Priority Queues

Example - 10 people are waiting for seat to get avigable for them in the plane.

1) FCFS
2) VIP Factor
3) Age.

FCFS → P_1 P_2 ——— P_{10}.
(First Come First Server)

Importance factor in FCFS is Time.

VIP Factor → P_1 P_2 P_3 P_4 P_5
(1) (2) (0) (8) (0) → VIP Factor
(whose VIP Factor is greatest will get the seat first)

Importance Factor in VIP Factor is value of VIP Factor.

Age → P_1 P_2 P_3 P_4 P_5
18 20 60 30 45 → Ages
(whose age is greatest will get the seat first and so on)

Importance Factor in Age is value of age of that person

Note: Har element ke saath uska importance factor save kar rahe hai aur uss importance factor ke bases pe he hum apne elements ko bahar nikala rahe hai.

Example- We have a hospital having 20 rooms. And all are currently booked (full hai) and we have 5 patients in waiting.

1) FCFS
2) Criticality Factor

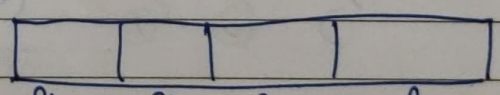
FCFS → P_1 P_2 P_3 P_4 P_5 (1 2 3)

Criticality Factor → P_1 P_2 P_3 P_4 P_5
(10) (5) (30) (20) (100) → Criticality Factor
(2) (1)
(we give room first to that patient whose criticality Factor is greatest)

Note: To deal such problems we have a special data structure called Priority Queues bcos we have now seen only FCFS shouldn't be the criteria for getting the elements out.

Importance Factor ← Priority Queues

Queue → FIFO
(First In First Out)



P_1, 50 P_2, 18 P_3, 20 P_4, 60 → Passenger → Priority

In Priority Queue, we are saving elements like queue only but hum hamare elements ko unki priority ke basis pe bahar nikala rahe hai. Priority is actually the importance factor

In Priority Queue, we will insert in simple Queue only i.e at last we will insert new element but nikalenge on basis of Priority i.e importance factor unlike in Queue we remove the first element first and so on.
Bahar wo phele nikalega in Priority jiss ki priority sab se zyada hai.

Priority Queue

Types of Priority Queue
- → Min → In this priority queue we will remove that element first whose priority i.e importance factor is minimum.
- → Max → In this priority queue we will remove that element first whose priority i.e importance factor is maximum.

Important Priority functions -

1) insert
2) getMax / getMin (Based on which priority queue we want to use) ← similar to top()
3) removeMax / removeMin ← similar to pop()

Ways to implement Priority Queues -

n numbers (n number will have n priority value)

14, 9, 8, ~~3~~, ~~1~~, ~~2~~, 20

Min
1
2
3

Max
14
20

~~14~~, 9, 8, 3, 1, 2, ~~20~~

↑ insert after removing 14.

DOMS Page No.
Date / /

	insert	getMin / getMax	removeMin / removeMax	(Overall) Complexity n no. of queries we have
Arrays (unsorted)	$O(1)$	$O(n)$	$O(n) \approx O(2n)$ $O(n)$ & $O(n)$ finding the element shifting all the elements	n^2
Arrays (Sorted) (Sorted based on its priority)	$O(n) \approx O(2n)$ we can need 2 transverals in worst case 1st traversal to find where to place and 2nd traversal to shift the elements	$O(1)$	$O(n)$ shifting of elements Ye case mai search nhi karna padega element bcos array is sorted.	n^2
Linked List (unsorted)	$O(1)$	$O(n)$	$O(n)$ Searching ke liye ye. We don't need to shift elements in LL.	n^2
Linked List (Sorted)	$O(n)$ Searching of correct position	$O(1)$	$O(1)$	n^2
BST	$O(h)$ height can be n in worst case.	$O(h)$ min element ke liye left-left jayenge. max element ke liye right-right jayenge	$O(h)$ Searching aur deleting mai	$n*h$
Best Balanced BST	$O(\log n)$ height in balanced BST is always $\log n$	$O(\log n)$	$O(\log n)$	$n \log n$
Hashmap	$O(1)$	$O(n)$	$O(n)$	n^2

Note: Balanced BST are those whose $|\text{left height of subtree} - \text{right height of subtree}| \leq 1$

Binary Tree means every node have maximum 2 nodes attached to it. Means 0, 1 and 2 children only are allowed.

Heaps Introduction -

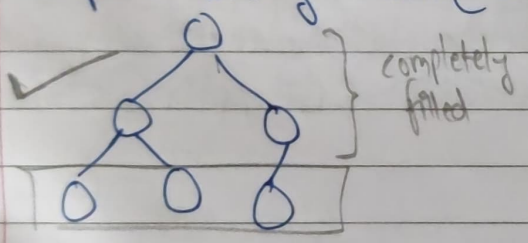
Using Balanced BST for making Priority Queue has two issues -
1) Balancing (After every insertion we balance hona chahiye)
2) Storing

For our Priority Queue we have design a new data structure i.e is named as Heaps.
Heaps is very similar to Binary Tree.

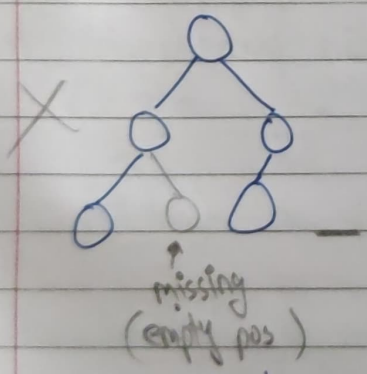
Heaps

1) Complete Binary Tree
2) Heap Order Property

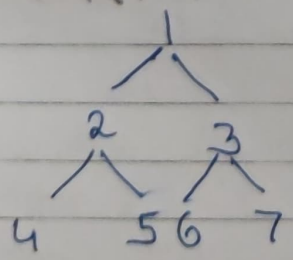
Complete Binary Tree (CBT)



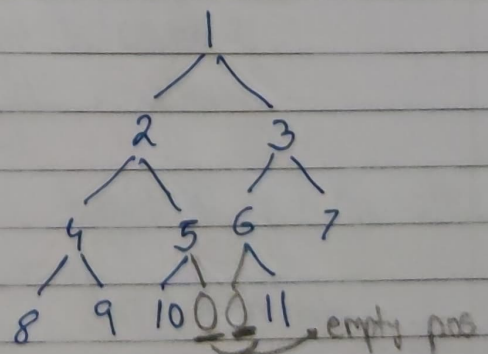
Complete Binary Tree means all levels must be filled completely expect the last level.
And last level too must be filled from left to right only.



