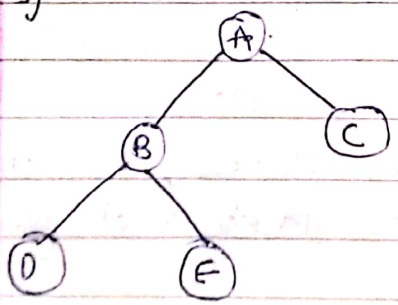
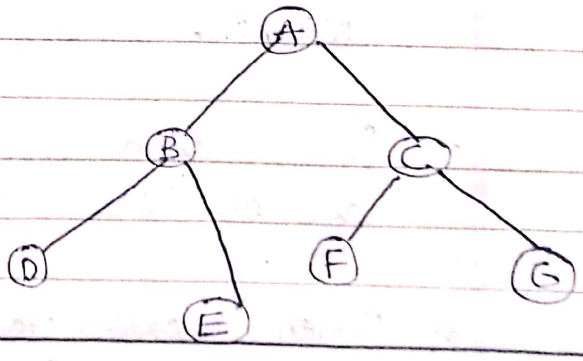
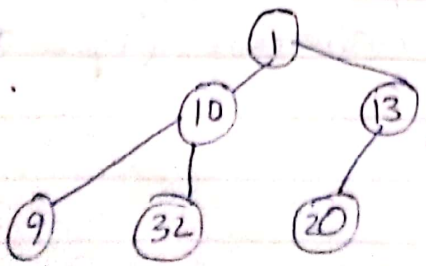
Name : Sarthak Jain.
Roll no: R177219162.

Ans. 1.

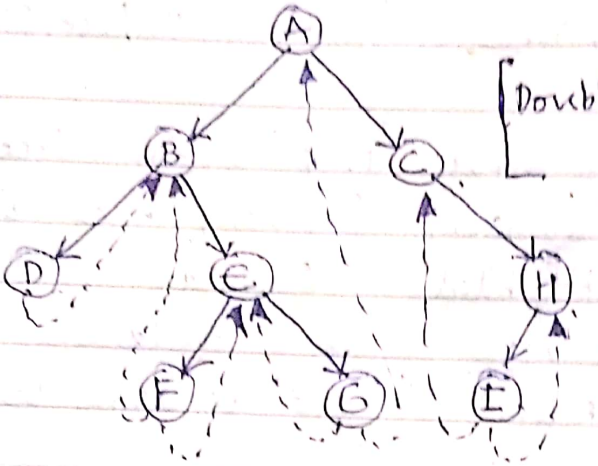
Complete Binary Tree	Perfect Binary Tree
1) A binary tree is a complete binary tree if all levels are completely filled except possibly the last level and last level has all keys as left as possible.	1) A binary tree is Perfect binary tree if all the internal nodes have two children and all leaves are at same level.
2) Every complete binary tree is not complete binary tree.	2) Every perfect binary tree is a complete binary tree.
3) Example. 	3) Example. 

Ans. 2. 1)



Sarthak Jain - R177219162.

Ans.2. 2). Example of a threaded binary tree:



[Double threaded binary tree]

Advantages over regular Binary Trees

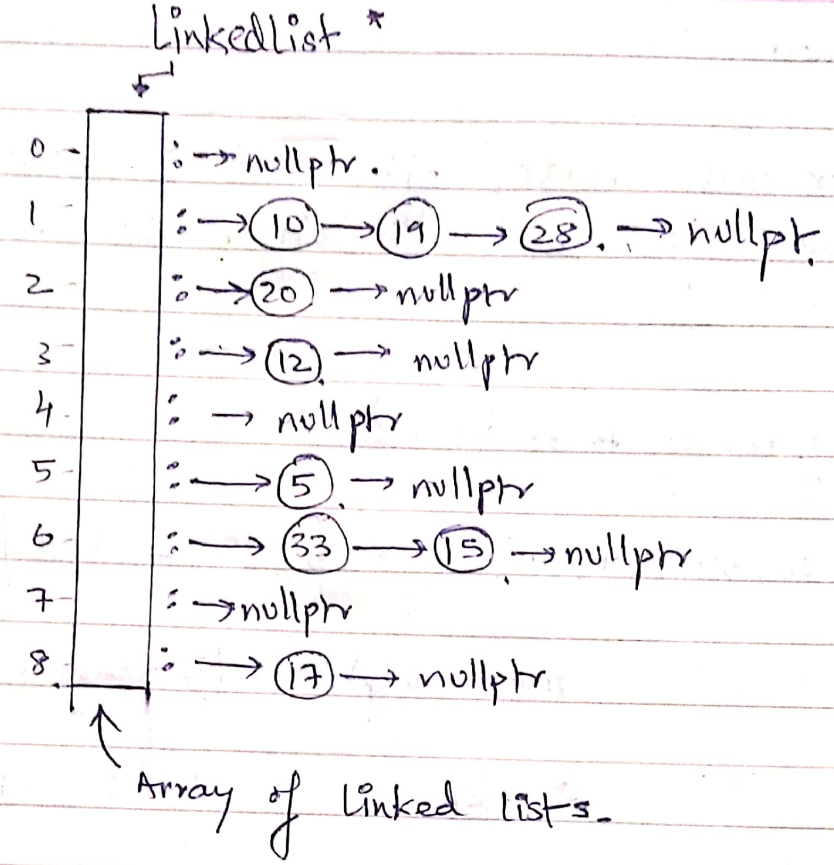
1) Each leaf node points to inorder successor of the leaf node. (And if doubly threaded binary tree is used, leaf nodes also point to inorder predecessor). This saves traversal of the tree without recursion.

2) Saves extra information per node, so traversal can be done without stacks and recursion.

~~3) Easier inorder traversal.~~

3) Nodes are circularly linked, so we can move in either direction (up & down).

Ans. 3. 1.)



Hash function : $[h(k) = k\%9]$.

2.)

Maximum Chain Length : 3.
Minimum Chain Length : 0
Average Chain Length : 1 (Including Null chains)
1.5 (Excluding empty chains)

Ans. 4.

1) AVL Trees are always height balanced Binary search Trees.
2) AVL Trees have specially designed insertion and deletion functions that maintain the height difference between -1 and 1.
3) In AVL, trees, insertions are done normally, after which a check is done to see if the operation created imbalance in the tree.

Ans. 3 4 Continued.....

4) For each node, left child is smaller than node and right child is bigger than node.

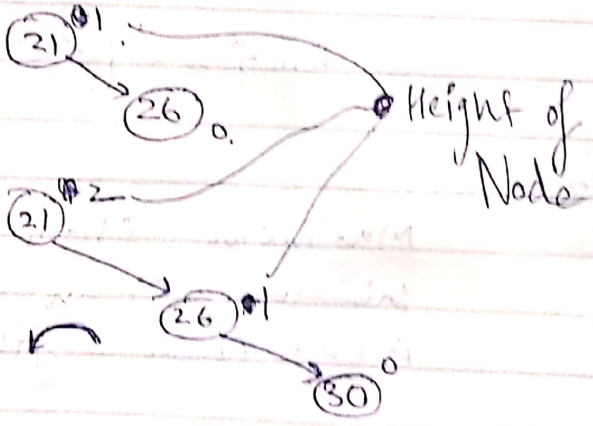
Ans. 4. 2).

Array of numbers to be inserted

: [21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7].

1. root → nullptr (Initialisation).

2. (21) (21 is root).

