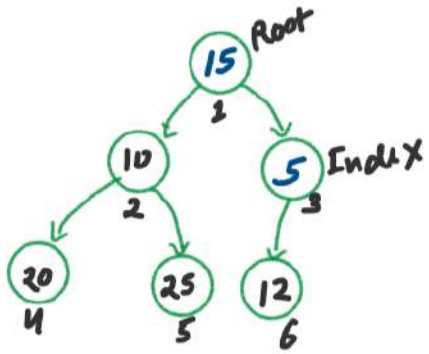
arr

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

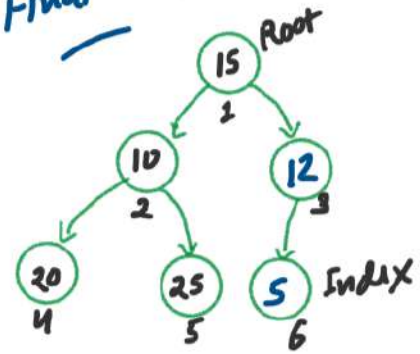
DRY RUN

EX2





Final output



arr

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

f(arr, 6, 1)

arr

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

f(arr, 6, 3)

arr

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

f(arr, 6, 6)

STOP (index = size)

```cpp
// 5. Heapify using recursion
void heapify(int *arr, int size, int index){
    int leftIndex = 2*index;
    int rightIndex = 2*index+1;
    int largestIndex = index;

    // Now find -> larget index from three indeces
    if(leftIndex <= size && arr[largestIndex] < arr[leftIndex]){
        largestIndex = leftIndex;
    }
    if (rightIndex <= size && arr[largestIndex] < arr[rightIndex]){
        largestIndex = rightIndex;
    }
    if(index != largestIndex){
        // 1 case hum solve kar lenge
        swap(arr[largestIndex], arr[index]);

        // Ab recursion solve kar lega
        index = largestIndex;
        heapify(arr, size, index);
    }
}

/*
Example 1:
Input:
arr[12 50 60 13] and size = 4

Output:
Printing Heap
60 50 12 13

Example 2:
Input:
arr[5 10 15 20 25 12] and size = 6

Output:
Printing Heap
15 10 12 20 25 5
*/
```

Time Complexity = $O(\log N)$

Space Complexity = $O(\log N)$, where $\log N = H$

↳ Due to Max function calls and equals to Height of B.T.

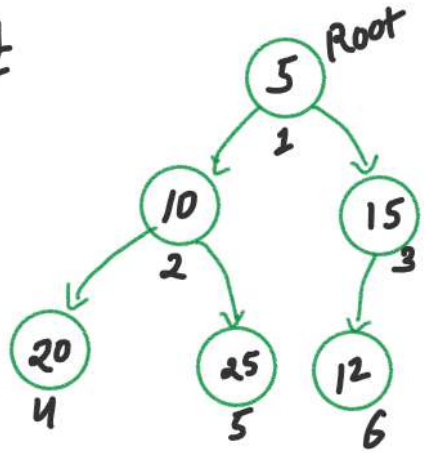
↳ Where N = Numbers of Element in Array

H = Height of Binary Tree

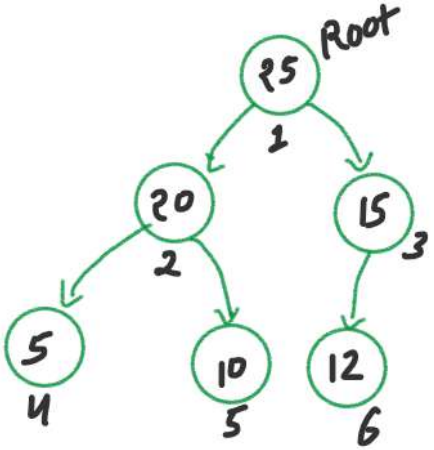
6. Convert Array to Heap (Create heap using array)

Time complexity = O(N)

