**Heap**

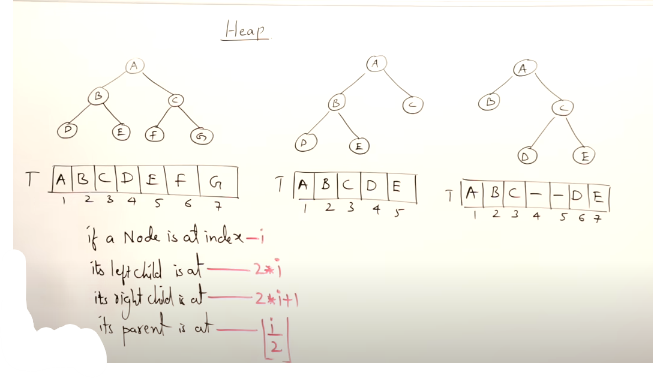
|  |  |  |  |
| --- | --- | --- | --- |
| **Easy Interview Questions on Heap Data Structure** | | | |
| 1. Kth smallest element in an array |  | 1. Height of a complete binary tree (or Heap) with N nodes |  |
| 1. Minimum product of k integers in an array of positive Integers |  | 1. Minimum sum of two numbers formed from digits of an array |  |
| 1. Sort an Almost Sorted Array |  | 1. Kth smallest element in a row-wise and column-wise sorted 2D array |  |
| 1. Top K Frequent Elements |  | 1. Sum of all elements between k1’th and k2’th smallest elements |  |
| **Medium Interview Questions on Heap Data Structure** | | | |
| 1. Implement a Min Heap |  | 1. Merge two binary max heaps |  |
| 1. Implement a Max Heap |  | 1. Find k closest numbers |  |
| 1. Heap Sort |  | 1. Sort an almost sorted array |  |
| 1. Convert max heap to min heap |  | 1. K maximum sum combinations from two arrays |  |
| 1. Convert min Heap to max Heap |  | 1. BST to max heap |  |
| 1. Check if a Binary Tree is a Min Heap |  | 1. Convert BST to Min Heap |  |
| 1. Check if a Binary Tree is a Max Heap |  | 1. K’th largest element in a stream |  |
| 1. Binary Heap |  | 1. Find k numbers with most occurrences in the given array |  |
| 1. Given level order traversal of a Binary |  | 1. Find the kth largest element in an array |  |
| 1. Tree, check if the Tree is a Min-Heap |  | 1. Merge overlapping intervals |  |
| 1. Implement a priority queue |  | 1. Game with String |  |
| 1. Heap Sort for decreasing order using min heap |  | 1. Maximize The Array |  |
| 1. Find kth smallest element in a row-column sorted matrix |  | 1. Rearrange characters |  |
| 1. Largest triplet product in a stream |  | 1. Minimum sum of squares of character counts in a given string after removing k characters |  |
| 1. Connect n ropes with minimum cost |  | 1. Maximum sum of at most two non-overlapping intervals in a list of Intervals |  |
| 1. Merge two binary max heaps |  | 1. K-th Largest Sum Contiguous Subarray |  |
| **Hard Interview Questions on Heap Data Structure** | | | |
| 1. Merge k sorted arrays |  | 1. Minimum cost to connect all cities |  |
| 1. Merge k Sorted Lists |  | 1. Single-Source Shortest Paths – Dijkstra’s Algorithm |  |
| 1. Find the median of a stream of running integers |  | 1. Sliding Window Maximum (Maximum of all subarrays of size K) |  |
| 1. Smallest range in K lists |  | 1. K maximum sum combinations from two arrays |  |
| 1. Huffman Encoding |  | 1. Merge two sorted arrays in O(1) extra space using Heap |  |

|  |
| --- |
| Link: <https://www.geeksforgeeks.org/top-50-problems-on-heap-data-structure-asked-in-interviews/> |