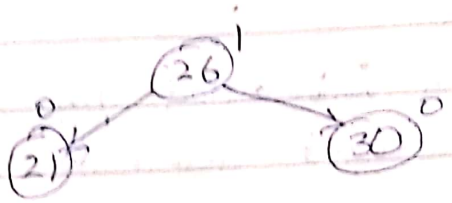
3. 

4.

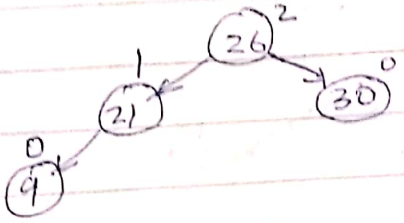
Now, since height difference at '21' is out of range of -1 to 1, we have to perform the rotation to balance the tree.

Since, new node is bigger than root and root→right, we will perform left rotation.

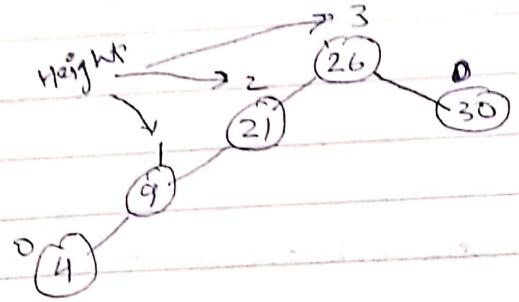
Ans. 4. 4.7) continued.



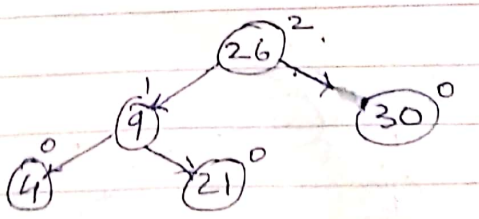
After the rotation, the tree is balanced now.

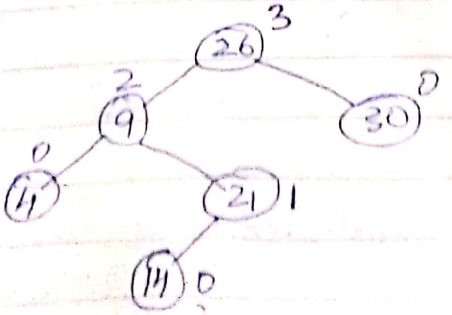
5. 

* balanced.

6. 

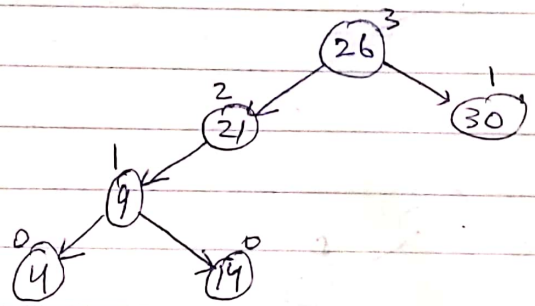
Now again, the tree is unbalanced, this time, at 21, balance is 2. Since new node is less than root (21) and root → left (9), we apply a right rotation at 21.



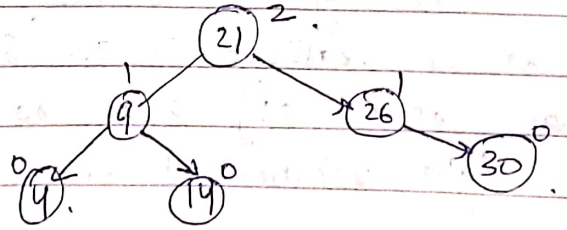
7. 

Ans. 4. Continued....

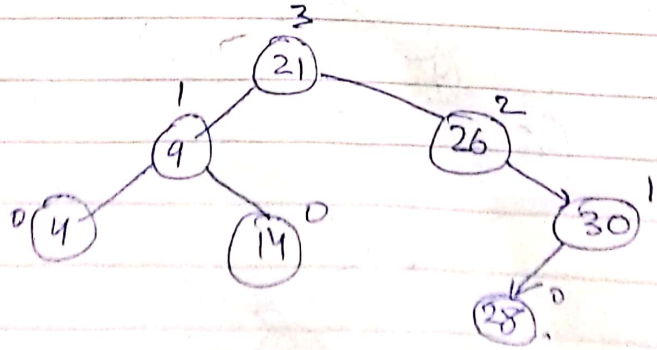
Again, tree is imbalanced from ~~9~~ 26. but this time, element new_node is not less than root → left, so we have to first apply left rotation at root → left, then apply Right rotation to root(26).



26, is still unbalanced, so we apply right rotations to 26.



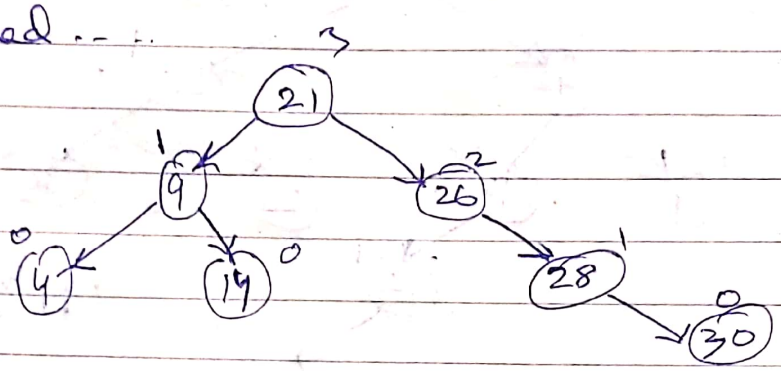
8.



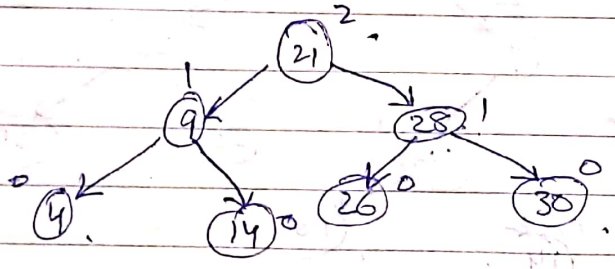
26, is unbalanced, so we apply right rotation at 30.

Ans 4. continued ----.

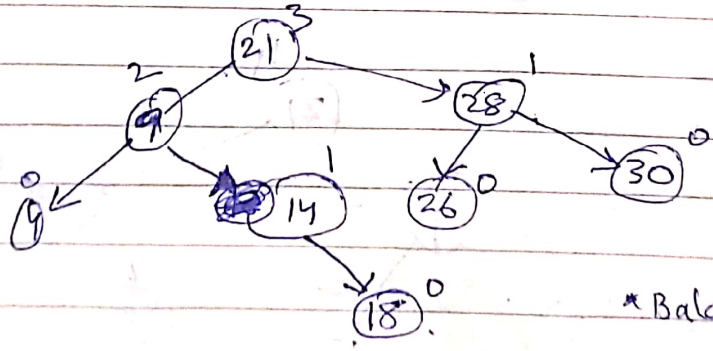
8) continued ....



Now, 26 is unbalanced so, we apply left rotation at 26.