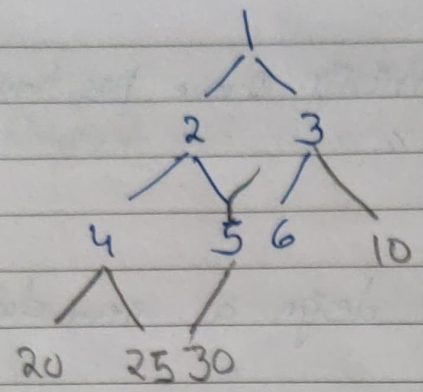
Not a Complete Binary Tree



This is a CBT

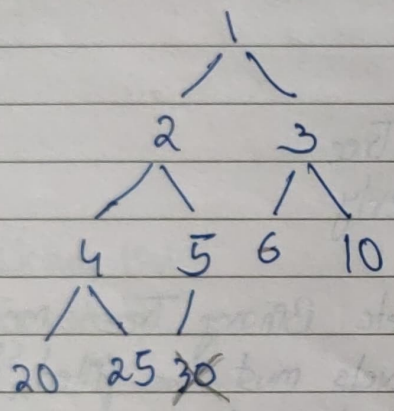


This is not a CBT.

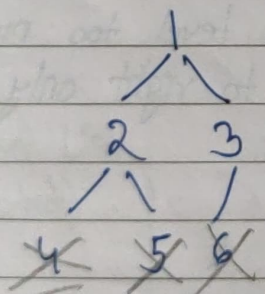


new element insert
10
20
25

Aise he elements add hone chahiye phir he woo CBT rahega.
Insert level-wise aur left to right he hoga



While deleting, we can't delete 6 elem and all.
While deleting, first we have to delete last element only i.e 30 to maintain CBT, phir 25 and so on.



This is a CBT

While deleting, first we will delete 6, then 5 then 4 to maintain CBT.

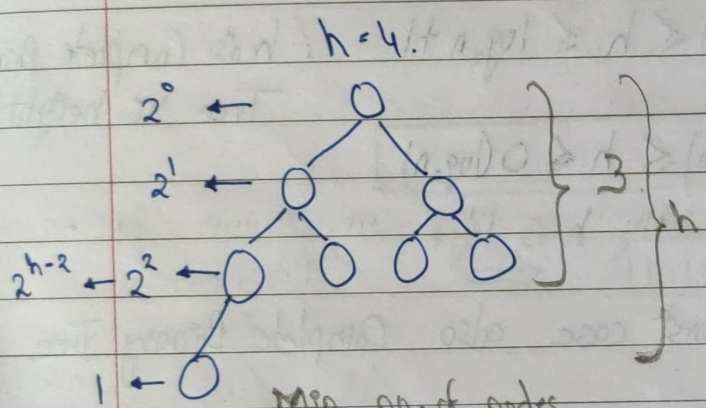
CBT and its implementation -

How CBT overcome these issues ← ~~Balanced~~ BST issues -
1) Balanced → Height
2) Store

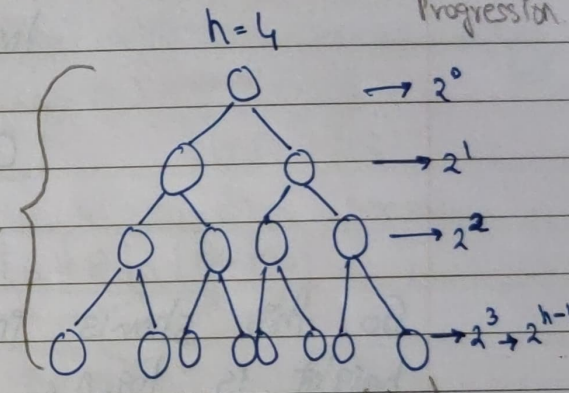
1) Height -

Minimum no. of nodes with height h CBT → $\underbrace{2^0+2^1+\ldots+2^{h-2}}_{\text{This is a Geometric Progression}}+1$

Maximum no. of nodes with height h CBT → $\underbrace{2^0+2^1+2^2+\ldots+2^{h-1}}_{\text{This is a Geometric Progression}}$



Min no. of nodes.



Max no. of nodes

Minimum no. of nodes with height h CBT -

$\underbrace{2^0+2^1+\ldots+2^{h-2}}_{\text{This is a GP}}+1 \Rightarrow \frac{2^0(2^{h-1}-1)}{2-1} = 2^{h-1} \cancel{-1}\cancel{+1} = 2^{h-1}$

∴ Minimum no. of nodes with height h CBT is 2^{h-1}

Maximum no. of nodes with height h CBT -

$2^0+2^1+\ldots+2^{h-1} \Rightarrow \frac{2^0(2^h-1)}{2-1} = 2^h-1$

∴ Maximum no. of nodes with height h CBT is 2^h-1

Min - 2^{h-1}

Max - 2^h-1

CBT → n
(CBT has n nodes)

$$2^{h-1} \le n \le 2^h-1$$

$2^{h-1} \leq n$

$\log_2(2^{h-1}) \leq \log_2 n$

$(h-1)\log_2(2) \leq \log_2 n$

$h-1 \leq \log_2 n$

$\boxed{h \leq \log_2 n + 1}$

$n \leq 2^h - 1$

$n+1 \leq 2^h$

$\log_2(n+1) \leq \log_2 2^h$

$\log_2(n+1) \leq h \log_2(2)$

$\boxed{\log_2(n+1) \leq h}$

$\log_2(n+1) \leq h \leq \log_2 n + 1$ (h is Complete Binary Tree ki height)

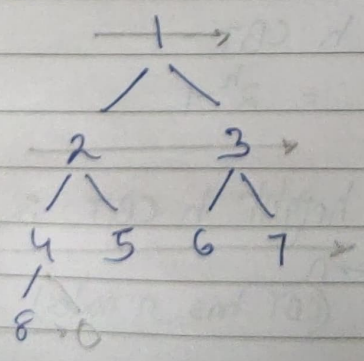
$\boxed{O(\log_2 n) \leq h \leq O(\log_2 n)}$

Strictly h is log n

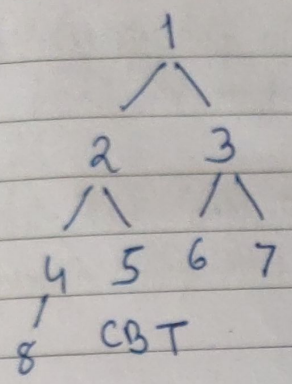
So, this shows in worst case also Complete Binary Tree height is $\log_2 n$.

Complete Binary Tree height is $\log_2 n$.
So, by doing this our one issue in BST is solved i.e balancing of BST.

2) Store -



If we travel level-wise to search empty position then this will take O(n) time to find empty position which is bad.



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Array

new node (insert)

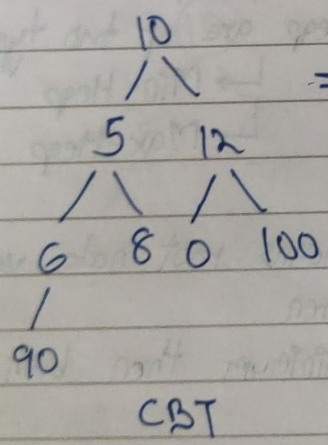
i $(2i+1, 2i+2)$

0th (index) → 1, 2 (indexes) (child hai)

1st index → 3, 4 indexes child hai.