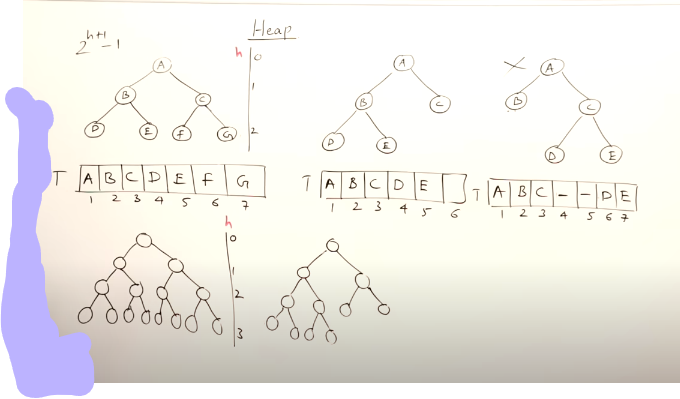
|  |
| --- |
| Subtopic of head   1. **Array representation of Binary tree** 2. **Complete Binary tree** 3. **Heap** 4. **Insert and Delete** 5. **Heap sort** 6. **Heapify** 7. **Priority Queue** |

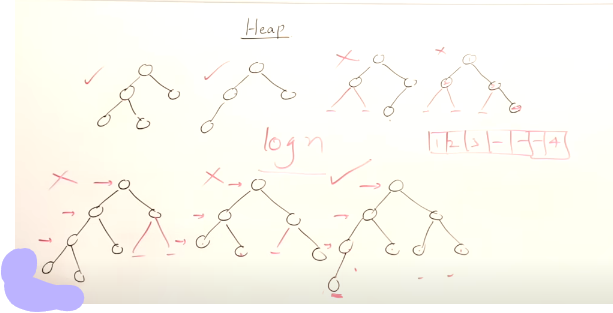
<https://youtu.be/HqPJF2L5h9U?si=VZBD70jKGY-_P12T>

Representation of binary tree using array



1. Full Binary tree and Full complete binary tree
2. Full binary tree can have maximum nodes
3. 1st to last element of array for binary tree if there is **no missing** element in between then it’s a **complete binary tree**
4. 1st to last element of array for binary tree if there is a **single missing** element in between nodes then it’s a **binary tree**
5. Every **full binary tree** is a **complete binary tree**
6. **Complete binary tree** element is filled from **left to right**
7. Height of a **complete binary tree** will always be (minimum)



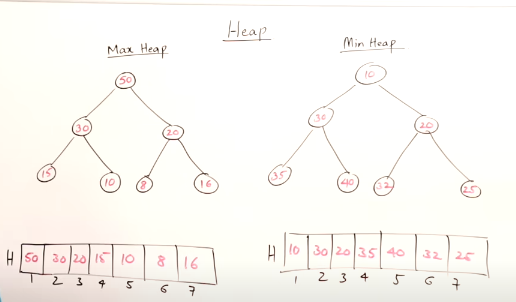


**Heap**

A **Heap** is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is less than or equal to its own value.

Heap is a complete binary tree heap types :

1. Max heap 🡪 duplicate elements are allowed (root is bigger than all of its decedents)
2. Min heap 🡪 duplicate elements are allowed (root is smaller than all of its decedents)

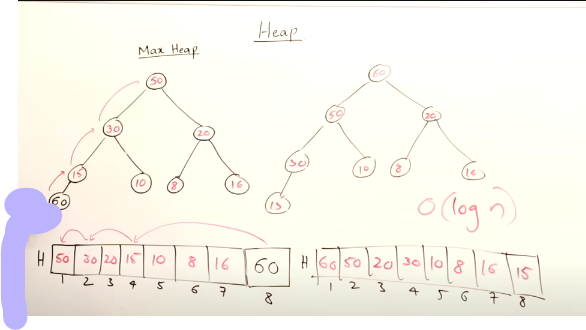


**Insert operation in max heap**

* Insertion is not performed at roots
* Set the new element as leave then compare
* Then new node will move forward until it reaches its place
* Time complexity of a complete binary tree **minimum O(1)** and **maximum** (time taken for insertion)
* Swapes are depend on height
* Elements moves **leaves towards roots** so the directions of adjustment is **upwards**
* New compare with roots
* Adjust upwards leaves towards roots