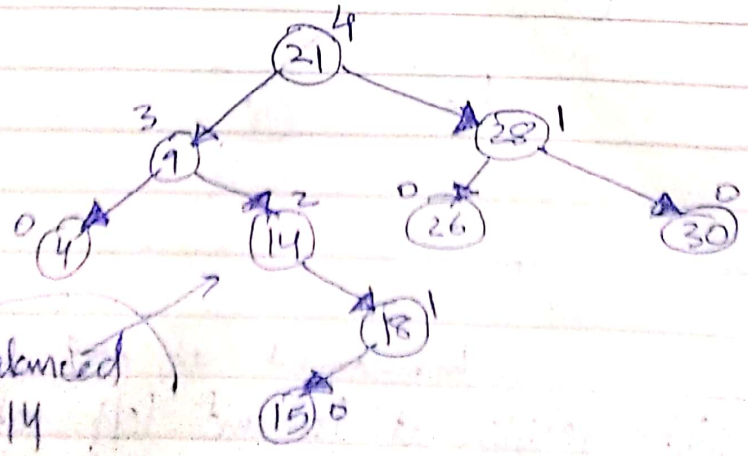


9.



*Balanced tree.

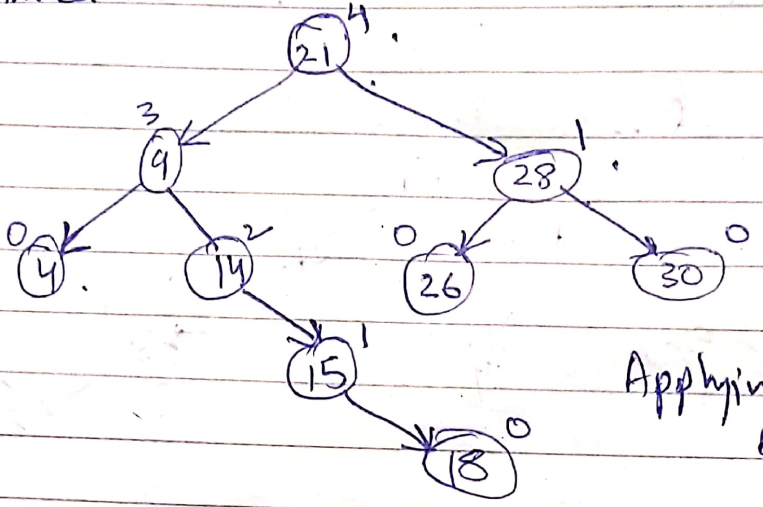
10.



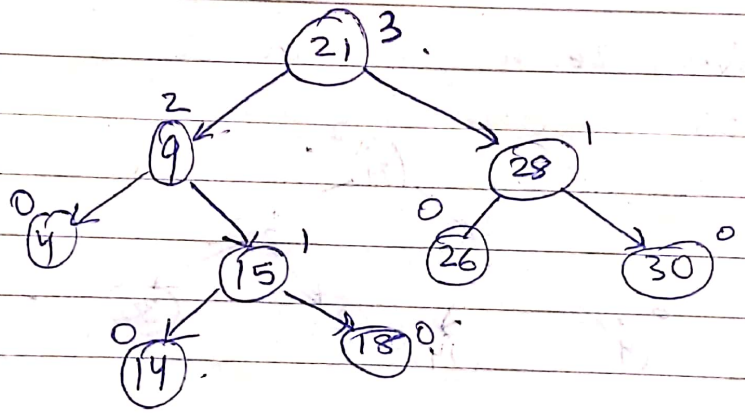
Imbalanced at 14

Rotate 18 to Right.

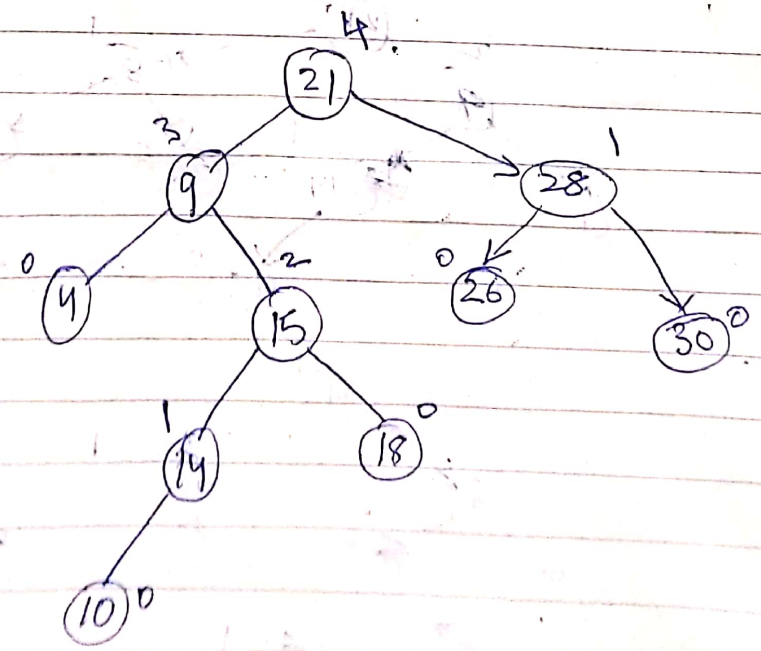
Ans. 4. 10. continued....



Applying left rotation at 14.

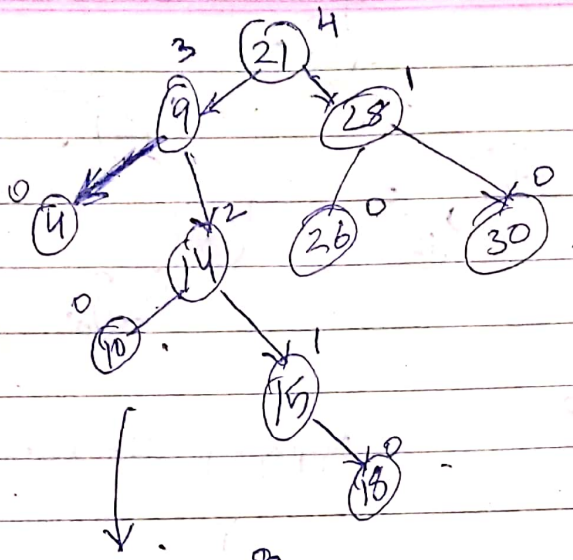


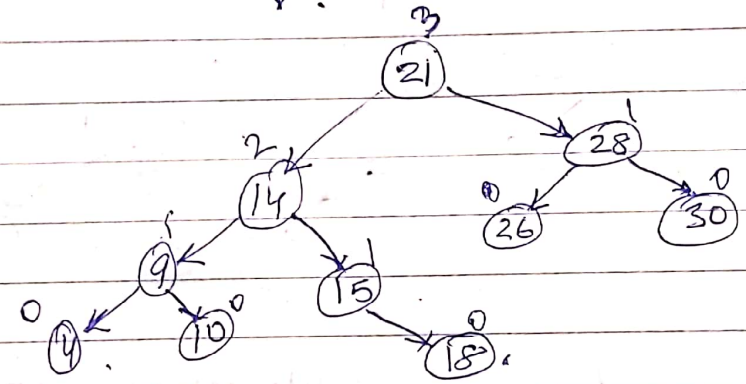
11.



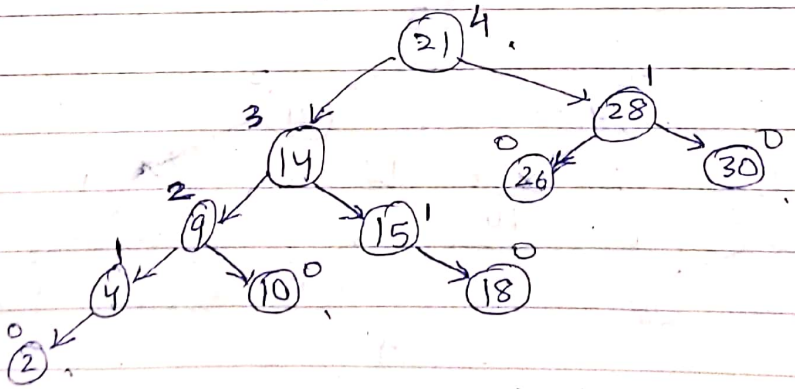
Imbalanced at 9, So we apply Right rotation at 15 and left rotation at 9.