Ex1



Output



arr

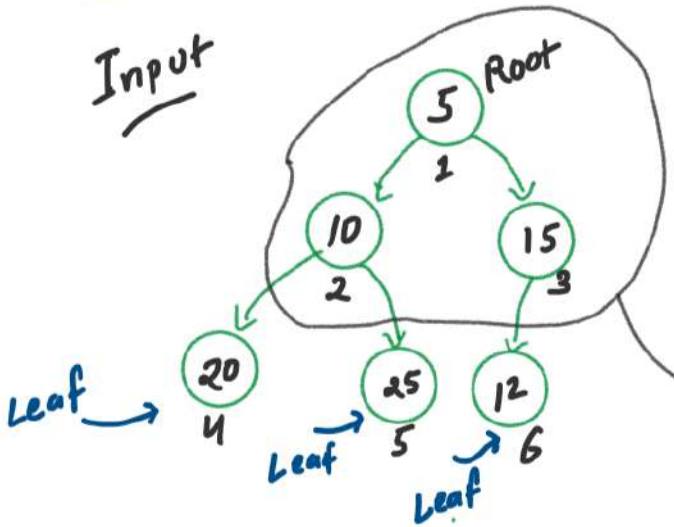
5	10	15	20	25	12
1	2	3	4	5	6

arr

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

Logic Building

Input



arr

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

N = 6

Leaf Nodes are valid Always

$\left(\frac{N}{2}+1\right)^{th}$ to N^{th} Node → (4 to 6)

Loop ($\frac{N}{2}$ to index)

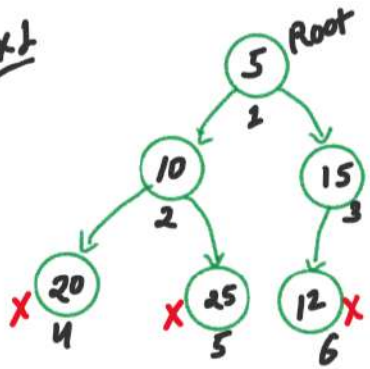
```
for( i = N/2 ; i > 0 ; i--) {
        Heapify( arr, N , i) ;
}
```

Non leaf Node (3 to 1)

DRY RUN

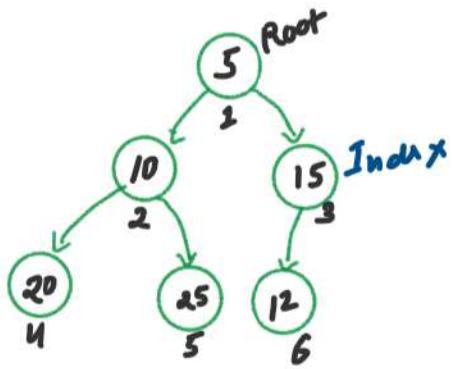
```
buildHeap(arr, N){
    for(3 to 1){
        Heapify(arr, N, index);
    }
}
```

Ex1



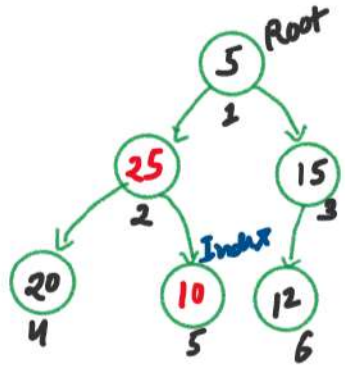
arr	5	10	15	20	25	12
	1	2	3	4	5	6

Heapify(arr, 6, 3)



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

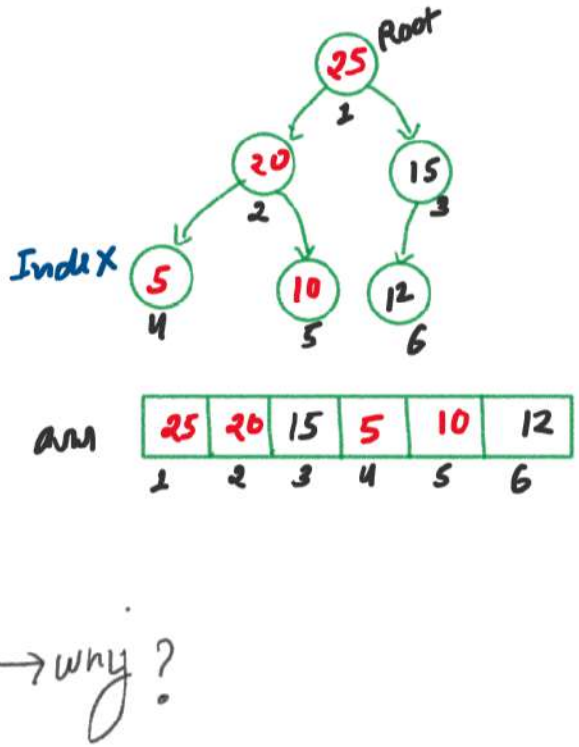
Heapify(arr, 6, 2)



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

Heapify(arr, 6, 1)

Final output



T.C. => O(N)

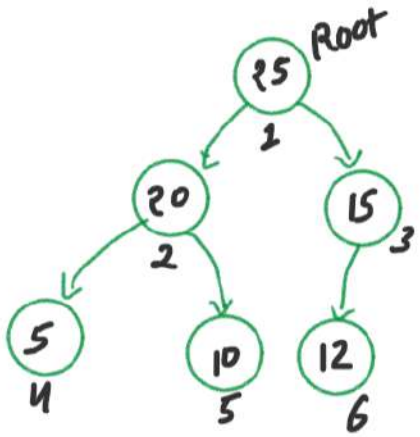
Where, N is number of elements in Array.