2nd index → 5, 6 indexes child hai

- i th index → $(\underbrace{2i+1}_{\text{left child}}, \underbrace{2i+2}_{\text{right child}})$ indexes child hai



0	1	2	3	4	5	6	7
10	5	12	6	8	0	100	90

Array

next Index

In this way insert in CBT will be O(1)

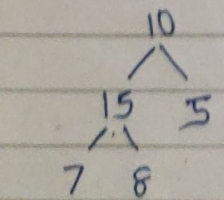
Note: We will be storing CBT in an Array but always visualize it as a Binary Tree only i.e CBT only.

CBT

10, 15, 5, 7, 8

Array

0	1	2	3	4
10	15	5	7	8



From Child to Parent

$\frac{2i+1}{2} \to i$

$\frac{2i+2}{2} \to i+1$

To overcome this problem, -1

$\text{parent-index} = \frac{\overset{\text{(child-index)}}{CI} - 1}{2}$

$\frac{2i+1-1}{2} = i$

$\frac{2i+2-1}{2} = i$

CI → ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5

$i \to 2i+1, 2i+2$

Parent — left Child — Right Child

3 → 7, 8

1 → 3, 4

Note: $\text{Parent-index} = \frac{(\text{Child-index})-1}{2}$ (To find parent-index from child-index)

$\text{Parent-index}(i) = \underbrace{2i+1}_{\text{left child}}, \underbrace{2i+2}_{\text{Right child}}$ (To find child-index from parent-index)

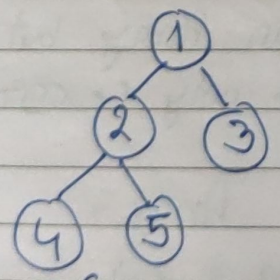
Insert and Delete in Heaps -

Priority Queues are of two types -
- → Min
- → Max.

Heap
1) Complete Binary Tree
2) Heap Order property

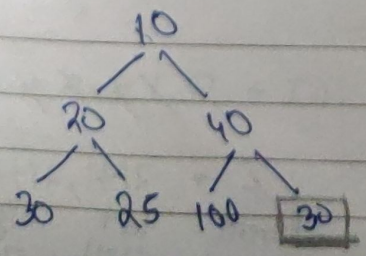
Heap are of two types - (on basis of Heap order property)
- → Min-Heap
- → Max-Heap

- Heap is a Complete Binary Tree.
- Heap order property of Min-Heap means root node value should be minimum than it's children.
  i.e Parent node value should be minimum then both of it's children node value.

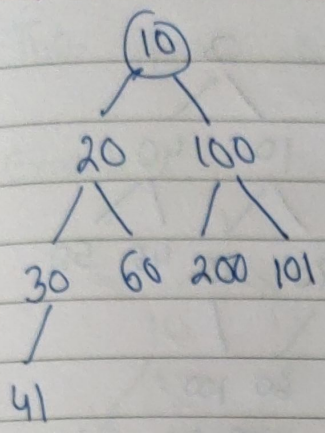


(Min-Heap)

→ This is a valid Min-Heap bcos it satisfies both the properties i.e
1) Complete Binary Tree
2) Heap Order Property

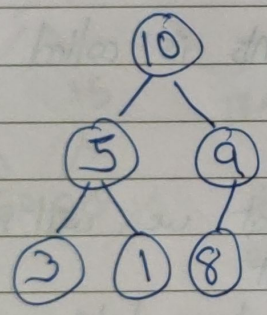


This is not a valid Min-Heap. Bcos of 30 is smaller than 40



This is a valid Min-Heap

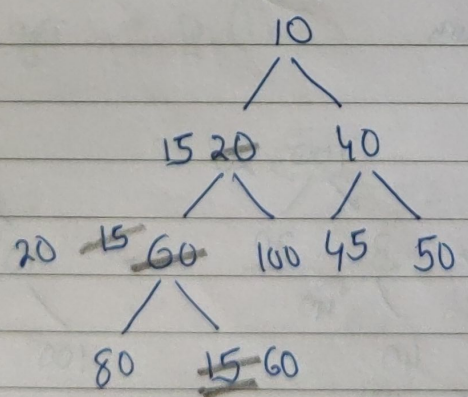
- Heap order property of Max-Heap means root node value should be maximum than it's children.
  i.e Parent node value should be maximum then both of it's children node value.



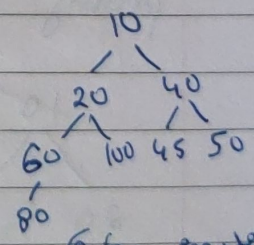
→ This is a ~~valid~~ valid Max-Heap bos it satisfies both the properties i.e
1) Complete Binary Tree
2) Heap order Property

Insert →

15



Original Heap -

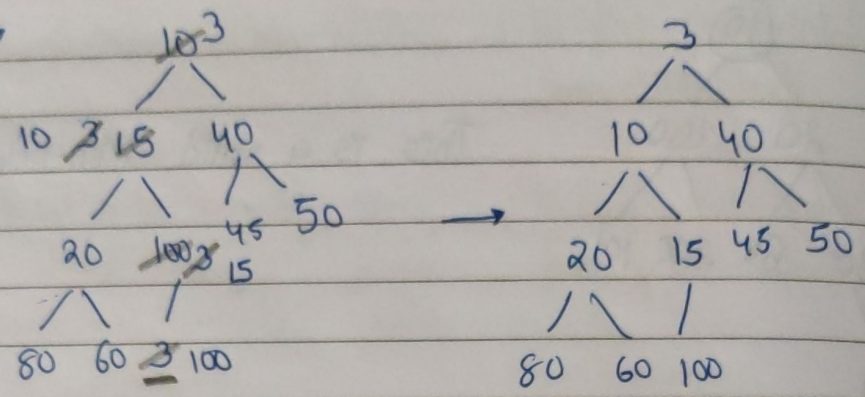


(Before inserting)

After inserting 15 here CBT property is followed but Heap order property is voilated.
So, we change Parent and child i.e we swap 15 and 60 to follow Min-Heap property

Now, both properties are satisfied.

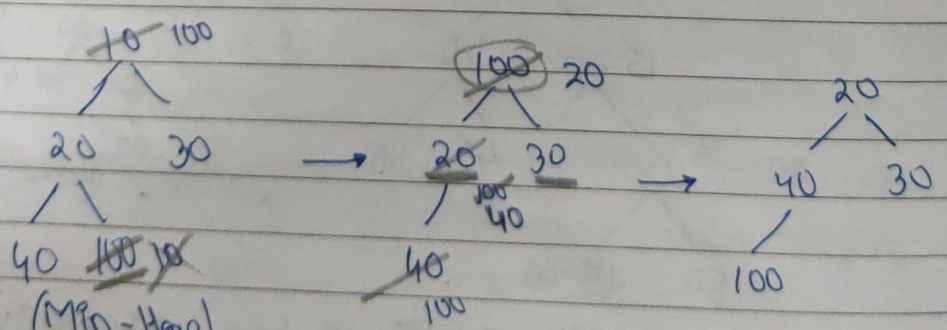
Insert → 3



Insert function Time Complexity is O(h)
↓
O(logn)

Insert toh O(1) mai ho jata hai but usko right place par place karne ke liye toogle karna padata hai usse liye O(h) lagega.
This toogaling and swaping elements is called up-Heapify.