Ans 4. 11. continued...

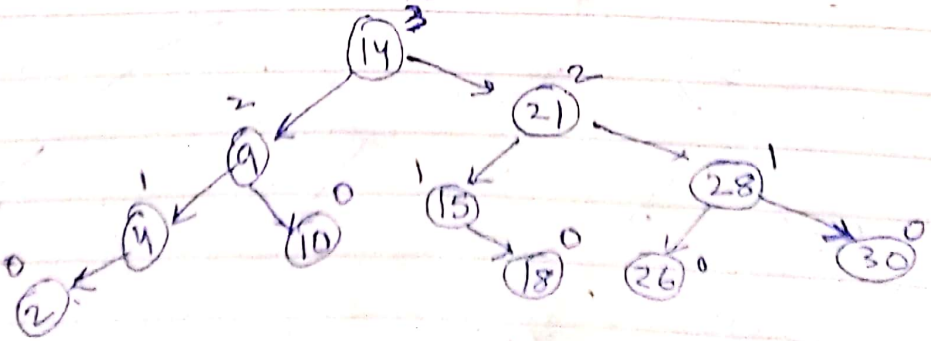




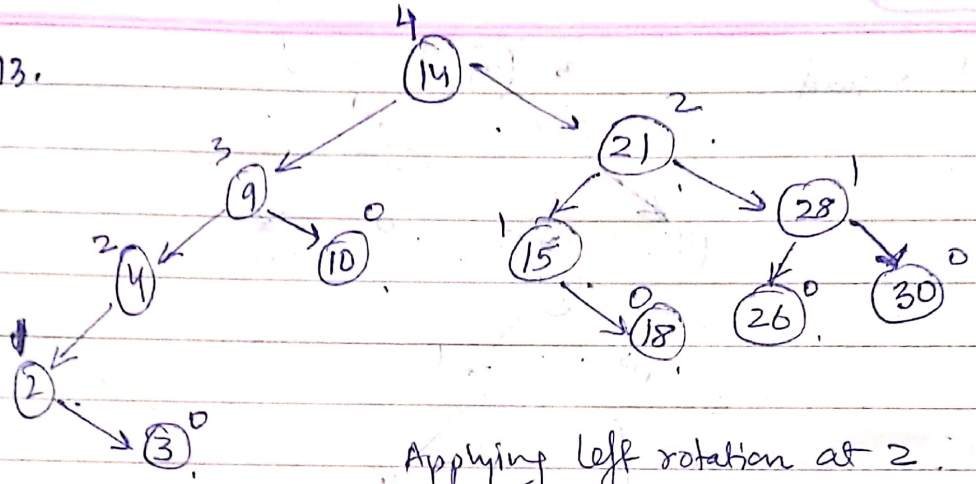
12.



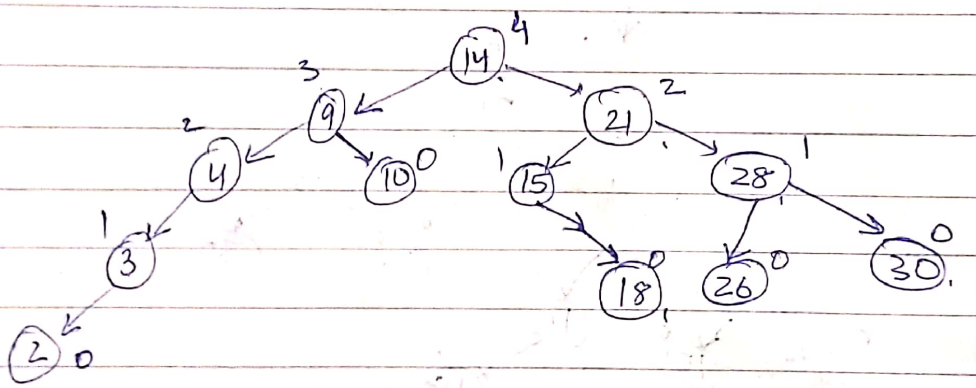
Applying right rotation at 21.



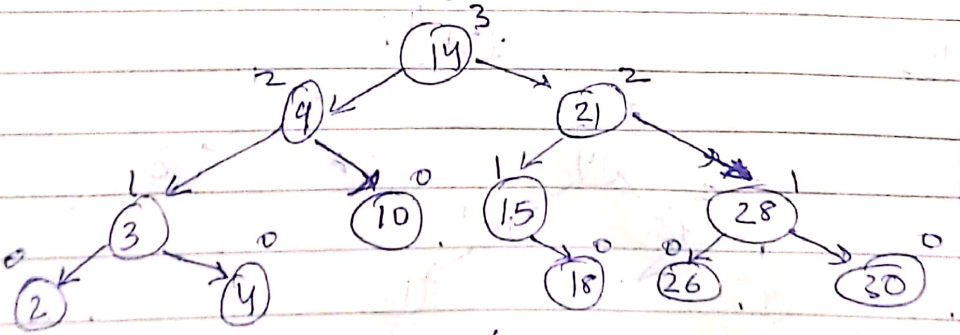
13.



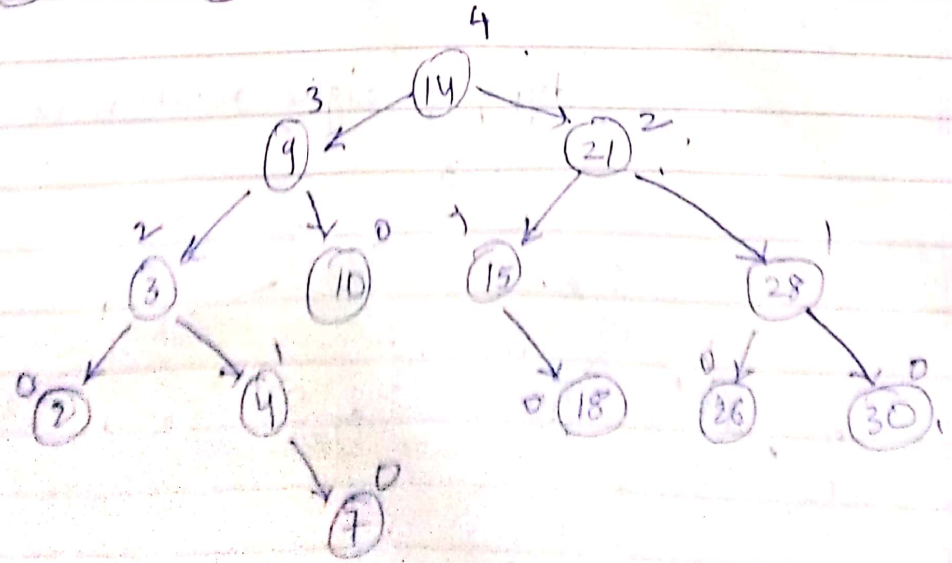
Applying left rotation at 2.



Applying right rotation at 4.