**Steps:**

1. Add New element as leaves in the last element of the array
2. Compare with its parent (ancestor) and if is bigger then replace the value



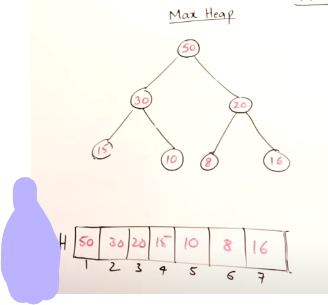
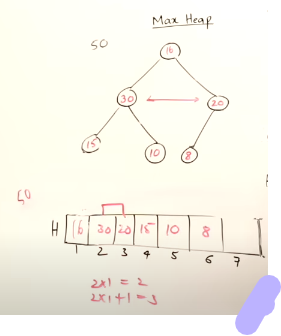
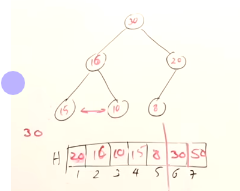
**Delete operation on max heap**

* Top heap = max heap
* Deletion depends on height
* Max time:
* Max head – deletion – you get next --largest element
* Min head – deletion – you get next --smallest element
* It gets sorted

**Steps:**

Hypothesis 🡪 delete the root (highest/ smallest)

1. Delete root
2. The last element will get the place the root has be abandon
3. Adjust towards downwards (root -> leaves ) (new root get compared with its children)
4. Store the deleted element at the last place of array where that replaceable element came form (its not part of heap its out side)

