21/05/2023

Priority queue

It will work same as max-heap and minheap. STL implementation of heap.

We need to include header file queue.

```
#include <queue>
```

(i) Creation

```
priority_queue <int> pq;
```

By the above line, we have created a max heap.

(ii) Insertion

```
pq.push(3);
pq.push(5);
```

```
pq.push(9);
pq.push(4);
```

(iii) top element

```
cout << pq.top() << endl;  → 9
```

(iv) Deleting an element

```
pq.pop();  → 9 will be deleted
```

(v) Size of priority queue

```
cout << pq.size() << endl;
```

(vi) Check empty or not

```
pq.empty();  → Returns a boolean value
```

Min heap

```
priority_queue <int, vector<int>, greater<int>> pq;
```

The above syntax is to create a min heap

- int → integer data type is stored.
- vector <int> → container
- greater <int> → comparator to use min-heap

Q1 Find the k^{th} smallest number

i/p →

3	11	6	9	4	12	2	8

* Approach-1 ⇒ Sort the input array and simply return arr[k-1]. This has time complexity O(n log n)
* Approach-2 ⇒ Here if we use min-heap, we create heap of size = n.

1st pop, 1st smallest

⋮

kth pop, kth smallest

But by using min-heap space complexity O(n). ↳ quite confusing

By using max-heap, we can reduce this space complexity.

Approach-3

Make heap with first k elements. Now a new element is inserted in heap when heap top > new element.

Now space complexity = O(k)

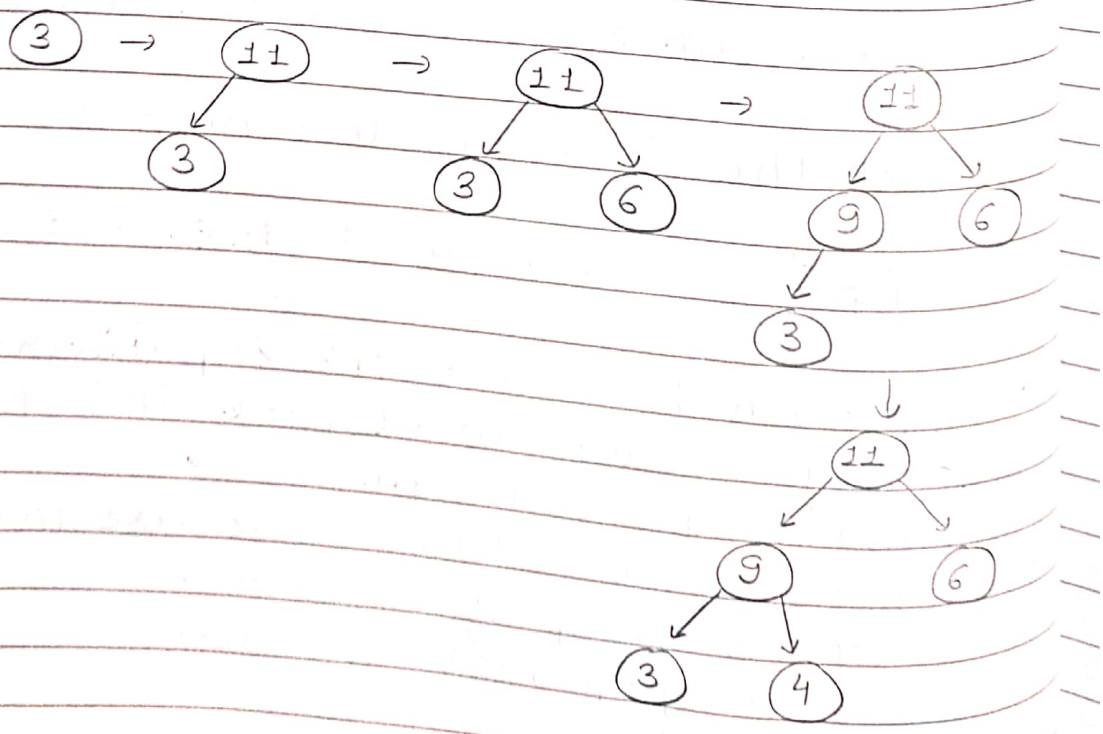
After traversing all the array elements, in max-heap k small elements are present. And top element will be the answer.