→ why ?

```c
// 6. Create heap using array
void buildHeap(int *arr, int n){
    for(int index = n/2; index > 0 ; index--){
        heapify(arr, n, index);
    }
}
```

7. Heap Sort Time complexity = O(N log N)

Ex 1



Input

Arr

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

Output

Arr

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

Approach

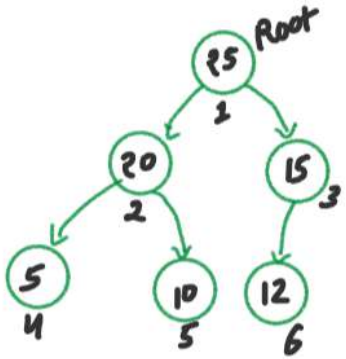
STEP1 Last Node ko First Node ko swap karke Rahe jab tak index $!_0 = 1$

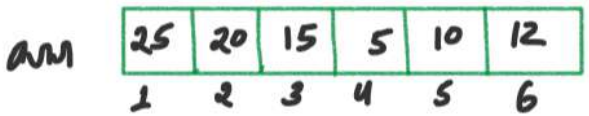
STEP2 First Node ko Heapify karo

↳ (Root Index = 1)

DRY RUN

1



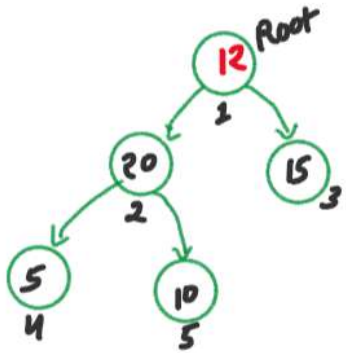


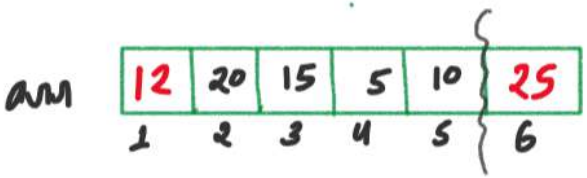
STEP1 Swap

Root Node = 25

Last Node = 12

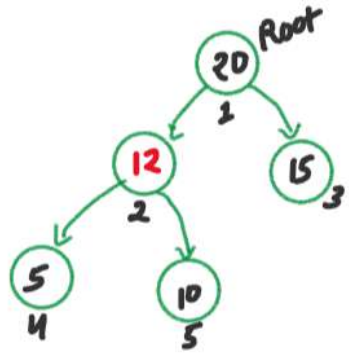
N = 6

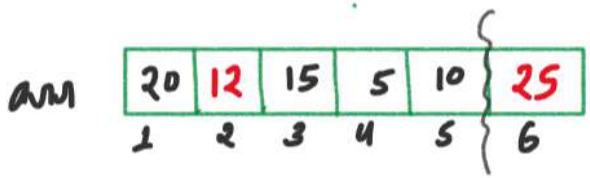




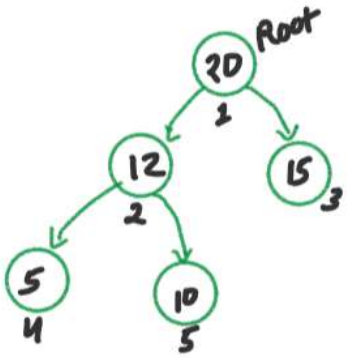
STEP2 Heapify 12

