17/12/2023

* **Priority Queue**

Priority Queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user provided compare can be supplied to change the ordering. Eg- using std::greater<T> would cause the smallest element to appear as the top.

STL priority queue is the implementation of heap Data structure.

PS: **Container Adaptor** - Container adaptors provide a different interface for sequential container. Limited accessibility. Eg- stack, queue, priority queue.

Sequential containers - Sequence containers implement data structures which can be accessed sequentially.
Eg- array, vectors, list.

- **Creating max Heap -**

priority_queue <int> maxHeap ;

By default, priority queue max heap create karti hai, that's why container type and comparator likhne ki need nahi hai.

- **Creating min Heap -**

priority_queue <int, vector<int>, greater<int>> minHeap;

= int = defines type of stored element.
= vector<int> = container type.
= greater<int> = comparator.

Accessibilities -

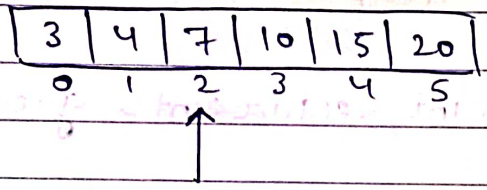
Push
Pop
top
size
empty

* kth smallest element (gfg) -

I/P - arr[] = 7 10 4 3 20 15
K = 3
l = 0 , r = 5 (l and r denotes starting and ending index of array)

O/P - 7

Explanation -



Approach 1 - Sort the array. Return arr[k-1].

3 4 7 10 15 20
↑
arr[k-1]

T.C. = O(n log n)

Approach 2 - Create min heap. Pop from min heap (K-1) times. Now, the top of min heap has kth smallest element.

T.C = O(n) + O((K-1) log n)

S.C = O(n)

Approach 3 - Space optimised approach

Create max-heap of first K elements in array to maintain initial state of heap.
For remaining elements, compare the top of max heap with current element in array.

```
if (maxHeap.top() > arr[i]){
        maxHeap.pop();
        maxHeap.push(arr[i]);
    }
```

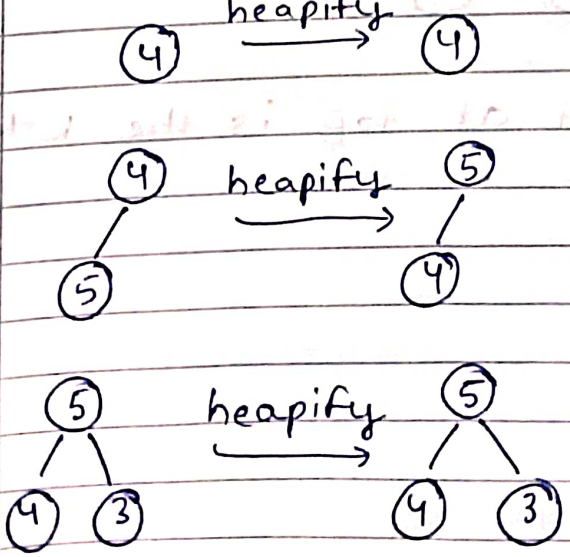
So, at last, the top of max Heap is the Kth smallest element.

T.C. = $O(K) + O((n-K)\log K) \simeq O(n \log K)$

S.C. = $O(K)$

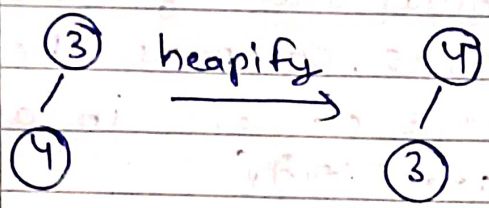
Let array, arr[] = {4, 5, 3, 1, 2}, K = 3

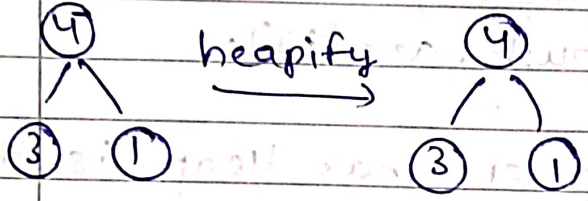
creating max-heap of first K elements -



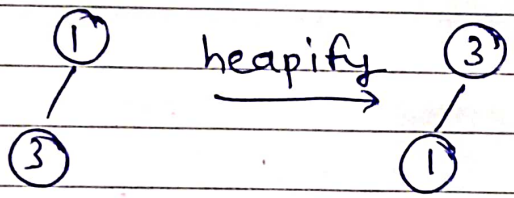
Now, rest of the steps -

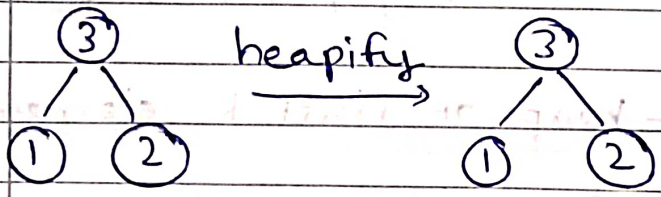
5 > 1 ✓





4 > 2 ✓





Now, the element at top is the K^{th} smallest element.

