Priority Queue

elements

a_1	a_2	a_3	a_4	a_5	a_6
p_1	p_2	p_3	i_4	15	10

importance factor

Queue → Priority queue

Queue → FIFO

Priority queue → FIFO → priority → Importance factor.

Priority queue

- Min-priority (minimum priority element)
- Max priority queue (maximum priority elem)

→ popped

- Insert
- getMax / getMin
- removemax / removemin

→ top() → pop() } stack

V-II

Problem →

n → insert ?

get min/max ? → priority based.

⇓

element = priority

(MIN priority queue)

14, 9, 8, 7, 1, 2, 20

1 get min → 1
2 get min → 2
3 insert → 20
4 get min → 3

	insert	get min/max	remove min/max
Arrays			
-(unsorted)	O(1)	O(n)	O(n)
-(sorted)	O(n)	O(1)	O(n) = shifting.
Linked List			
-(unsorted)	O(1)	O(n)	O(n)
-(sorted)	O(n)	O(1)	O(1)
BST	O(h) h=height	O(h)	O(h) {worst case h=n}
Best			
Balanced BST	O(log n)	O(log n)	O(log n)
Hash-map	O(1)	O(n)	O(n)

Heap

→ Issues with Balanced BST

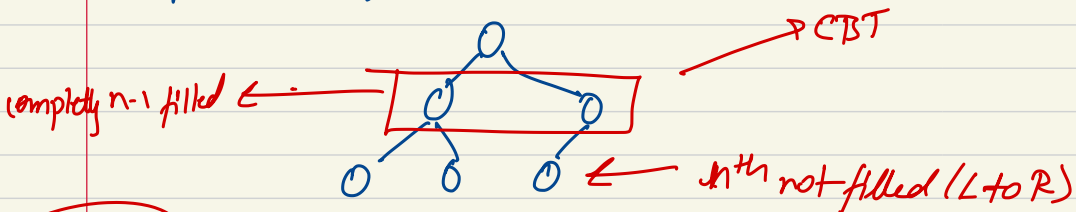
- Balancing
- Complicated storing

↓

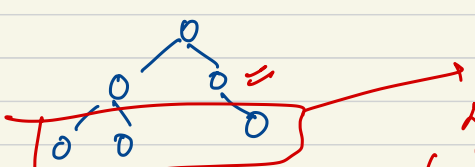
Heaps

- A complete Binary Tree
- Heap order property

→ Complete Binary Tree (CBT)



nth → Right



Heap property
① → CBT → Height → overlooking → balancing of BST

↳ minimum no. of nodes with height h in CBT.

h=4 → 2^0, 2^1, 2^2, 1 } Minimum

total = $2^0 + 2^1 + 2^2 + 1^{h-2} + 1 = 2^{h-1}$

↳ maximum " " " " " = ?

h=4 → 2^0, 2^1, 2^2, 2^3 } h=4.

$\frac{a_1(r^{n+1}-1)}{r-1}$

$2^0 + 2^1 + 2^2 + 2^3 + \cdots 2^{h-1}$

$1 + 2 + 2^2 + 2^3 + \cdots 2^h$

$= 2^{h-1}$

CBT → n

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

$2^{h-1} < n$ ⓐ

$n \leq 2^h - 1$ ⓑ

for (a)

$$2^{h-1} \le n \Rightarrow \log(2^{h-1}) \le \log n$$

$$\Downarrow$$

$$(h-1)\log 2 \le \log n$$

$$h-1 \le \log_2 n$$

$$h-1 \le \log_2 n$$

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

for (b)