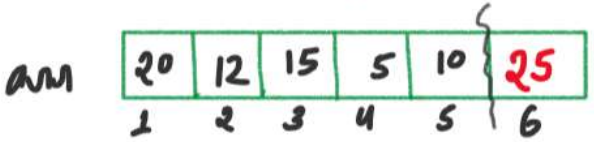
N = 5





2



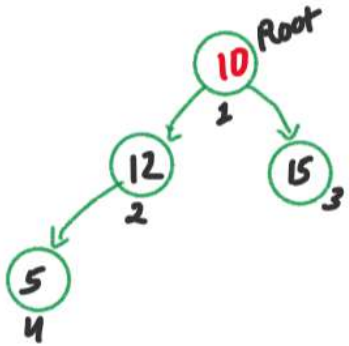


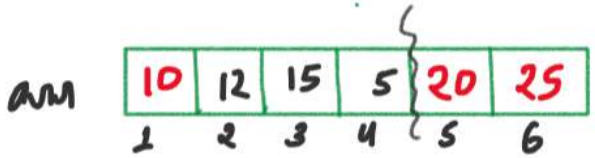
STEP1 Swap

Root Node = 20

Last Node = 10

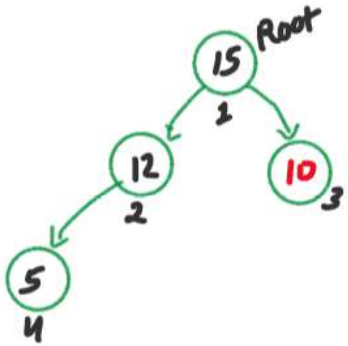
N=5

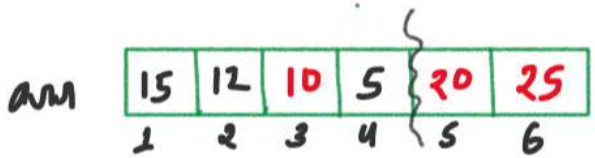




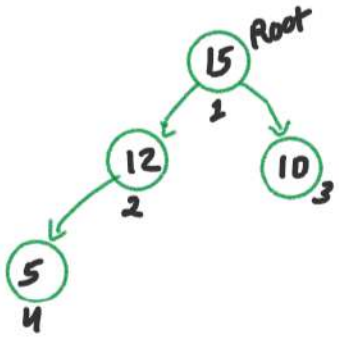
STEP2 Heapify 10

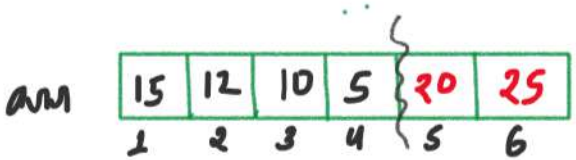
N=4





3



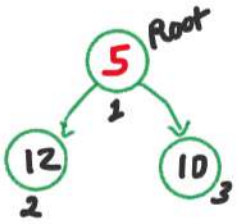


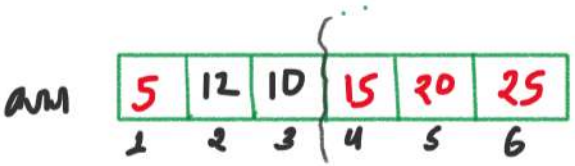
STEP1 Swap

Root Node = 15

Last Node = 5

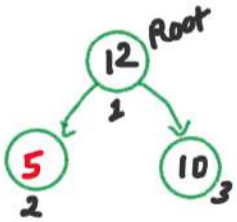
N = 4

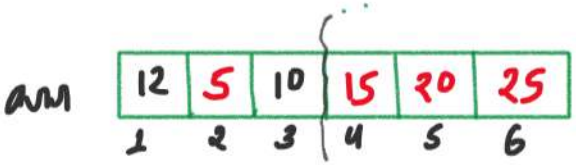




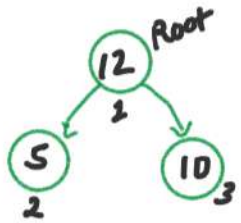
STEP2 Heapify 5

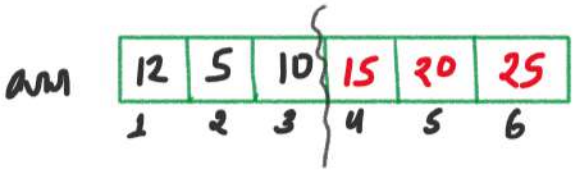
N = 3





4



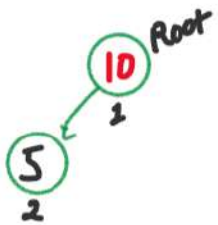


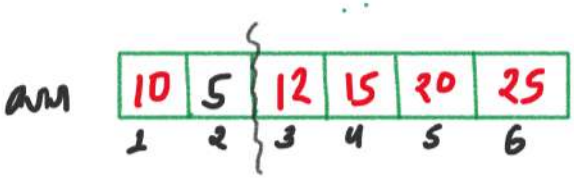
STEP1 Swap

Root Node = 12

Last Node = 10