Dry run

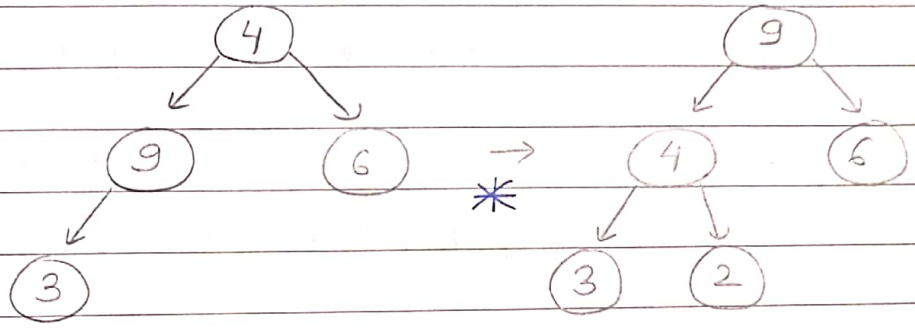
i/p → 3 11 6 9 4 12 2 8

k = 5

1)

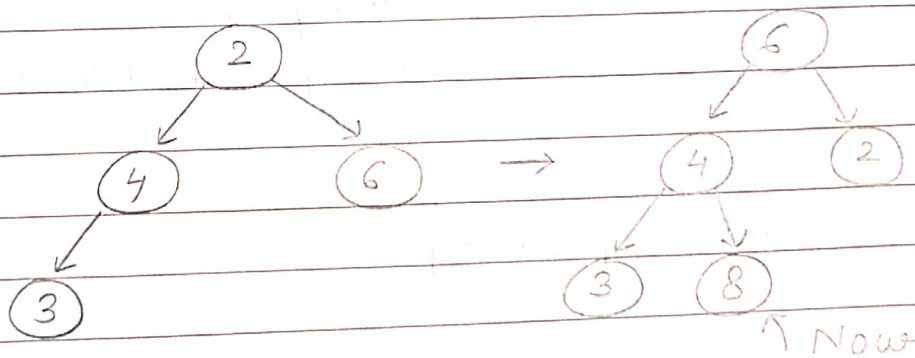


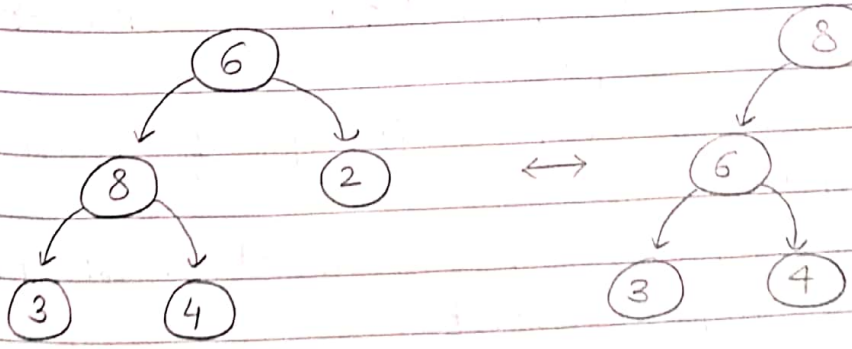
2) 12 is the next element. 11 > 12 (False) & hence do nothing
2 is the next element. 11 > 2 (True) & hence delete 11 and insert 2.



* Heapify + Insert 2

Now 8 is the next element. 9 > 8 (True) & hence delete 9 & insert 8.