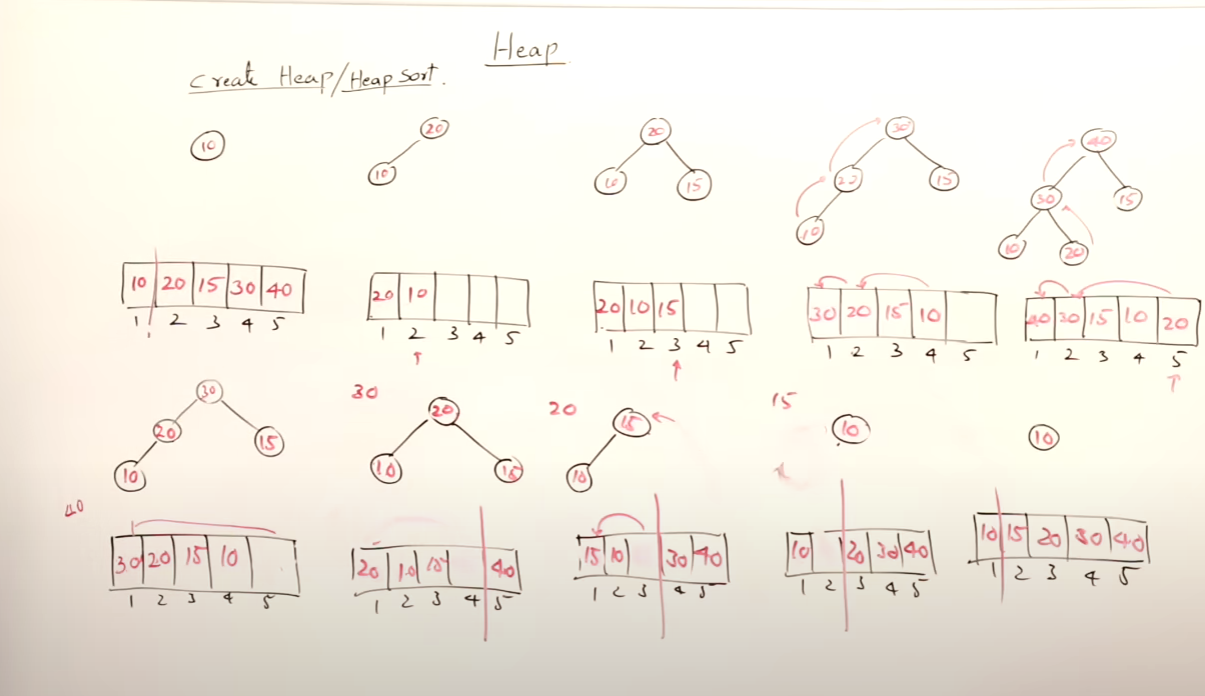
  

**Heap sort**

It has two steps

1. For given set of elements create a heap by inserting all the element one by one
2. One the heap is formed delete all the element from heap one by one the element will get sorted

Time complexity for insertion: **nlogn**



2nd step: step:

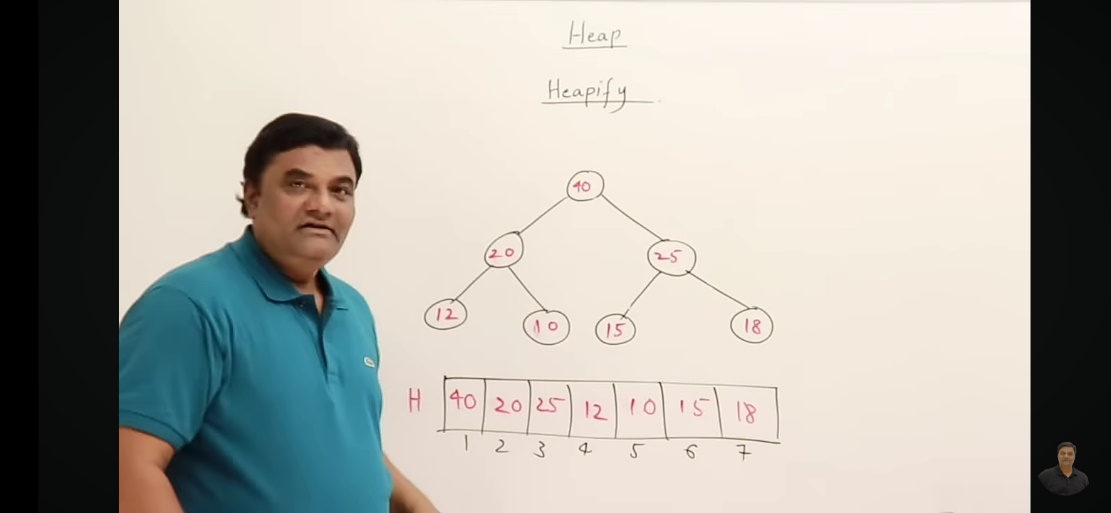
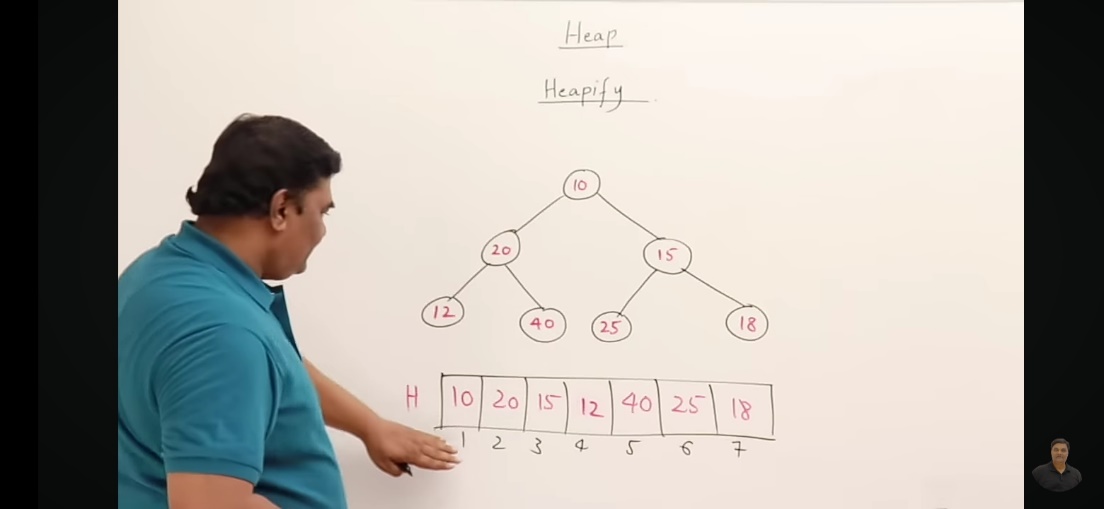
1st step:

Time complexity for deletion: **nlogn**

Total time : **O(nlogn)**

**Heapify**

* **The process in which the binary tree is reshaped into a Heap data structure**



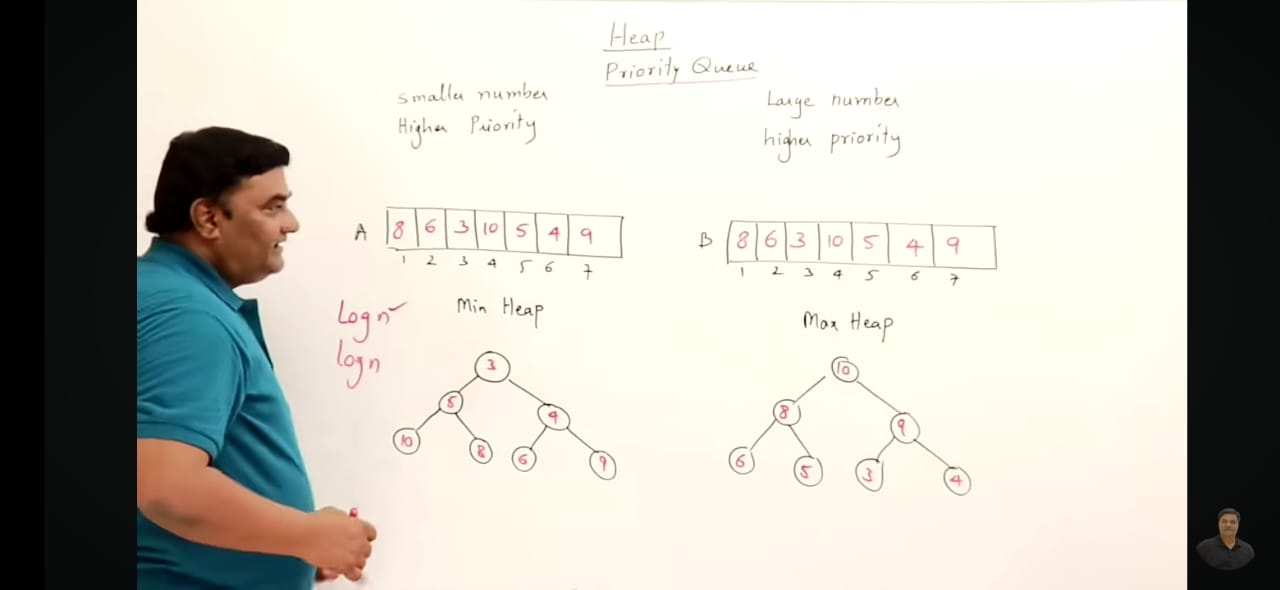
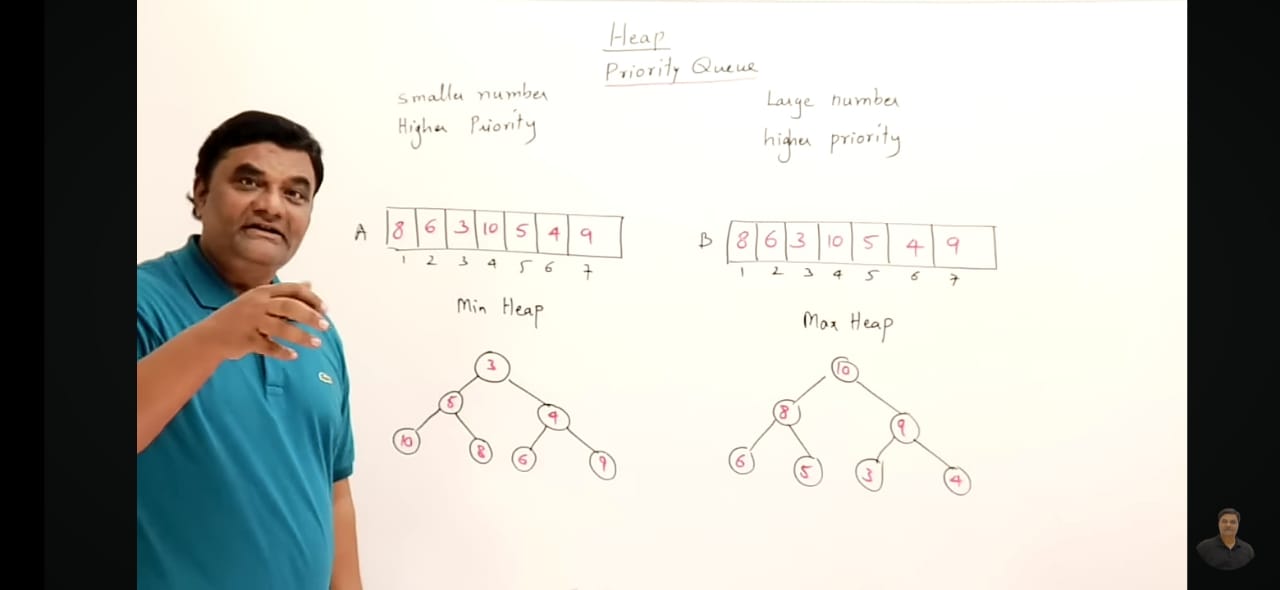
* It is a procedure of creating a heap
* Isn’t not max heap but complete binary tree
* We want max heap
* Time complexity for hrapify : O(n) 🡪 faster (minimum time taken to create a heap)
* Creating a heap time complexity: O(nlogn)