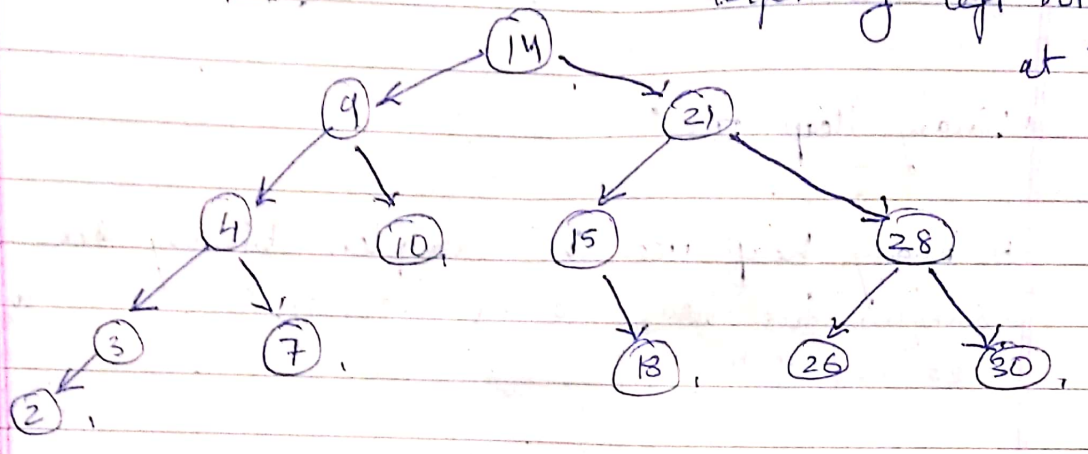


14.

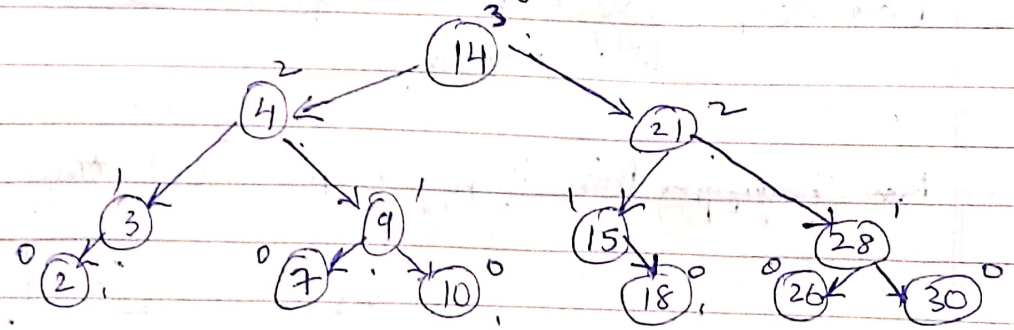


Ans. 4. continued ....

Performing left rotation at 3.

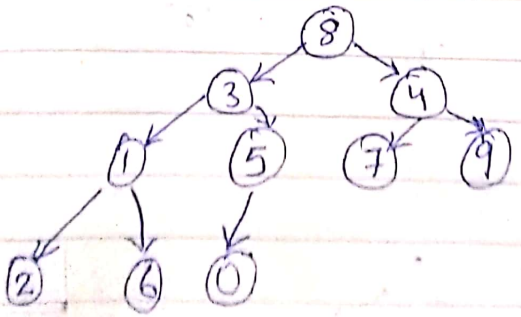


Now, performing right rotation at 9,



* Final Result *

Ans. 5. Array: [8, 3, 4, 1, 5, 7, 9, 2, 6, 0]



Complete binary tree version of array *.

Ans. 5. continued....

* Binary Heap tree *

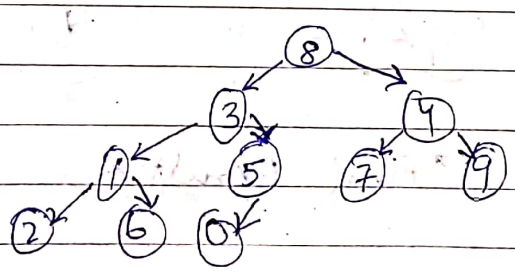
A binary heap tree is a complete binary tree that satisfies the heap ordering property. The ordering can be one of the following types:

Min Heap : Parent node is less than or equal to child nodes.

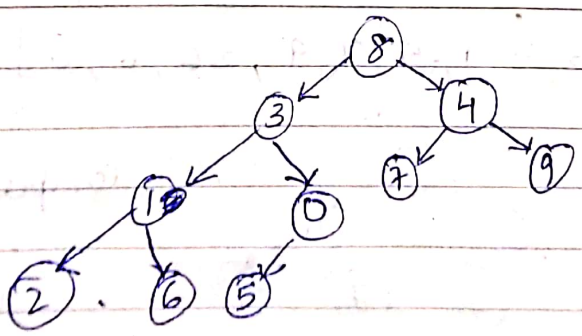
Max Heap : Parent node is Greater than or equal to child nodes.

~~Max Heapifying~~ (Min) Heapifying Given CBT from '5' (n/2).

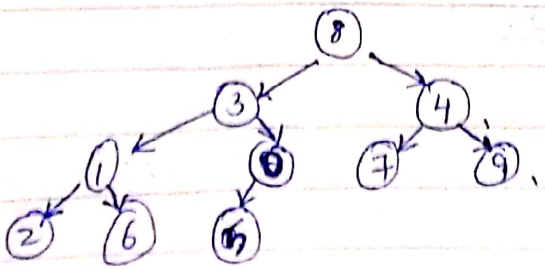
1)



i = 5



2)



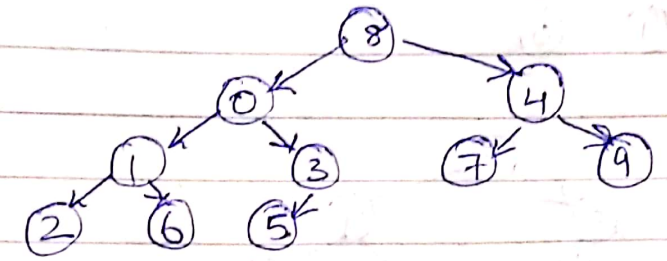
i = 4

* Already satisfied.

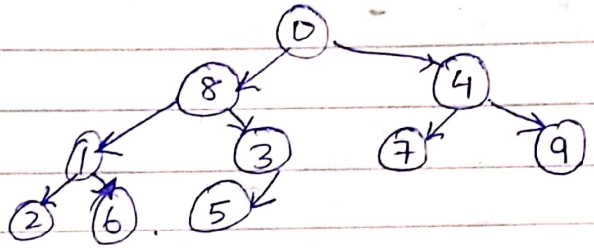
Ans. 5. Continued.

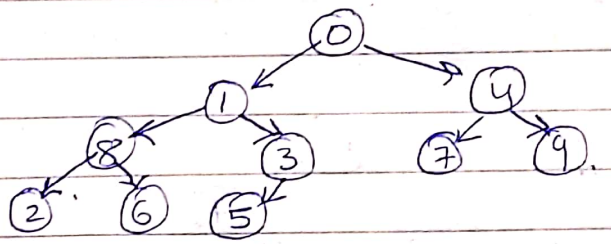
3) No changes. $i = 3$

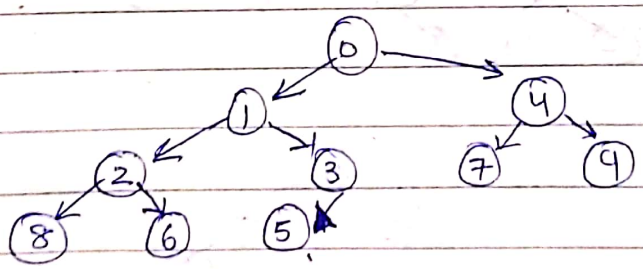
4) $p = 2$



5) $i = 1$.