Delete →
In Priority Queue we have dicussed, that we will remove that element whose Priority is most important
- Min-Heap mai wo element remove hona chahiye whose priority is minimum
- Max-Heap mai wo element remove hona chahiye whose priority is maximum

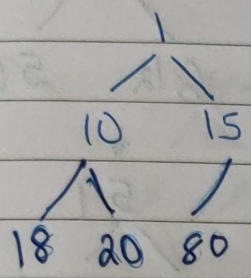


(Min-Heap)

To delete first we will swap root value and last node value. And the delete the last node. After this CBT is satisfied by Heap order property is violated.
To overcome this violated property we will compare Parent with child after swap if is violated i.e swaping Parent node with smallest child.

This process is known as Down-Heapify.

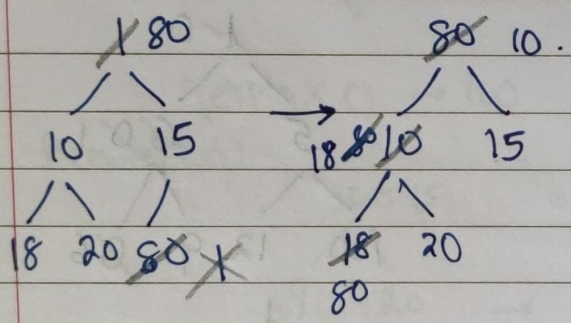
Example -



(Min-Heap)

Note:
minimum element in Min-Heap is the root node.

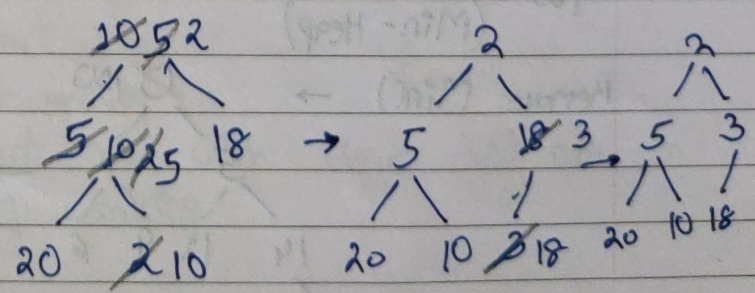
get function will only read min value of Min-Heap and it will take O(1) time



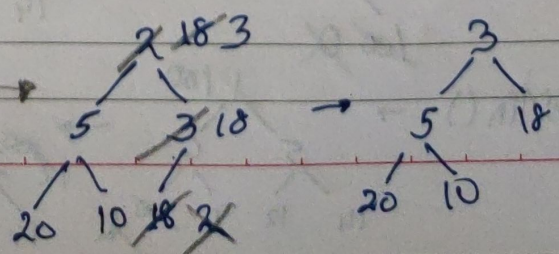
Delete function is taking O(h) time i.e O(logn) time.

Delete → O(logn)
Insert → O(logn)
get → O(1)

Min-Heap -
10, 5, 18, 20, 2
3,



getMin() → 2
RemoveMin()

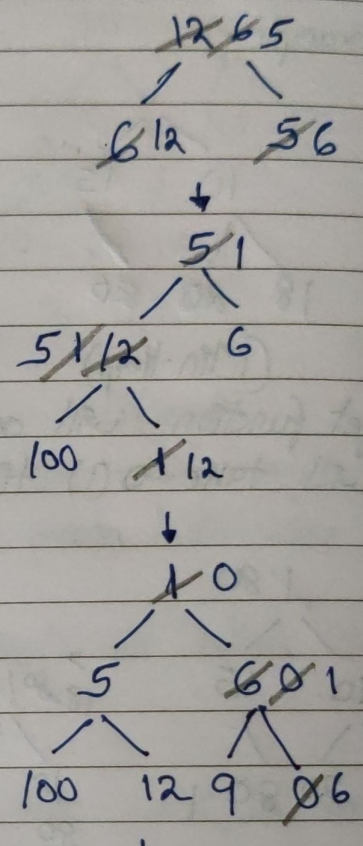


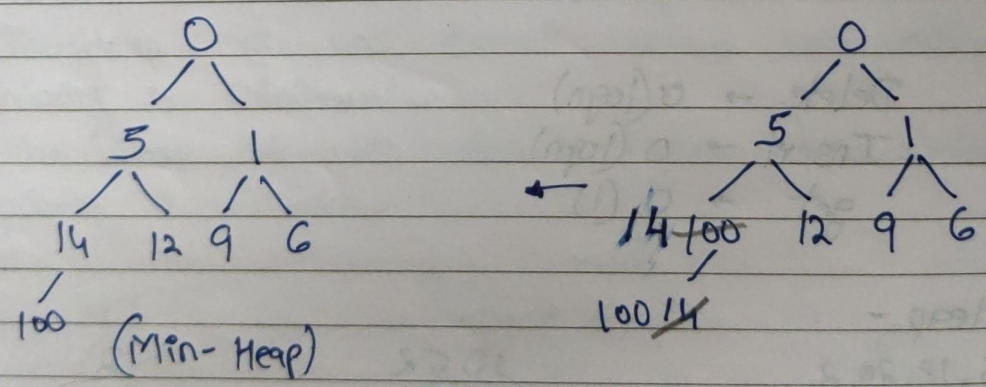
Min-Heap-

12, 6, 5, 100, 1, 9, 0, 14.

Create Heap with these elements

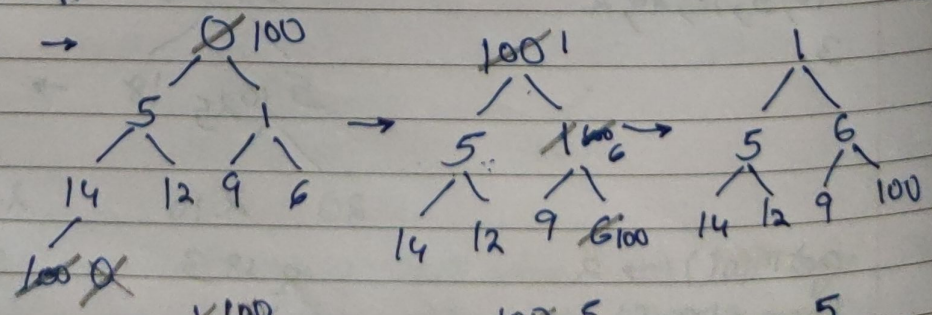
RemoveMin() → 2



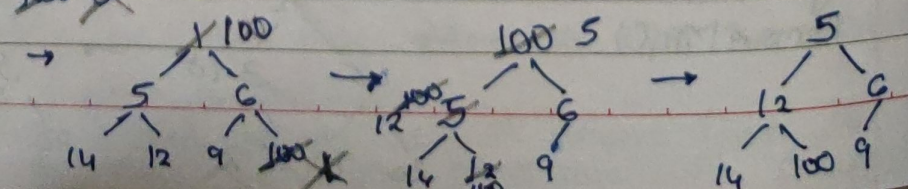


(Min-Heap)