Steps:

1. If one element is alone then it’s a heap go forward
2. One element is being compare to is children in order to find max value
3. Adjusted the elements downwards and we started from last element of the array
4. We scan the array from right to left (it has the same procedure as deletion just the direction is different)

**Priority queue**

A priority queue is **a type of queue that arranges elements based on their priority values**



1. It does not follow FIFO principle
2. The element has priority and insertion and deletion are performed based on priority
3. There are two method that are available in this
   * + **Smaller number priority** 🡪 **create min heap**
     + **Larger number priority** 🡪 **create max heap**
4. From normal array ----- in between **insertion and deletion** one of them is faster and one of them is slower
5. From heap ----- in between **insertion and deletion** both has same time complexity **logn** (best data structure for priority queue)

<https://www.geeksforgeeks.org/priority-queue-set-1-introduction/>

A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

Priority queue implementation:

1. **Array**
2. **Link list**
3. **Heap data structure**
4. **Binary search tree**

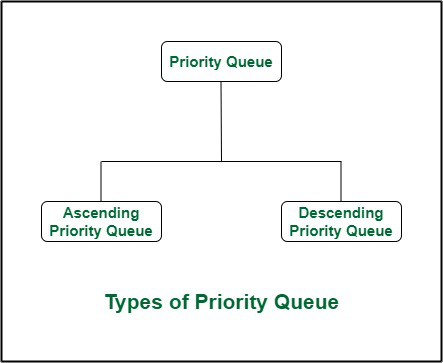
Priority queues are often used in real-time systems, where the order in which elements are processed can have significant consequences. **They are also used in algorithms to improve their efficiencies, such as Dijkstra’s algorithm for finding the shortest path in a graph and the A\* search algorithm for pathfinding.**

Operations in priority queue

1. Insertion
2. Deletion
3. Peek

Types of priority queue:

1. Ascending order ( the element with a lower priority value is given a higher priority in the priority list. )
2. Descending order ( the element with a higher priority value is given a higher priority in the priority list.)