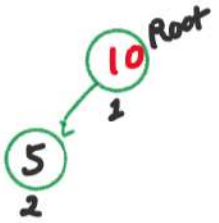
N=3

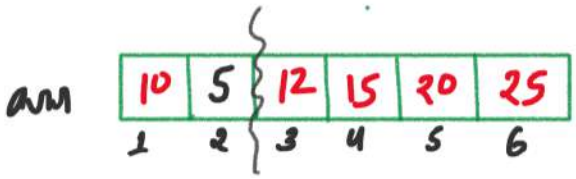




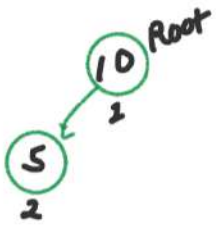
STEP2 Heapify 10

N=2





5



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

STEP1 Swap

Root Node = 10

Last Node = 5

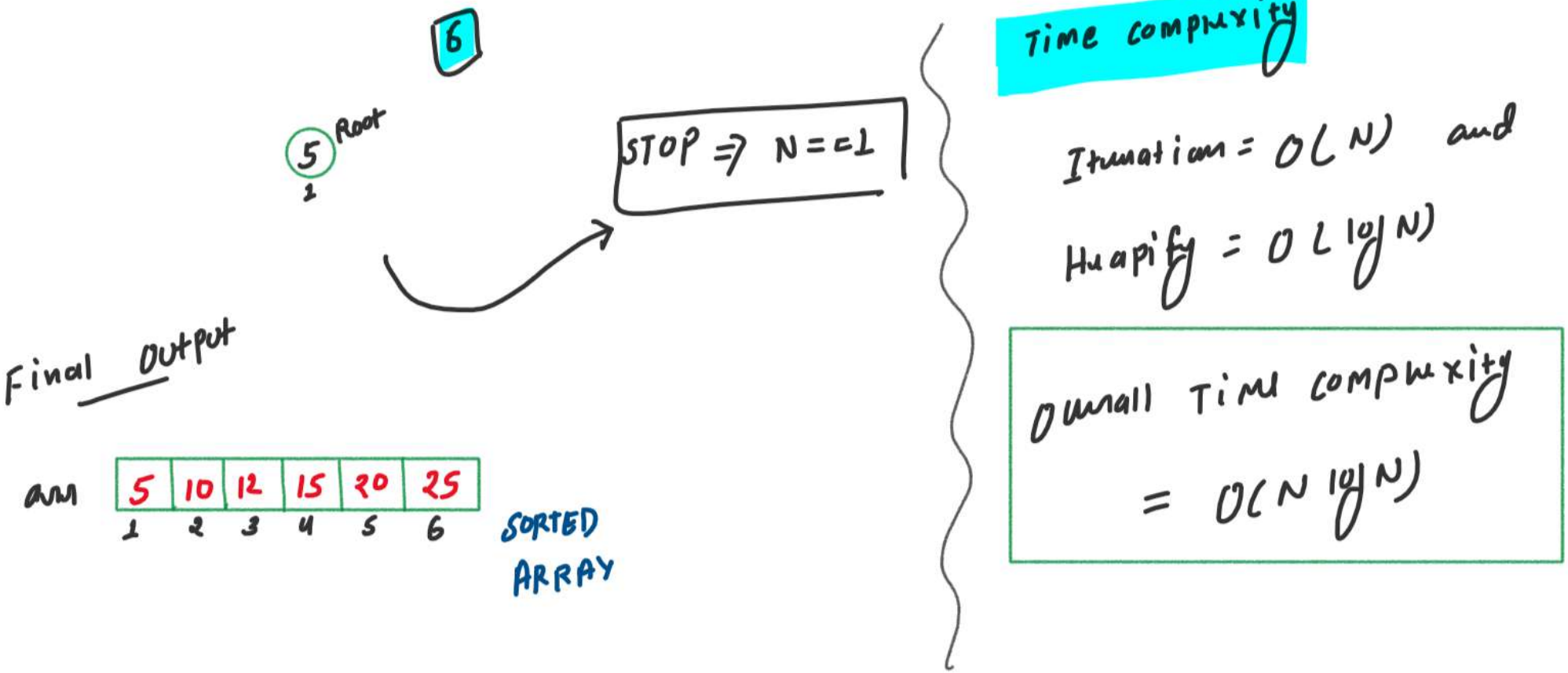
N = 2

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

STEP2 Heapify 5

N = 1

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


6
5
Root
1
STOP => N==1
Final Output
arr
5 10 12 15 20 25
1 2 3 4 5 6
SORTED ARRAY
Time complexity
Iteration = O(N) and
Heapify = O(log N)
Overall Time complexity
= O(N log N)

```c
// 7. Heap Sort
void heapSort(int *arr, int n){
    while (n != 1)
    {
        swap(arr[1], arr[n]);
        n--;
        heapify(arr, n, 1);
    }
```

Time Complexity: *O(N log N),*
Where N is number of elements in array

Space Complexity: *?*