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

$$n+1 \le 2^h$$

$$\log(n+1) \le h \log 2$$

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

$$\log_2(n+1) \le h \le \log_2 n + 1$$

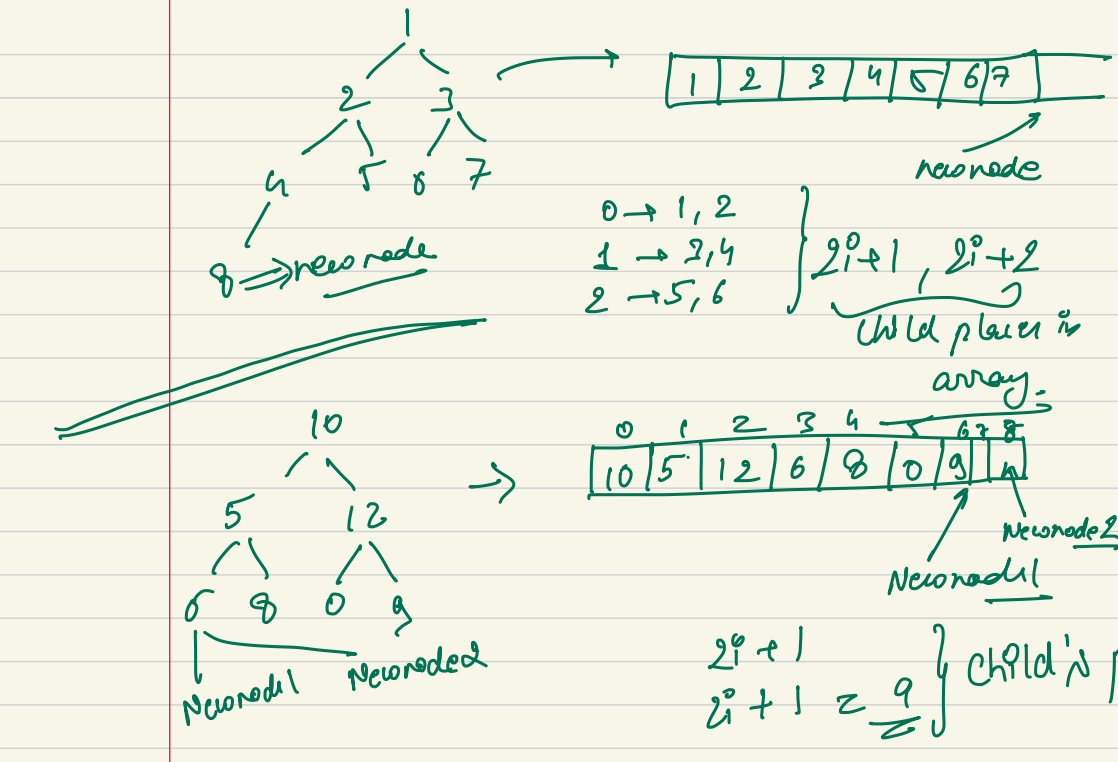
$$O(\log_2 n) \le h \le O(\log_2 n)$$

$$\Downarrow$$

$O(\log_2 n)$ → Best structure for priority queue

→ Storing of BST →

- level order traversal, first empty place is to be picked for inserting the value in BST, but it is O(n), hence CRT doesn't use this