**Learned from theory class**

Binary\_heap() 🡪 bacically making a complete binary tree representation

Algorithom:

**max\_heapify() or min\_heapify()**

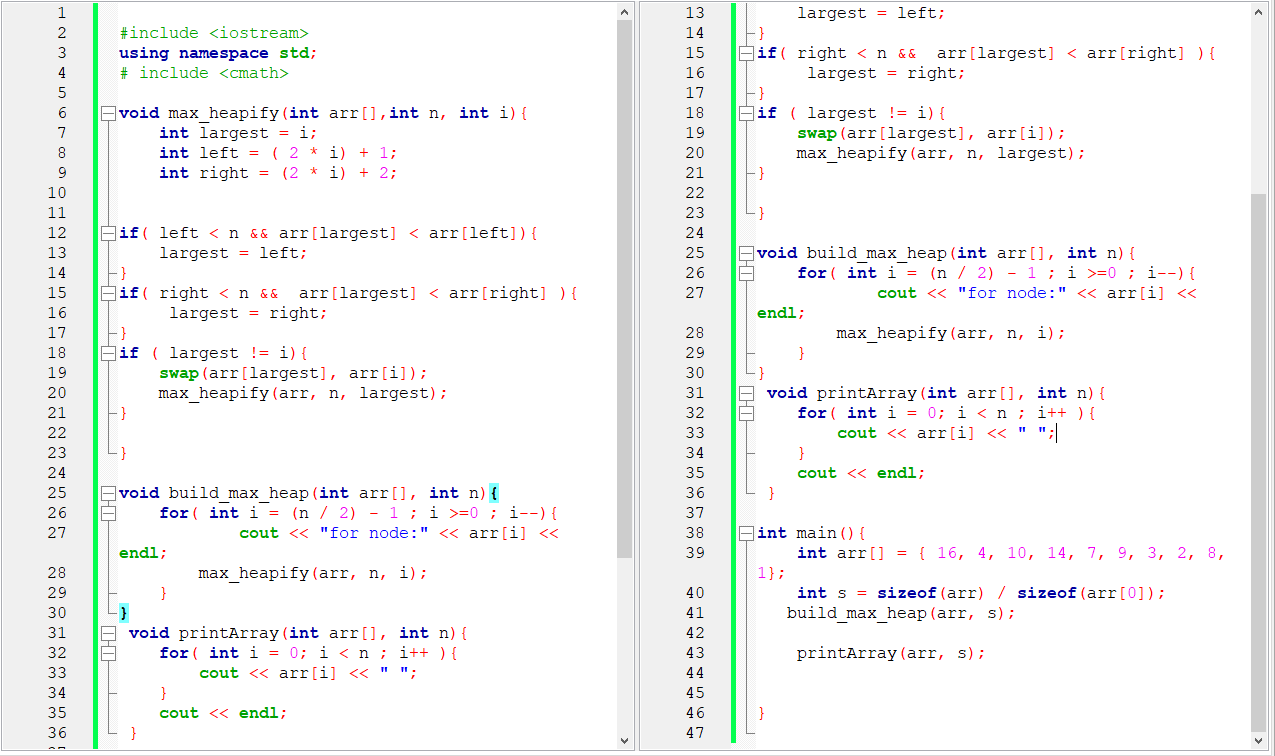
Binary Heap is 2 type

1. Max\_heaps ( the root has the largest value )
2. Min\_heaps ( the root has the smallest value )

**Max\_Heapfiy() algorithm:**

Max\_Heapify() 🡪 transform a normal complete binary tree to max heap complete binary tree along with maintain max\_heap property

It also include built\_max\_heap ()