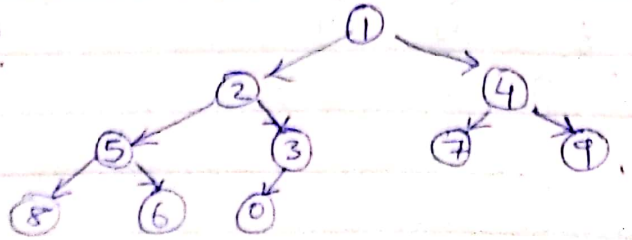




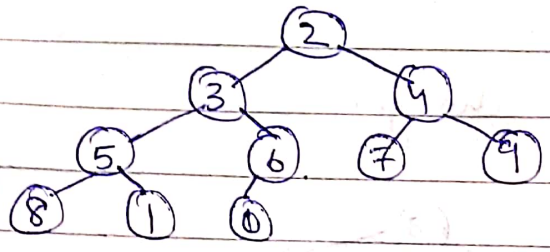


Swapping first and last element and applying Min Heapify function on root node, taking heap length as i for each 'i' iterations.

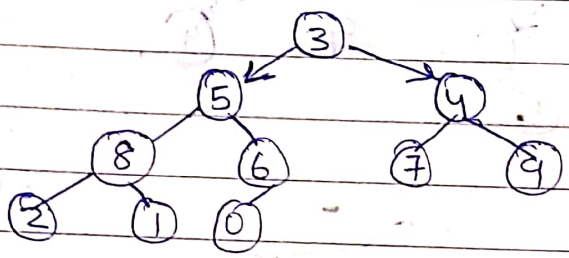
$i = 9$



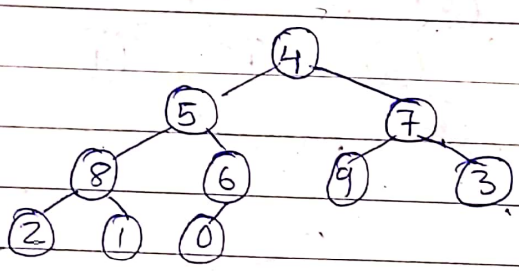
i = 8



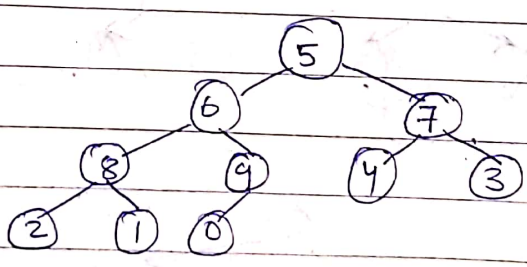
i = 7



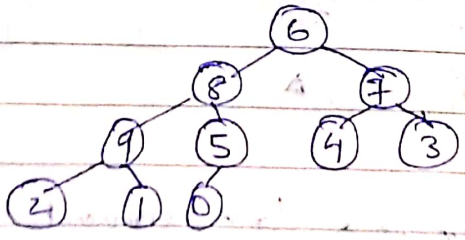
i = 6



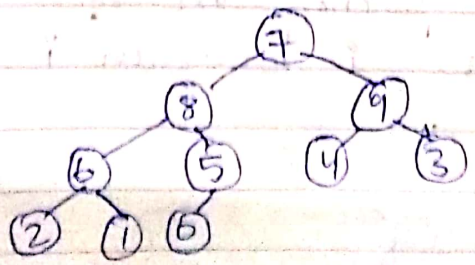
i = 5



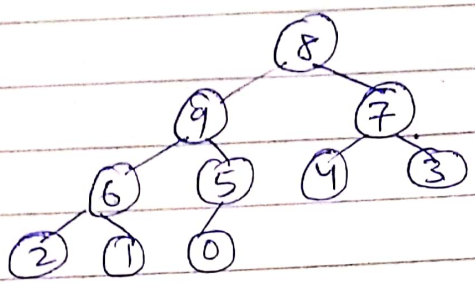
i = 4



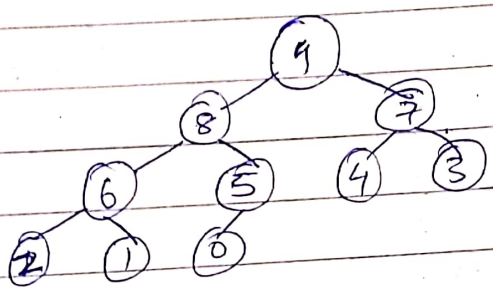
i = 3



i = 2



i = 1



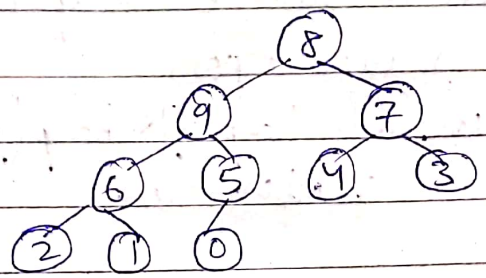
hence, the Heap array is now sorted in descending order.

Array (Final): [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

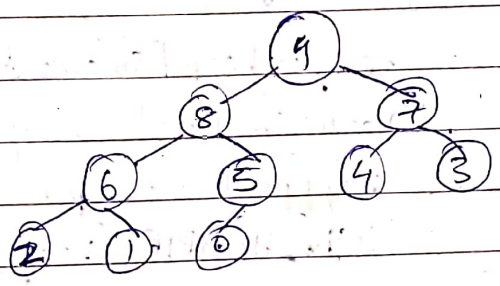
Extra Question - 1.

Write code to find kth largest element in an array through Heap Sort.

i = 2



i = 1



hence, the Heap array is now sorted in descending order.

Array (Final): [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Extra Question - 1.

#Q1 Write code to find kth largest element in an array through Heap Sort.

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

#define LEFT(pos)   (pos<<1)+1
#define RIGHT(pos)  (pos<<1)+2.

void MIN_HEAPIFY(int *heap, size_t len, size_t pos){
    size_t l = LEFT(pos);
    size_t r = RIGHT(pos), smallest{};
    size_t int temp{};
```

```
if(l<len && heap[l] < heap[pos]) smallest = l;
else smallest = pos;
if(r<len && heap[r] < heap[smallest])
        smallest = r;

if(smallest != pos){
        temp = heap[smallest];
        heap[smallest] = heap[pos];
        heap[pos] = temp;
        MIN_HEAPIFY(heap, len, smallest);
    }
}

void HEAP_SORT(int * heap, size_t len){

    for(int i = len/2 - 1; i>=0; i--)
        MIN_HEAPIFY(heap, len, i);

    int temp{};
    for(int i{len-1}; i>=0; --i){
        temp = heap[0];
        heap[0] = heap[i];
        heap[i] = temp;
        MIN_HEAPIFY(heap, i, 0);
    }

int main(){

    size_t size{};
    printf("Enter size of array: ");
    in>>size;
```

Ans. Q1. continued....