Remove Min() →

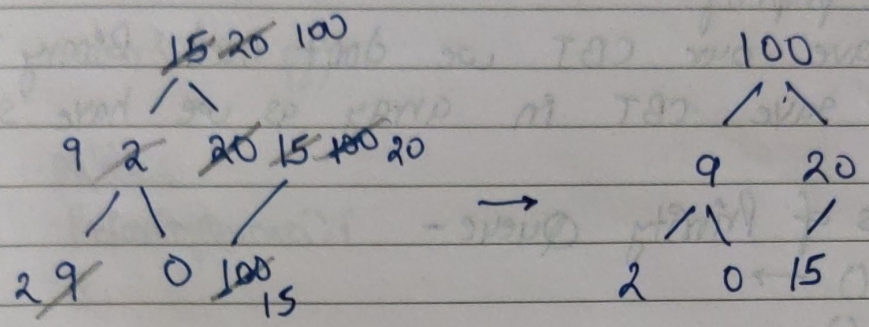


RemoveMin() →



Example Solution and Max Heap-

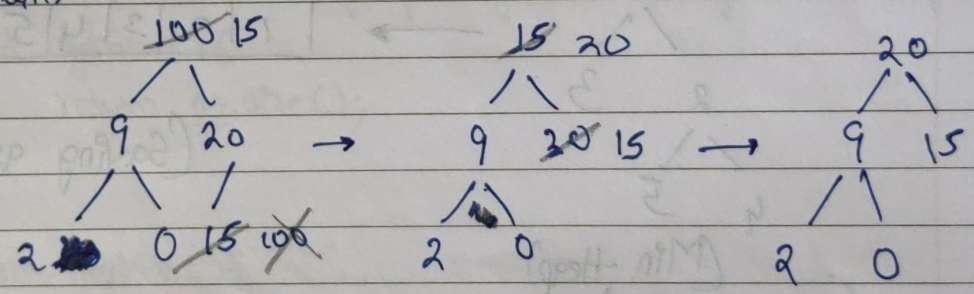
Max Heap → 15, 2, 20, 9, 0, 100



Note - ~~getMa~~ In Max-Heap, maximum element is at the root only

getMax () → 100

RemoveMax()



When removeMax() function is called we will swap root node with last node and the delete last node. It's till a CBT ~~by~~ but Heap order porperty isn't working So we will compare and swap Parent node with biggest child node to make Heap order property working

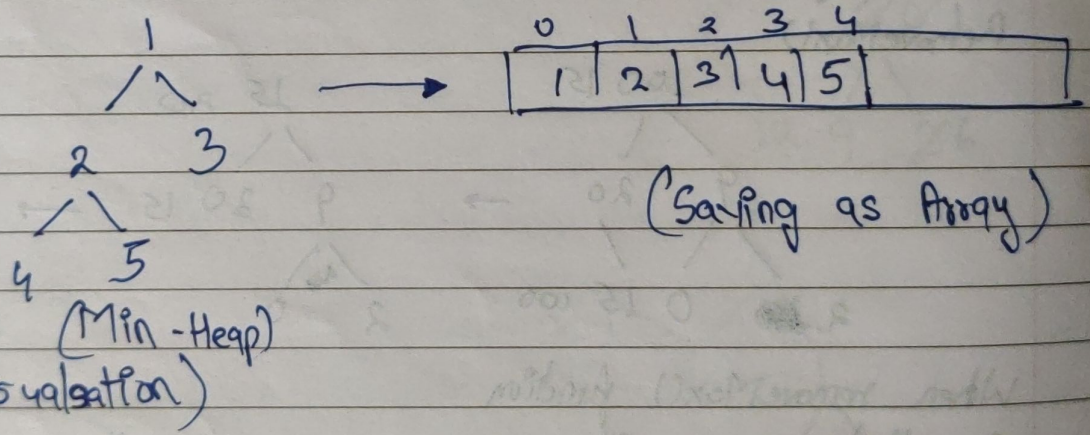
Implementation of Priority Queues-1

To implement Priority Queue we need Heap and Heap should satisfied both it's properties i.e CBT and Heap order property.

To save our CBT we don't need Binary Tree, we can save CBT in array as we have seen this.

Functions of Priority Queue -

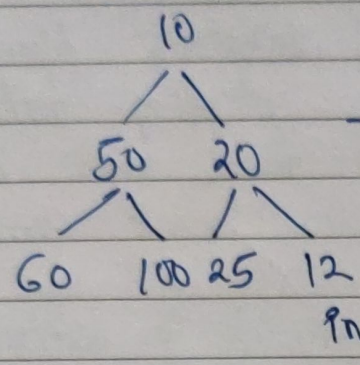
insert() →
delete() →
getMin() →
isEmpty() →
getSize() →



Note: Always remember internally we are storing Heap as Array but we are visualize it as a Binary Tree only.

```
# include <vector>

class PriorityQueue{

    vector<int> pq;

    public:
    PriorityQueue(){
    }

    bool isEmpty(){
        return pq.size()==0;
                ↳ 0==0 → T
                  1==0 → F
    }

    // Return the size of priorityQueue- no. of elements present
    int getSize(){
        return pq.size();
    }

    int getMin(){
        if (isEmpty()){

            return 0;
        }
        return pq[0]; // Priority Queue is empty
    }
};
```

Implementation - Insert



pq → (vector)

swap

0	1	3	3	4	5	6
10	50	~~20~~ 12	60	100	25	~~12~~ 20

$i \to 2i+1, 2i+2$

$\frac{ChildI-1}{2}$

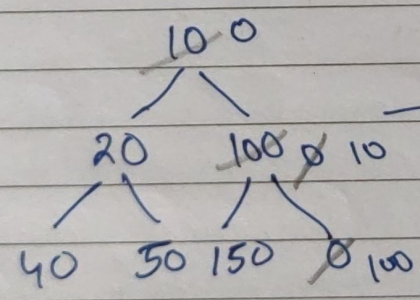
$\frac{6-1}{2} = 2.$

CHI → 6

~~pIIx~~ . pI = 2.