⇒ Ans

Hence 5th smallest element is 8. (As we have traversed all the array elements)

$TC = n \times \log(k)$ → k is size of heap

Code

```
int findkthSmallest (int arr[], int n, int k
{
    //Create a max-heap
    priority_queue <int>pq;
    //Insert 1st k elements of array
    for (int i=0; i<k; i++){
            pq.push (arr[i]);
    }
    //Process remaining elements
    for (int i=k; i<n; i++){
    //Insert in heap if top > element
        if (pq.top() > arr[i]){
            pq.pop();
            pq.push (arr[i]);
        }
    } //Top element will be the answer
    return pq.top();
}
```

Q2 Find kth largest element in array

1) In the above code, instead of max-heap create a min heap.
2) Now just insert in the heap if heap top < element.
3) After traversing all the elements, simply return the top element.

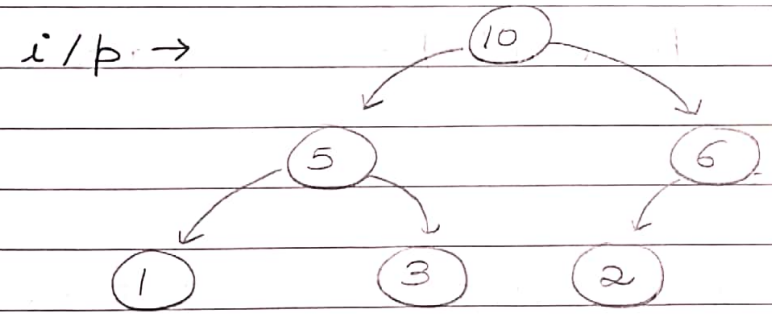
Q3 Merge 2 max heaps.

Approach-1 ⇒ Merge both the arrays and then apply buildHeap

arr → merge (heap1, heap2) ⇒ O(m+n)
Buildheap on arr ⇒ O(m+n)
O(m+n)

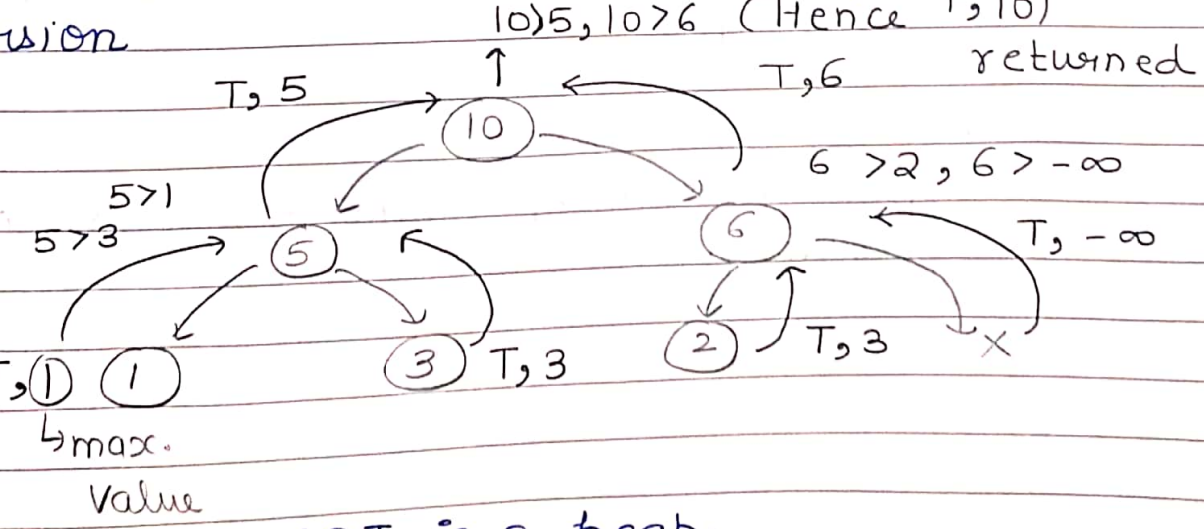
Approach-2 ⇒ 1st one is heap. Now just insert the elements of 2nd heap in the 1st heap. (Using STL, max-heap property would automatically be satisfied).

Q4 We are given a complete binary tree and we have to tell whether it is a heap or not.