```
int arr[size];

printf("Enter Elements %llu for the array.\n",
                                        size);
for(size_t i{}; i<size; i++)
        cin>>arr[i];

HEAP_SORT(arr, size);

for(size_t i{}; i<size; i++)
        printf("%d ", arr[i]);
cout<<endl;

size_t pos{};
printf("Enter value of 'k' for kth Largest element:");
cin>>pos;

if(pos<size && pos)
        printf("Order %llu Largest element is %d\n", pos
                                        , arr[pos-1]);

return 0;

}
```

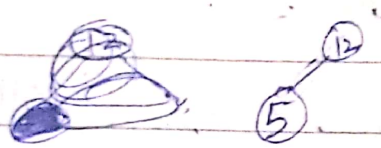
Q2. Create Max heap and Min heap from given data.

12, 5, 18, 19, 50, 9, 183, 17, 34, 54, 73, 84, 10.

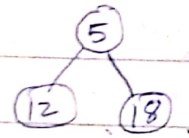
Ex. Ex. Max HEAP

1.

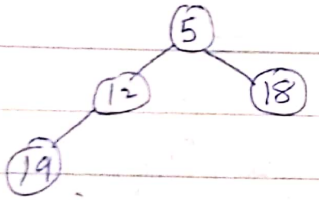
2.



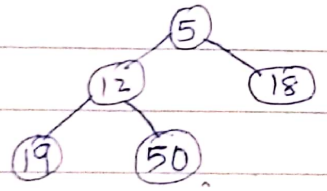
3.



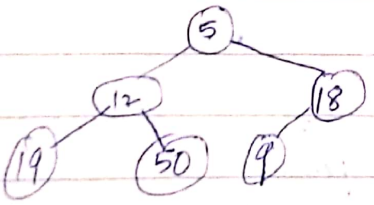
4.

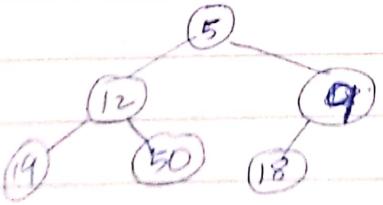


5.

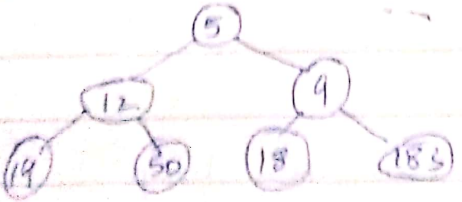


6.