CI	pI
~~6~~	2
2	0

Example-



0	1	2	3	4	5	6.
~~10~~ 0	20	~~100~~ 0	40 10	50	150	~~0~~ 100

(internally storing)

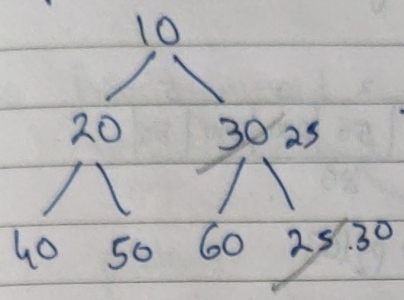
CI	pI
~~6~~	2
~~2~~	0
0	

New element is inserted and both properties of Heap are satisfied

Ye hya ye case jab humko toogle karte karte root tak jana hai

```
while (childIndex>0) {
        if (pq [childIndex] < pq [parentIndex])
            swap)
}
```

Example -



→

0	1	2	3	4	5	6
10	20	~~30~~ 25	40	50	60	~~25~~ 30

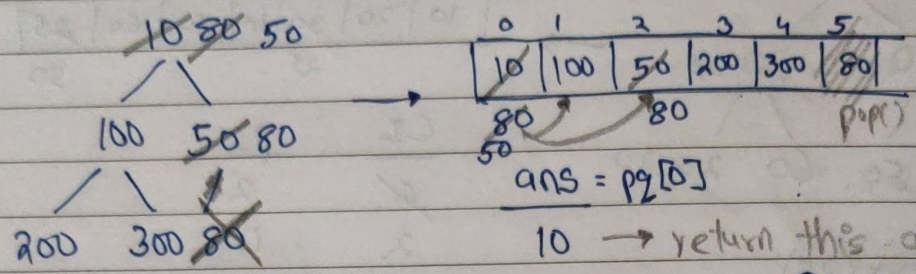
CI	PI
~~6~~ 2	2 0

insert (25)

Ye case mai humko root tak toogle karne ki zarrot nhi hai.

```
while (childIndex>0){
        if ( pq [childIndex] < pq[parentIndex])
                swapping
        else
                break; // when we don't need
                          to toogle till root
}
class PriorityQueue{
        : Refer previous code of class PriorityQueue here
void insert (int element){
    pq. push_back(element);

    int childIndex = pq. size() -1;
    while (childIndex >0){
        int parentIndex = (childIndex -1)/2;
        if (pq [childIndex] < pq [parentIndex]){
            int temp = pq [childIndex];
            pq [childIndex] = pq [parentIndex];
            pq [parentIndex] = temp;
        }
        else {
            break;
        }
        childIndex = parentIndex;
}
}
}
```

Remove Min Solution and Complexity -



ans = pq[0]

10 → return this as ans bcos this element is removed

pq[0] = pq[pq.size()-1]

pq.pop_back();

(parentIndex) (leftchildIndex) (Rightchild Index)

PI	LCI	RCI	minIndex
i	(2*i+1)	(2*i+2)	
0	1	2	∅ (initial)
2	5	6	2

Leaf Node

Note: If leftchild Index is not existing we can surely say that rightchild Index will also be not existing bcos to maintain CBT.

Priority Queue is Heap Sort.
Sort means increasing or decreasing order mai element.
Internally in Min-Heap when we remove elements they come out in increasing order i.e Heap Sort.
Internally in Max-Heap when we remove elements they come out in decreasing order i.e Heap Sort.

- Internally Priority Queue is doing sort and this is called Heap Sort.
- Heap Sort -

Time Complexity - $\underbrace{n\log n}_{\text{Inserting n elements in Heap}} + \underbrace{n\log n}_{\text{Removing n elements from Heap}} \approx O(n\log n)$

Space Complexity - $O(n)$ → Bcos we are using a vector.

PriorityQueue.h File [Min-Priority Queue]

```
# include <vector>

class PriorityQueue {
    vector<int> pq;

    public:
    PriorityQueue(){
    }

    bool isEmpty(){
        return pq.size()==0;
    }

    // Return the size of priorityQueue - no. of elements present
    int getSize(){
        return pq.size();
    }

    int getMin(){
        if (isEmpty()){
            return 0;      // Priority Queue is empty
        }
        return pq[0];
    }

    void insert (int element){
        pq.push_back(element);

        int childIndex = pq.size()-1;
```

```
while (childIndex > 0) {
    int parentIndex = (childIndex -1)/2;

    if (pq[childIndex] < pq[parentIndex]){
        int temp = pq[childIndex];
        pq[childIndex] = pq[parentIndex];
        pq[parentIndex] = temp;
    }
    else {
        break;    // when we don't need to toogle
                  till root
    }
    childIndex = parentIndex;
}

int removeMin(){
    if (isEmpty()){
        return 0;        // Priority Queue is empty
    }
    int ans = pq[0];
    pq[0] = pq[pq.size()-1]
    pq.pop_back();

    // down-heapify

    int parentIndex = 0;
    int leftChildIndex = 2* parentIndex +1;
    int rightChildIndex = 2* parentIndex +1;

    while (leftChildIndex < pq.size()){
        int minIndex = parentIndex;
        if (pq[minIndex] > pq[leftChildIndex]) {
            minIndex = leftChildIndex;
        }
```

```
        if (rightChildIndex < pq.size() && pq[rightChildIndex] <
                                                pq[minIndex]){
            minIndex = rightChildIndex;
        }
        if (minIndex == parentIndex){
            break;
        }
        int temp = pq[minIndex];
        pq[minIndex] = pq[parentIndex];
        pq[parentIndex] = temp;

        parentIndex = minIndex;
        leftChildIndex = 2* parentIndex +1;
        rightChildIndex = 2* parentIndex +2;
    }
    return ans;
  }
};
# include <iostream>
using namespace std;
# include "PriorityQueueh"
int main() {
    PriorityQueue p;
    p.insert(100);
    p.insert(10);
    p.insert(15);
    p.insert(4);
    cout << p.getSize() << endl;
    cout << p.getMin() << endl;
    while(! p.isEmpty()) {
        cout << p.removeMin() << " ";
    }
    cout << endl;
}
```

Heap Sort -
Elements comes out in ascending order

DOMS Page No.
Date / /

Max-Priority Queue
PriorityQueue.h File

```
# include <vector>

class PriorityQueue{
        vector <int> pq;

        public:
        PriorityQueue(){
        }

        void insert (int element){
            pq.push_back (element);

            int childIndex = pq.size()-1;
            while (childIndex >0){
                int parentIndex= (childIndex-1)/2;
                if (pq [childIndex] > pq [parentIndex])
                {
                    int temp = pq [childIndex];
                    pq [childIndex] = pq [parentIndex];
                    pq [parentIndex] = temp;
                }
                else {
                    break;
                }
                childIndex = parentIndex;
            }
        }
```

```
int getMax() {
    if (isEmpty()){
        return 0;
    }
    return pq[0];
}

int removeMax() {
    if (isEmpty()) {
        return 0;
    }

    int ans = pq[0];
    pq[0] = pq [pq.size()-1]
    pq. pop_back();

    int parentIndex =0 ;
    int leftChildIndex = 2* parentIndex +1 ;
    int rightChildIndex = 2* parentIndex + 2;

    while (leftChildIndex < pq.size()) {
        int maxIndex = parentIndex;
        if (pq [leftChildIndex] > pq [maxIndex] ){
            maxIndex= leftChildIndex;
        }
        if ( rightChildIndex < pq.size() && pq[rightChildIndex] >
                                                pq [maxIndex])
        {
            maxIndex = rightChildIndex;
        }
        if (maxIndex == parentIndex) {
            break;
        }
```

```
            int temp= pq [maxIndex];
            pq [maxIndex]= pq [parentIndex];
            pq [parentIndex] = temp;

            parentIndex = maxIndex;
            leftChildIndex = 2* parentIndex +1;
            rightChildIndex= 2* parentIndex +2;
        }
        return ans;
    }

    int getSize(){
        return pq.size();
    }

    bool isEmpty() {
        return pq.size()==0;
    }
};
```

Inplace Heap Sort -

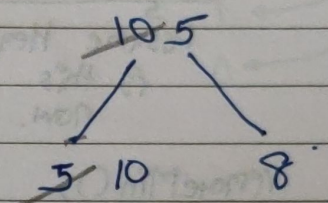
Here our task is to optimized the space complexity of our Heap Sort from O(n) to O(1).

user enter the array a → | 10 | 5 | 8 | 1 | 4 |

In order to optimized we don't need to create a new array, we can use the input array only.