parent → child

$i \longrightarrow 2i+1,\ 2i+2$

child → parent

$i \longrightarrow \frac{i-1}{2}$

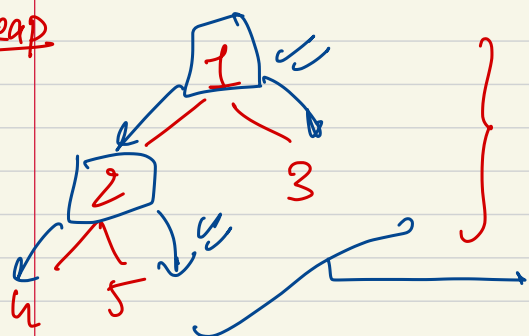
Priority Queue
- Min
- Max

Heap property

③ → Heap order property →

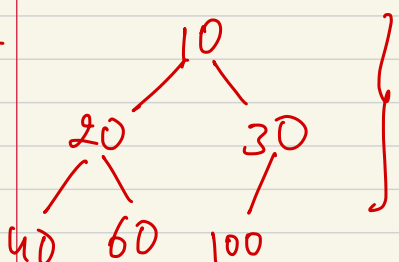
- Min-heap
- Max-heap

Min heap



Root must have value less than both of his child

Valid heap (min) ✓

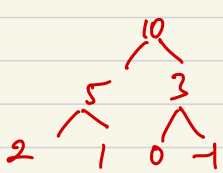


Min-max heap order property validated

↳ min-heap ✓

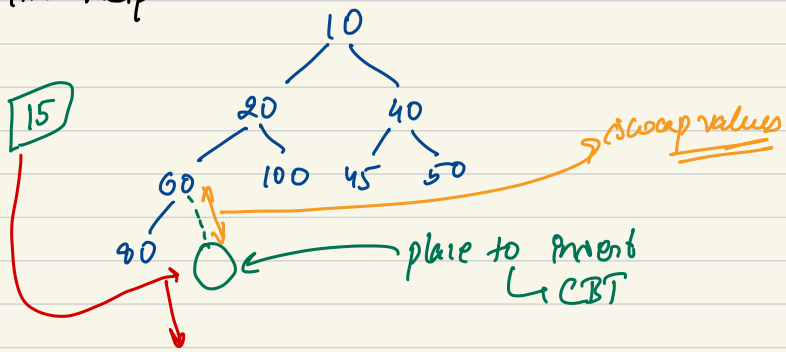
Max heap →

All parents have value greater than both childs

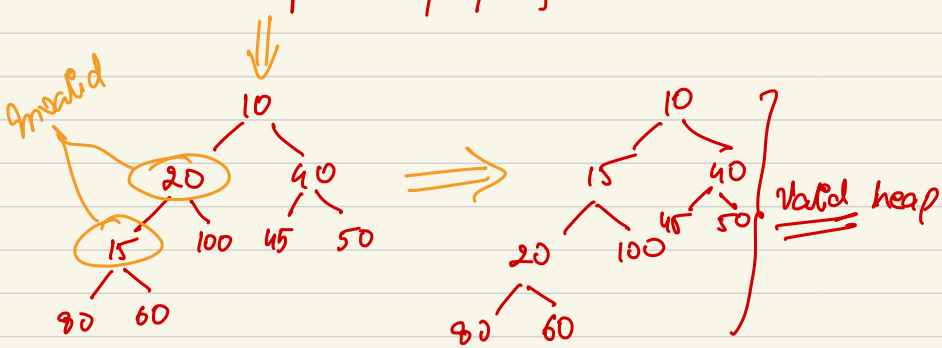


Heap Insert →

- Min heap →



Invalid heap order property



O(h) → worst case insert function

O(log n) → CBT

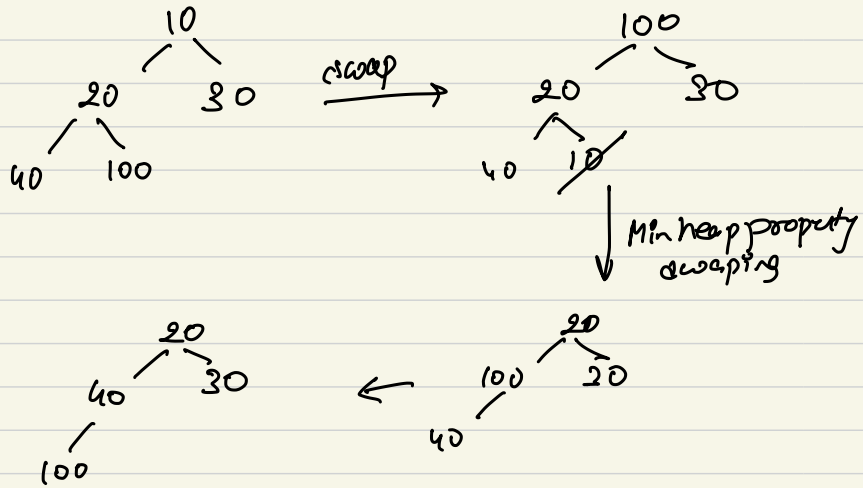
[Up-heapify]

Heap delete →

Min heap

→ swap if needed
→ Delete the desired item
maintaining the properties.

Delete(10)