2 conditions
1) Complete binary tree. → Already given
2) Heap property → We need to check

We can check the heap property via recursion



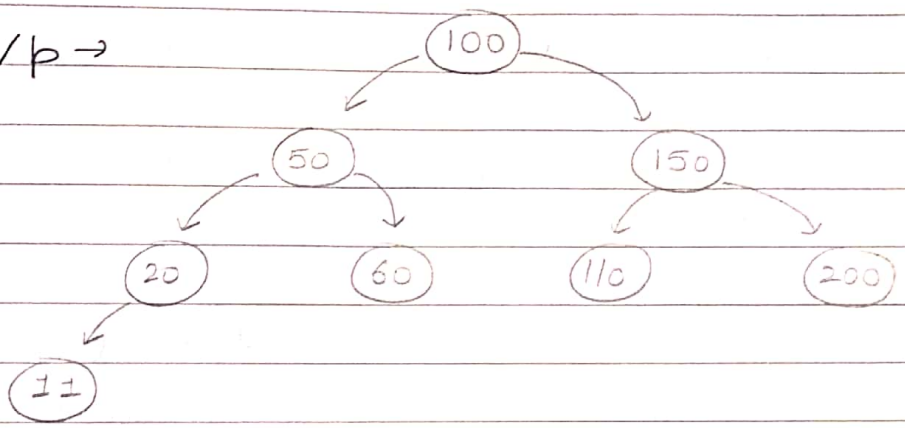
Hence above CBT is a heap.

Code

```
pair <bool, int> solve (Node * root){
    // Base case
    if (root == NULL) {
        pair <bool, int> p = make_pair (true,
                                        INT_MIN
        return p;
    }
    // Leaf node
    if (root -> left == NULL && root -> right ==
                                        NULL){
        pair <bool, int> p = make_pair (true, root
                                        ->data
    }
    //Solve for left and right subtrees
    pair <bool, int> lA = solve (root -> left);
    pair <bool, int> rA = solve (root -> right);
    //Check for conditions
    if (    lA.first && rA.first && lA.second
        < root -> data && rA.second < root -> data)
    { //Condition satisfied & hence return true
        pair <bool, int> p = make_pair (true, root->data
        return p;
    }
    else { // Not a heap
        pair <bool, int> p = make_pair (false, root->data)
        return p;
    }
}
```

Q5 Convert a BST into max-heap. Assume BST given is complete binary tree.

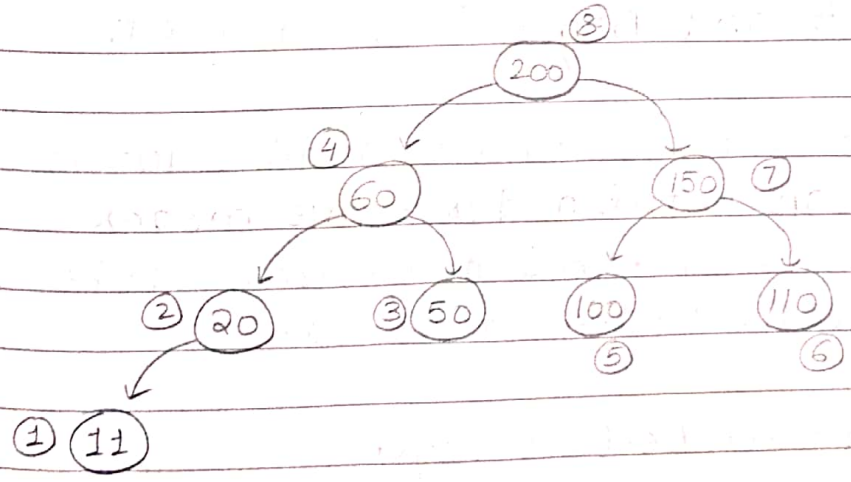
i/p →



1) As it is BST, the structure of max-heap will be same. We just have to place the values correctly.
2) Store the inorder traversal and insert the values in postorder style.