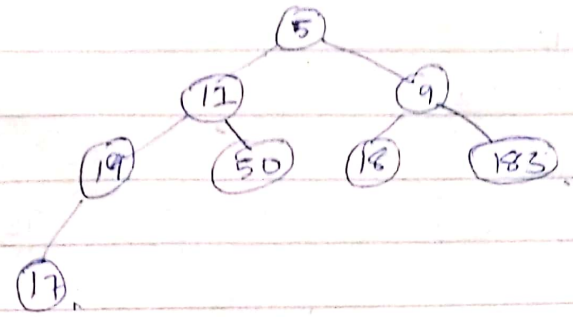


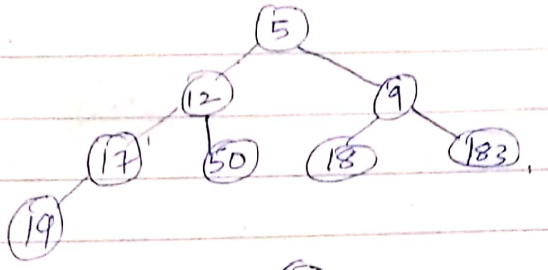


7.

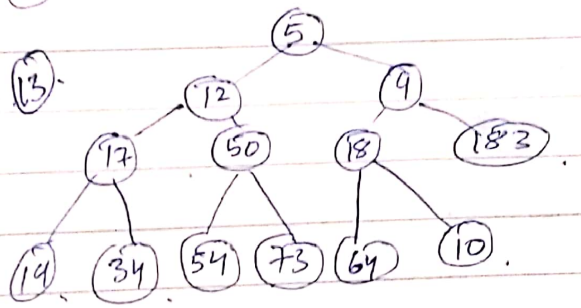


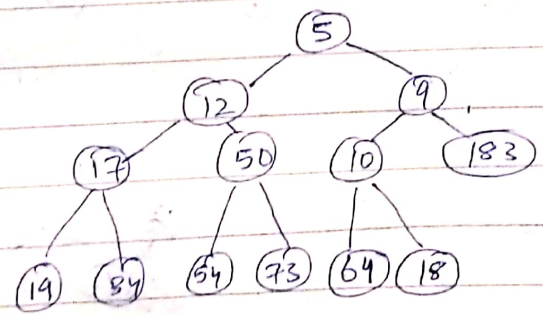
(8)





(9) ... (13).





Min Heap.

(1) 12

(2)

(3)

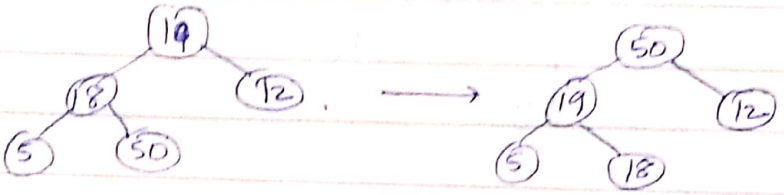
(3).


18
5
12

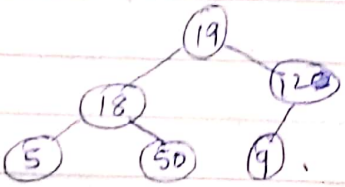
(4).


18
5
12
19
19
18
12
5

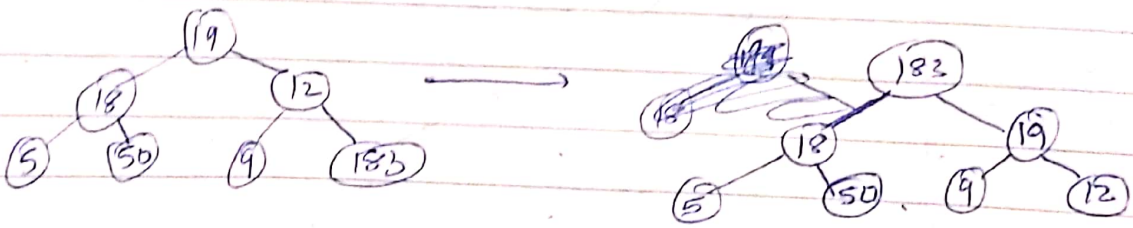
(5).


19
18
12
5
50
50
19
12
5
18

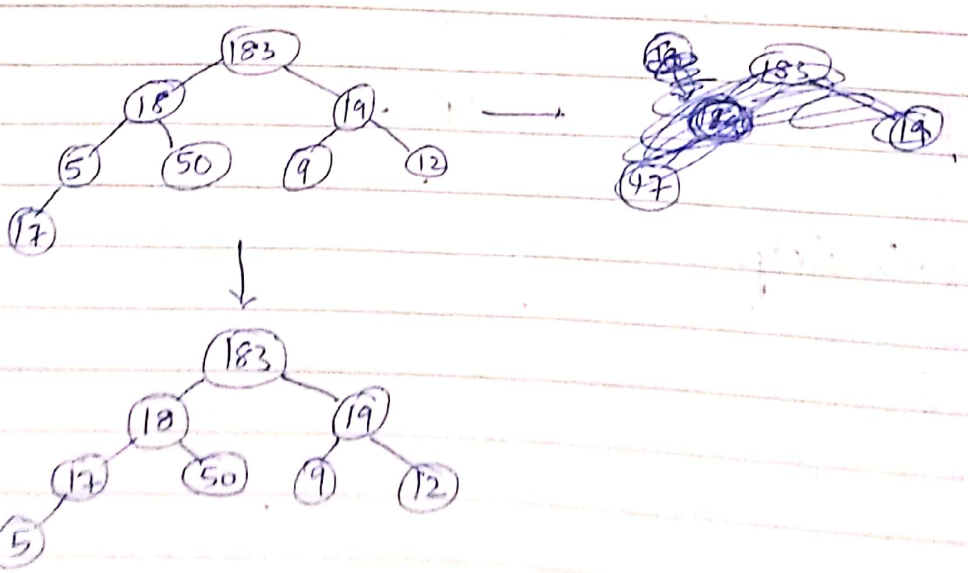
(6)


19
18
12
5
50
9

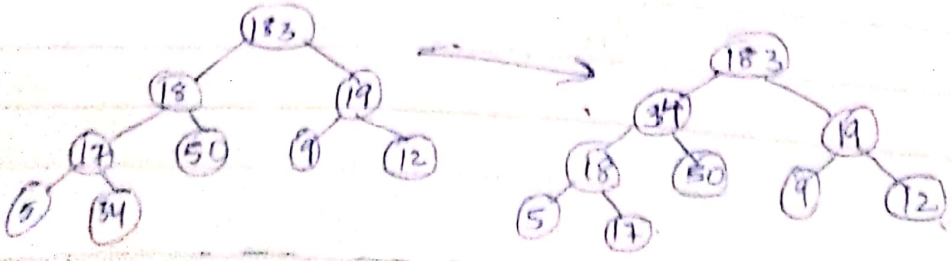
(7)


19
18
12
5
50
9
183
18
183
19
5
50
9
12

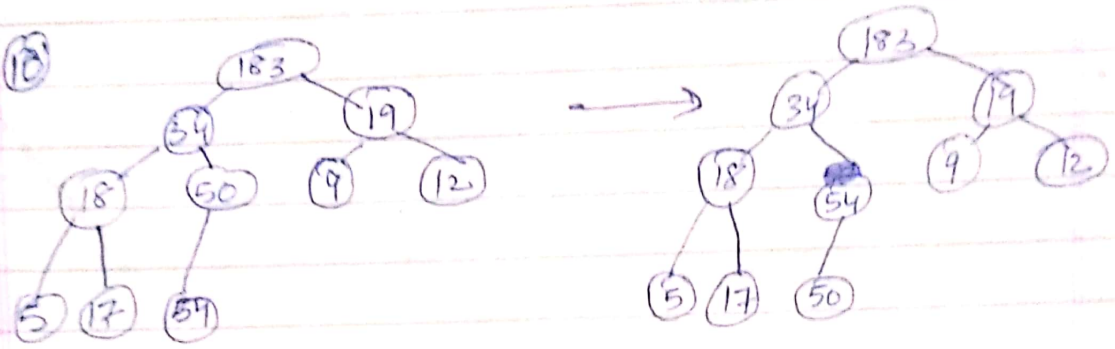
(8)


183
18
19
5
50
9
12
17
183
18
19
17
50
9
12
5

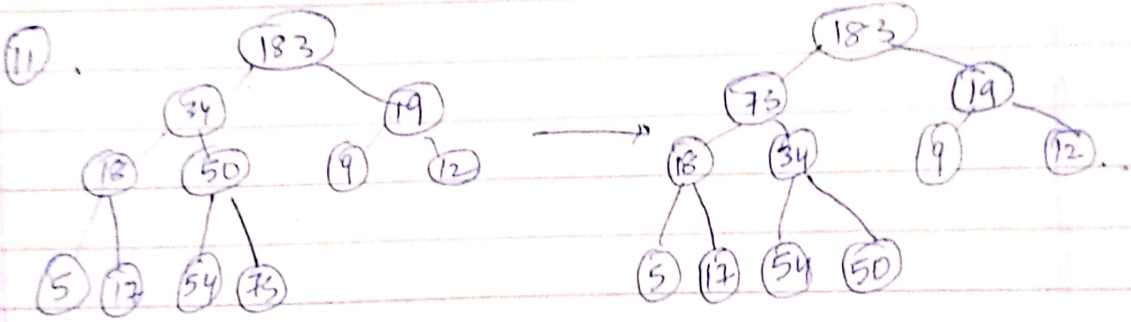
(9)


183
18
19
17
50
9
12
5
34
183
34
19
18
50
9
12
5
17

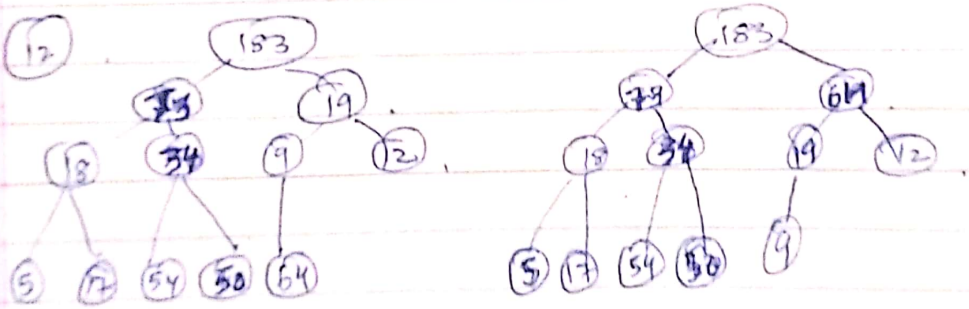
10.



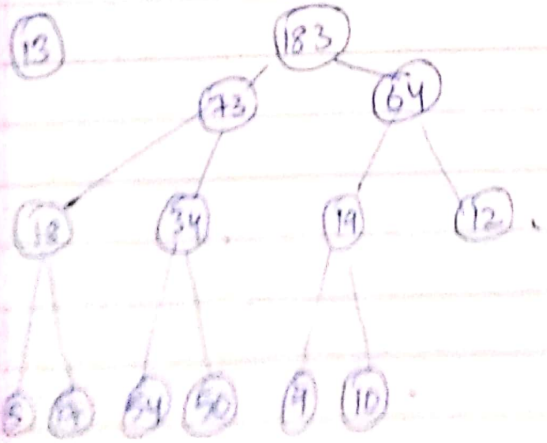
11.



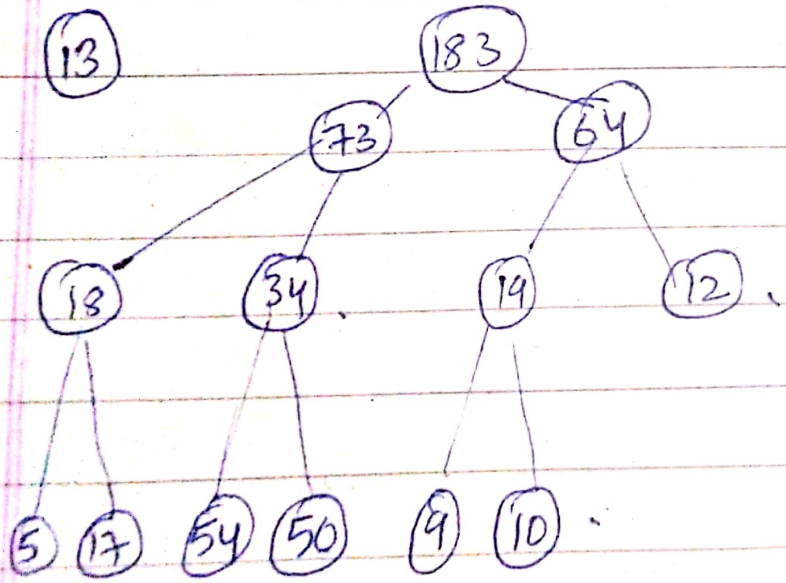
12.



13.



- Final Result.



* Final Result.