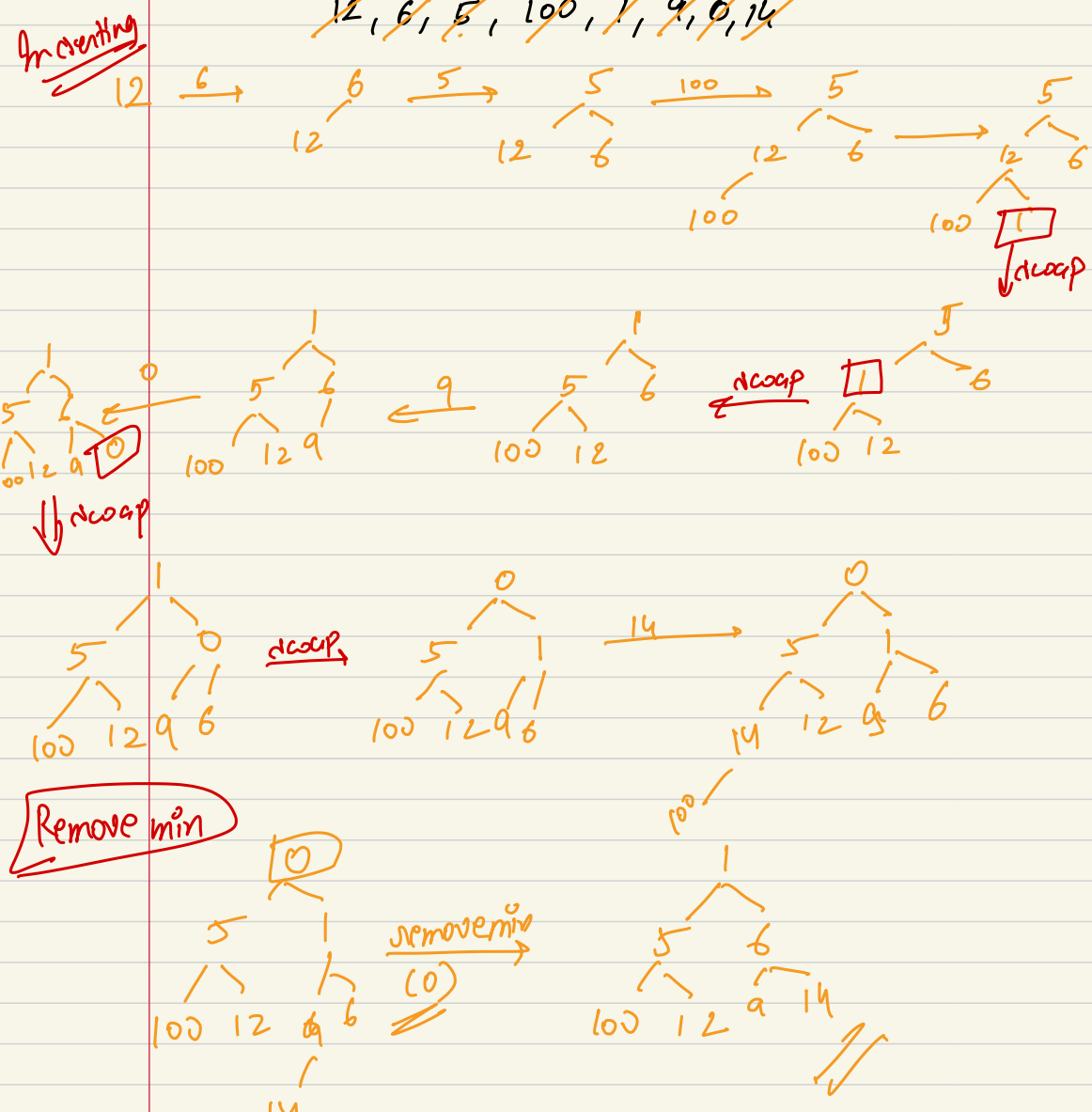
Min heap → minimum value at root.

get() → return root value [O(1)]

dellete() → desired properties must
stand ≡[O(log₂n)]
≡[O(n)]

→ Draw Min heap & call remove_min() twice?

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



```
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];
```

```
  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;
    }
    childIndex = parentIndex;
  }

}
```

```
// down-heapify

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

while(leftChildIndex < pq.size()) {
  int minIndex = parentIndex;
  if(pq[minIndex] > pq[leftChildIndex]) {
    minIndex = leftChildIndex;
  }
  if(rightChildIndx < pq.size() && pq[rightChildIndx] < pq[minIndex]) {
    minIndex = rightChildIndx;
  }
  if(minIndex == parentIndex) {
    break;
  }
  int temp = pq[minIndex];
  pq[minIndex] = pq[parentIndex];
  pq[parentIndex] = temp;

  parentIndex = minIndex;
  leftChildIndex = 2 * parentIndex + 1;
  rightChildIndx = 2 * parentIndex + 2;
}

return ans;
```

Heap sorting

Time complexity → O(n log n + log n)
O(n log n)

(inserting, remove/swaping)

Space complexity → O(n)

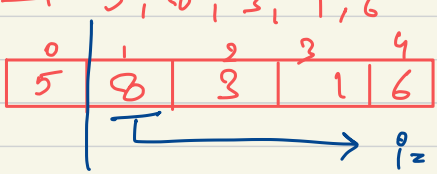
Inplace heap sort
↓
O(1)

→ Inplace Heap Sort →

No new space to be used. O(1) space complexity at average case.