Code -

```
class Solution{
    public:

    int kthSmallest(int arr[], int l, int r, int k){
        priority_queue<int> maxHeap;

        for(int i=l; i<k; i++){
            maxHeap.push(arr[i]);
        }
        for(int i=k; i<=r; i++){
            if(maxHeap.top() > arr[i]){
                maxHeap.pop();
                maxHeap.push(arr[i]);
            }
        }
        return maxHeap.top();
    }
};
```

* Kth largest Element in an Array, (leetcode - 215)

I/P - nums = [3, 2, 1, 5, 6, 4], k = 2

O/P - 5

Approach - Create min heap of first k elements.
For remaining elements, check -

```
if (minHeap.top() < nums[i]) {
        minHeap.pop();
        minHeap.push(arr[i]);
    }
```

Now, the top of the min Heap is kth largest element.

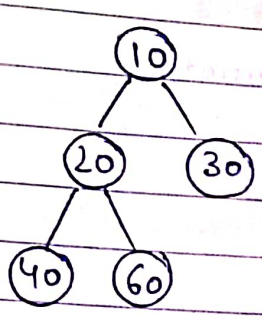
Code -

```
class Solution {
public:
  int findKthLargest(vector<int>& nums, int k) {
   priority_queue<int, vector<int>, greater<int> >
                                        minHeap;

    for(int i=0; i<k; i++){
        minHeap.push(nums[i]);
    }
    for(int i=k; i<nums.size(); i++){
        if(minHeap.top() < nums[i]){
            minHeap.pop();
            minHeap.push(arr[i]);
        }
    }
  return minHeap.top();
  }
};
```

* ...kiye hai, just go and check.

Is Binary Tree Heap (gfg) -

I/P -



O/P - false

For a Binary tree to be heap -

It must follows = CBT + Heap property (Max heap)

Approach -

Logic for complete Binary Tree -

Agr null ke baad koi non-null node mil jaye while traversing level order, then CBT nahi hai otherwise CBT hai.

logic for checking max heap -

For every node, agr parent node ki value children nodes ki value se greater hai, then max heap hai.

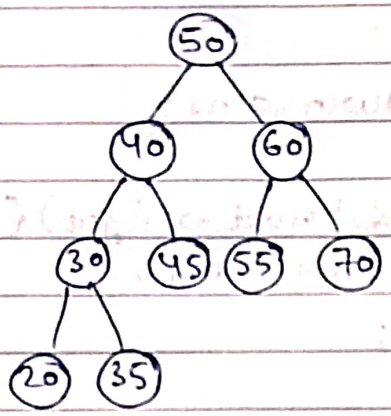
Code -

```
class Solution {
public: // for checking CBT
bool isCompleteTree(Node* root){
        queue<Node*> q;
        q.push(root);
        bool nullfound = 0;

        while(!q.empty()){
          Node* front = q.front();
          q.pop();

          if(!front){
             nullfound = 1;
          }
          else{
            if(nullfound) return 0;

            q.push(front->left);
            q.push(front->right);
          }
        }
      return 1;
    }

// this class will return two values for every node.
class Info{
public:
int maxVal;
bool isHeap;

Info(){
  this->maxVal = INT_MIN;
  this->isHeap = true;
}};
```

```
// for checking, whether max heap or not
Info checkMaxHeap (Node * root) {
    Info ans;

    if (!root) return ans;

    if (!root->left && !root->right) {
        ans.maxVal = root->data;
        return ans;
    }

    Info leftAns = checkMaxHeap(root->left);
    Info rightAns = checkMaxHeap(root->right);

if (root->data > leftAns.maxVal && root->data >
    rightAns.maxVal && leftAns.isHeap && rightAns.isHeap){
            ans.maxVal = root->data;
            return ans;
        }
else {
    ans.maxVal= max(root->data, max(leftAns.maxVal,
                                    rightAns.maxVal));
    ans.isHeap= false;
    return ans;
  }
}

bool isHeap(struct Node *tree){

    Info ans = checkMaxHeap(tree);
    return ans.isHeap && isCompleteTree(tree);
  }
};
```

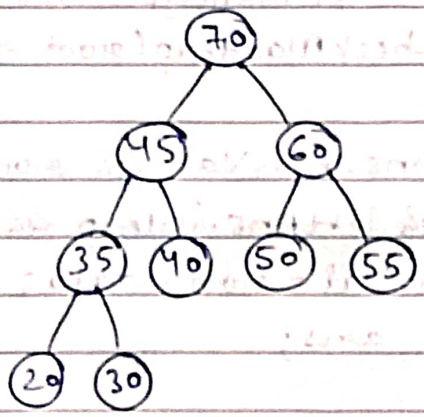
* Convert CBT + BST into max heap -.

I/P -



→ CBT + BST

O/P -



→ max heap

Approach -.

= Find inorder for the given CBT + BST.

= Replace the node's data using inorder array and by doing postorder traversal on tree.

Code -

```
class Solution{
public:
void findInorder(Node* root, vector<int>&inorder){
        if(!root) return;

        findInorder(root->left,inorder);
        inorder.push_back(root->data);
        findInorder(root->right, inorder);
    }

void replaceUsingPostorder(Node*root,vector<int>&inorder,
                           int&index){

    if(!root) return;

    replaceUsingPostorder(root->left,inorder,index);
    replaceUsingPostorder(root->right,inorder,index);
    root->data = inorder[index++];
}

void convertToMaxHeap(Node* root){

    vector<int>inorder;
    findInorder(root, inorder);

    int index = 0;
    replaceUsingPostorder(root,inorder,index);
}
};
```