1 2 3 4 5 6 7 8

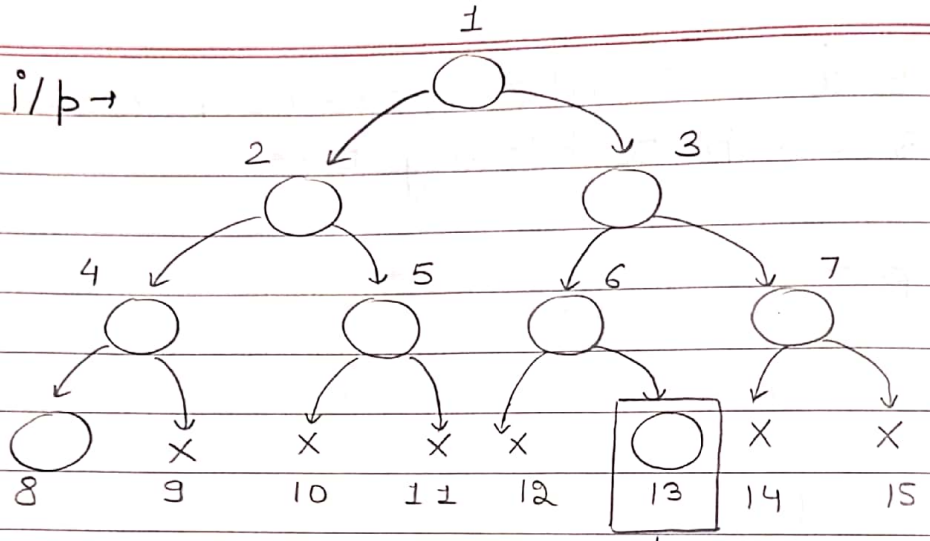
Inorder ⇒ 11, 20, 50, 60, 100, 110, 150, 200



Q6 Check whether a tree is a complete binary tree or not.

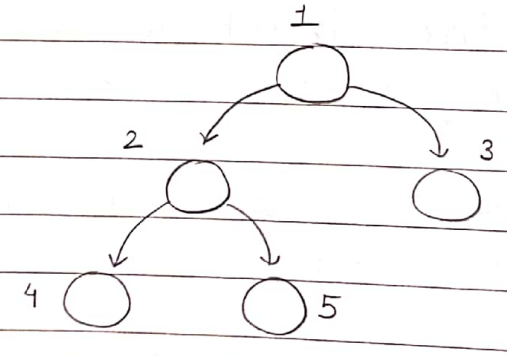
CBT → All levels filled (except last one) & filling should be done from left to right.

i/p →



↳ Total nodes = 9

13 > 9 and hence not a CBT.



Total nodes = 5

5 == 5 and hence it is a CBT.

If we get a node whose node count is greater than total nodes, this means it is not a CBT. This numbering can be done via level order traversal.

Q7 Merge k-sorted arrays.

arr1 → {2, 4, 6, 8, 10, 12}

arr 2 → {3, 9, 15, 18, 21, 24}

⋮

arr_k → {5, 20, 25, 30, 35, 40}

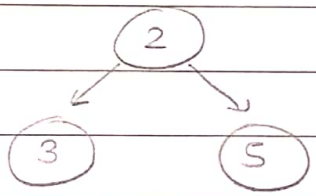
Approach-1 ⇒ Simply merge all arrays & then sort the merged array.

Time complexity = $nk \log(nk)$

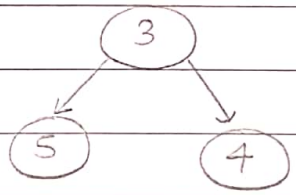
Approach-2

Simply insert k elements in the heap (min) and that to first element of each array.

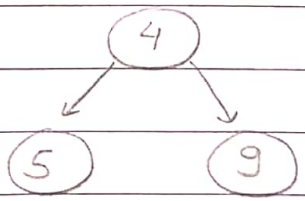
heap → {2, 3, 5}
ans → {2}



heap → {3, 5, 4}
ans → {2, 3}



heap → {4, 5, 9}
ans → {2, 3, 4}



⋮

Hence by this approach we can merge the k sorted arrays.