Max heap

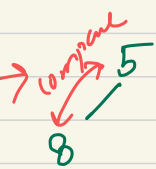
a → 5, 8, 3, 1, 6



heap

unsorted array

child index = i
parent index = (i-1)/2



till reach the end

```
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;
    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 elements
```

14.Inplace heap sort Solution.m4v

```
int size = n;

while(size > 1) {
  int temp = pq[0];
  pq[0] = pq[size - 1];
  pq[size-1] = temp;

  size--;

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

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

    parentIndex = minIndex;
    leftChildIndex = 2 * parentIndex + 1;
    rightChildIndx = 2 * parentIndex + 2;
  }
```

Inbuilt Priority queue → STL → C++.

#include <queue>

priority_queue< > → template

- isEmpty → empty()
- getSize → size()
- void insert(e) → push(e)
- getMin() → T top()
  - max element
- T removeMin() → void pop()
  - Delete root (max elem

Max priority queue by default

```
#include <queue>

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();
  }
}
~
~
```

Nearly sorted

Medium Accuracy: 75.25% Submissions: 25K+ Points: 4

Given an array of **n** elements, where each element is at most **k** away from its target position, you need to sort the array optimally.

Example 1:

Input:
n = 7, k = 3
arr[] = {6,5,3,2,8,10,9}
Output: 2 3 5 6 8 9 10
Explanation: The sorted array will be
2 3 5 6 8 9 10

```
class Solution
{
    public:
    //Function to return the sorted array.
    vector <int> nearlySorted(int arr[], int num, int K){
        // Your code here
        vector<int> ans;
        priority_queue<int,vector<int>, greater<int>> pq;

        for(int i=0;i<K+1;i++)
        {
            pq.push(arr[i]);
        }
        //cout<<pq.top();
        for(int i=K+1;i<num;i++)
        {
            ans.push_back(pq.top());
            pq.pop();
            pq.push(arr[i]);

        }
        while(!pq.empty())
        {
            ans.push_back(pq.top());
            pq.pop();
        }
        //sort(ans.begin(),ans.end());
        return ans;
    }
```

Kth smallest element

Medium Accuracy: 35.17% Submissions: 414K+ Points: 4

Given an array **arr[]** and an integer **K** where K is smaller than size of array, the task is to find the **K^{th} smallest** element in the given array. It is given that all array elements are distinct.

Example 1:

Input:
N = 6
arr[] = 7 10 4 3 20 15
K = 3
Output : 7

```
class Solution{
    public:
    // arr : given array
    // l : starting index of the array i.e 0
    // r : ending index of the array i.e size-1
    // k : find kth smallest element and return using this function
    int kthSmallest(int arr[], int l, int r, int k) {
        //code here
        priority_queue<int> pq;
        for(int i=0;i<k;i++)
        {
            pq.push(arr[i]);
        }
        for(int i=k;i<r+1;i++)
        {
            if(pq.top()>arr[i])
            {
                pq.pop();
                pq.push(arr[i]);
            }

        }
        return pq.top();
```

K largest elements

Basic Accuracy: 61.15% Submissions: 45K+ Points: 1

Unable to Crack Interviews of Your Dream Companies ? Click Here to End This Problem!

Given an array of N positive integers, print k largest elements from the array.

Example 1:

Input:
N = 5, k = 2
arr[] = {12,5,787,1,23}
Output: 787 23
Explanation: First largest element in the array is 787 and the second largest is 23.

```
class Solution
{
    public:
    //Function to return k largest elements from an array.
    vector<int> kLargest(int arr[], int n, int k)
    {
        // code here
        priority_queue<int,vector<int>,greater<int>> pq;
        for(int i=0;i<k;i++)
        pq.push(arr[i]);

        for(int i=k;i<n;i++)
        {
            if(pq.top()<arr[i])
            {
                pq.pop();
                pq.push(arr[i]);
            }
        }
        vector<int> ans;
        while(!pq.empty())
        {
            ans.push_back(pq.top());
            pq.pop();
        }
        sort(ans.begin(),ans.end(),greater<int>());
        return ans;
    }
};
```