	0	1	2	3	4
a →	10	5	8	1	4.

Heap — unsorted array.



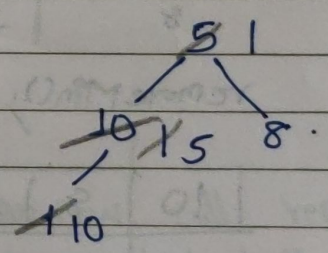
CBT hai but not Heap order property.

	0	1	2	3	4
a →	~~10~~	~~5~~	8	1	4

5 10 — Heap — unsorted array

a →	5	10	8	1 | 4

Heap — unsorted array

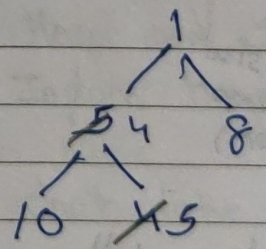


a →	~~5~~	~~10~~	8	~~1~~	4

1 ~~1~~ Heap 10 — unsorted array.
5

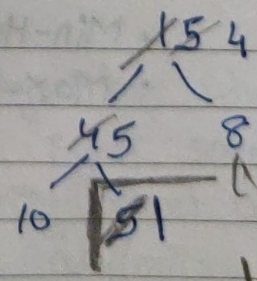
a →	1	~~5~~	8	10	~~4~~

4 Heap 5



Now input array is converted into Heap.

	0	1	2	3	4
a →	~~1~~	~~4~~	8	10	~~5~~
	~~5~~ 4	5			1



We use to swap and pop_back(), but here we want all the elements so we won't pop_back(), we will assume now the Heap is till 3 only

(We don't have to compare 5 and 1 bcos 1 is not in Heap)

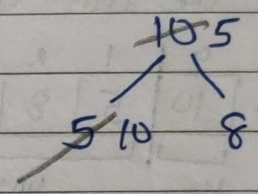
0	1	2	3	4
~~4~~	5	8	~~10~~	1

Assume Heap is this much only

removeMin();

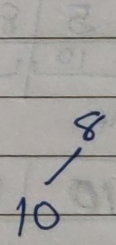
0	1	2	3	4
~~4~~	~~5~~	8	~~10~~	1
~~10~~ 5	10		4.	

Assume Heap is ~~this~~ now.



removeMin();

~~5~~	10	~~8~~	4	1
8		5		



removeMin();

10	8	5	4	1

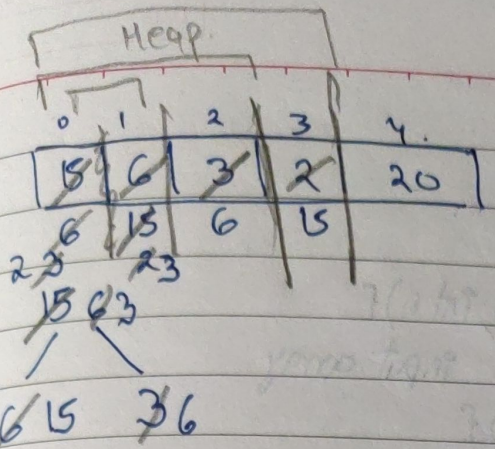
10

10	8	5	4	1

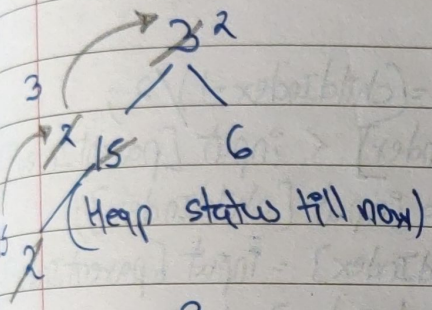
(Heap size is zero now)

→ array status is now this and this is sorted after calling n time removeMin()

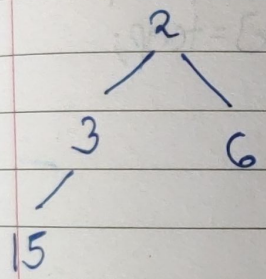
Note :- When we don't take an extra array, then
- Min-Heap will be sorted in descending order.
- Max-Heap will be sorted in increasing order.



i	childIndex	PI
1	i = 1	$\frac{i-1}{2} = 0$
		0
2	2	0
		0
3	3	1
		1
		0



(Heap status till now)



Now, after complete Heap is created, now call removeMin() i.e swap first and last element of array, but don't pop-back() only assume array size is decrease by 1.

- Without taking any extra array for Heap and utilizing the input array for Heap is called Inplace Heap Sort.

Inplace means those algorithm which don't take any extra space besides from input space

Inplace is generally used in sorting algorithms wherein we don't need any extra space and use the input space only. Inplace algorithms have space complexity almost O(1). We use Inplace to improve our space complexity.

Watch Inplace Heap Sort Video Again.

DOMS Page No.
Date / /

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

void inplaceHeapSort (int input[ ], int n){
    // Build the heap in input array
    for (int i=1; i<n; i++){
        int childIndex = i;                    // This is same as
        while (childIndex>0){                  insert element function
            int parentIndex=(childIndex-1)/2;  of Heap (Priority Queue)
            if (input [childIndex] < input [parentIndex]){
                int temp = input [childIndex];
                input [childIndex] = input [parentIndex];
                input [parentIndex] = temp;
            }
            else {
                break;
            }
            childIndex= parentIndex;
        }
    }

    // Remove elements                         // This is same as
    int size =n;                               remove element function
    while (size>1){                            of Heap (Priority Queue)
        int temp = input[0];
        input [0]= input [size-1];
        input[size-1] = temp;

        size --;
        int parentIndex = 0;
        int leftChildIndex= 2*parentIndex +1;
        int rightChildIndex= 2*parentIndex +2;
```

```
while (leftChildIndex < size) {
        int minIndex = parentIndex;
        if (input [minIndex] > input [leftChildIndex]) {
                minIndex = leftChildIndex;
        }
        if (rightChildIndex < size && input [rightChildIndex]
                                        < input [minIndex])
        {
                minIndex = rightChildIndex;
        }
        if (minIndex == parentIndex) {
                break;
        }
        int temp = input [minIndex];
        input [minIndex] = input [parentIndex];
        input [parentIndex] = temp;

        parentIndex = minIndex;
        leftChildIndex = 2* parentIndex +1;
        rightChildIndex = 2* parentIndex +2;
    }
  }
}
int main (){
    int input [] = {5,1,3,0,8};
    inplaceHeapSort (input, 5);
    for (int i=0; i<5; i++)
    {
        cout << input[i] << " ";
    }
    cout << endl;
}
```

Note: Min-Heap will be sorted in descending order in Inplace Heap Sort
Max-Heap will be sorted in ascending order in Inplace Heap Sort

Inbuilt Priority Queue -

In #include <queue> Header file only has Priority Queue implementation. And class name is priority_queue < _ > and it is made with template.
Max-PriorityQueue

Inbuilt Priority Queue names-

isEmpty() → empty()

getSize() → size().

void insert (elem) → void push (elem)

getMin () → T top () → max element milega bcos Max-Priority Queue hai so root node has the maximum value.

T removeMin() → void pop() → delete ho jayega But it's return type is void so kuch return nhi hoga in Inbuilt

Note: Inbuilt Priority Queue is by Default Max-Priority Queue.

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

int main () {
    priority_queue <int> pq;

    pq. push (16);
    pq. push (1);
    pq. push (167);
    pq. push (7);
    pq. push (45);
    pq. push (32);
```

```
cout << "size : " << pq.size() << endl;
cout << "Top : " << pq.top() << endl;

while (! pq.empty()){
        cout << pq.top() << endl;
        pq.pop();
    }
}
```

o/p -

Size : 6

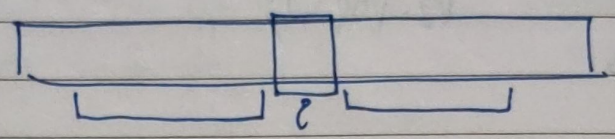
Top : 167 (Bcos our inbuilt PriorityQueue type is Max-PriorityQueue)

167
45
32
16
7
1

K Sorted Array -

k-sorted → a → 10 12 15 6 9 $\quad k=3$

15 12 10 9 6. (Sorting in descending manner)



k-sorted array means an element at ith index can go to left or right only 0 to k-1 places only to it's sorted position. If the above conditions are valid it is a k-sorted array.

$k=3$ a → 10 12 6 15 9 → ✗

15 12 10 9 6.

We have to move 15 3 pos which isn't allowed bcos range to left and right according to k=3 is 0 to k-1 i.e 0 to 2 in this case.

Thus, this is not a k-sorted array.

$k=2$ a → 10 15 6 4 5

15 10 6 5 4. This is a valid k-sorted array.

$k=2$ a → 10 6 15 4 5. This is not a valid k-sorted array.

k=2
(0 to 1)

Note: Sorting can't be done in less than O(nlogn).

a → | 12 | 15 | 7 | 4 | 9 |

k = 3
(Range 0-2)

Sorting in general is O(nlogn).

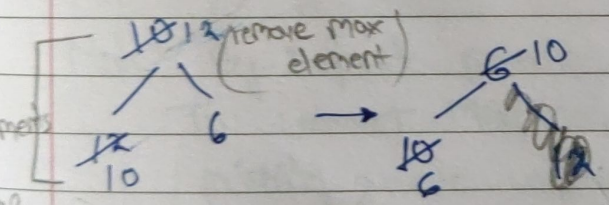
0	1	2	3	4
15	12	9		

indexes: 0,1,2 0,1,2,3 0-4
where max element may be present

Note: Using Max-Priority Queue Solve this.

a → 0 12 6 7 9 k=3

Sorted → 12 _
index 0 1



k elements in Heap always

(Like this we have to do)

n=5 k=3 input →

15	12	9	7	6.
~~12~~	~~15~~	~~6~~	~~7~~	~~9~~.
0	1	2	3	4.

(index of sorted array) j = ~~0~~ ~~1~~ ~~2~~ 3.

i = ~~0~~ ~~1~~ 5.

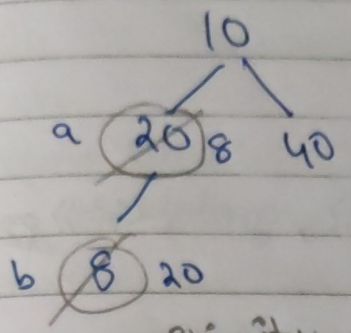
~~15~~, ~~12~~, 6 → pq.
~~12~~, 6, 7, 9

Watch video once again.


```
#include <iostream>
using namespace
#include <queue>

void kSortedArray(int input[], int n, int k){
    priority_queue<int> pq;
    for(int i=0; i<k; i++){
        pq.push(input[i]);
    }                          klogk
}

    int j=0;
    for(int i=k; i<n; i++){
        input[j] = pq.top();
        pq.pop();
        pq.push(input[i]);
        j++;                   (n-k) logk
    }

    while(!pq.empty()){
        input[j] = pq.top();
        pq.pop();
        j++;                   → klogk
    }
}
int main(){
    int input[] = {10, 12, 6, 7, 9};
    int k=3;
    kSortedArray(input, 5, k);
    for(int i=0; i<5; i++){
        cout << input[i] << " ";
    }
}
```

Total time complexity -
$k\log k + n\log k$ ~~$+k\log k + n\log k$~~
$(n+k)\log k$
$[n >>> k]$
$n \log k$
(k is very small log k is smaller)
n
$O(n)$ is time complexity.

Inbuilt Min Priority Queue -



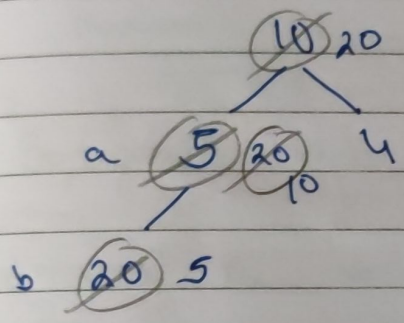
→ Min - Heap

a > b

to do this ← (parent > child) then perform swap to create Min-Heap.

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

(do include this in inbuilt priority_queue)



→ Max - Heap

a < b

By-Default this happens. ← (parent < child) then perform swap to creat Max-Heap.

Note: By-Default priority_queue is Max-Heap one.
To convert Max-Heap into Min-Heap in inbuilt priority_queue only do this -

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

Only include this and Max-Heap will change to Min-Heap now.

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

int main(){
    priority_queue <int, vector<int>, greater<int>> pq;
    pq.push(100);
    pq.push(21);
    pq.push(7);
    pq.push(165);
    pq.push(78);
    pq.push(4);

    cout << pq.size()<<endl;
    cout << pq.empty()<<endl;
    cout << pq.top()<<endl;

    while(!pq.empty()){
        cout << pq.top()<<endl;
        pq.pop();
    }
}
```

This is a min-priority queue (inbuilt)

Output -

6 → size()
0 → empty()
4 → top()
4
7
21
78
100
165

min-priority queue printed