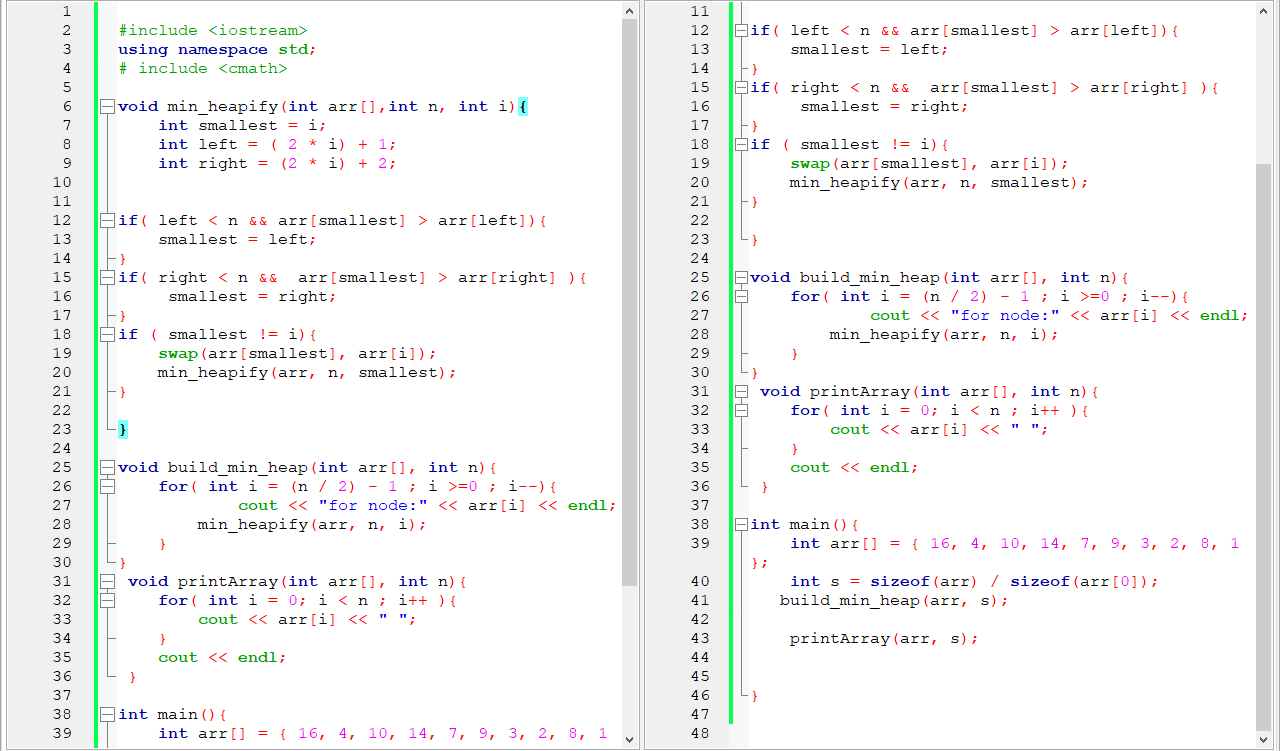
Output:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

**Min\_Heapfiy() algorithm:**

Min\_Heapify() 🡪 transform a normal complete binary tree to min heap complete binary tree along with maintain min\_heap property

It also include built\_min\_heap ()



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 14 | 30 | 18 | 37 | 49 | 33 | 24 | 22 | 56 |

Output:

Analyzing heapify

* If the heap at i has n elements, at most how many elements can the subtrees at l or r have?

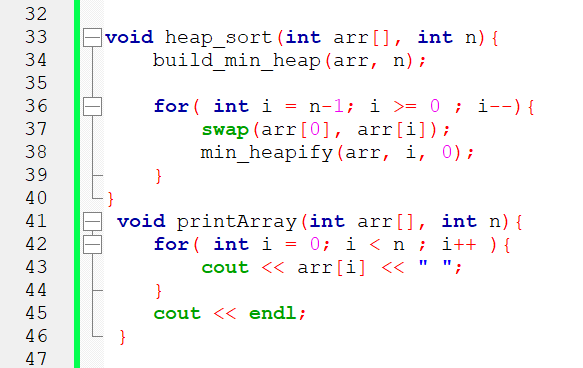
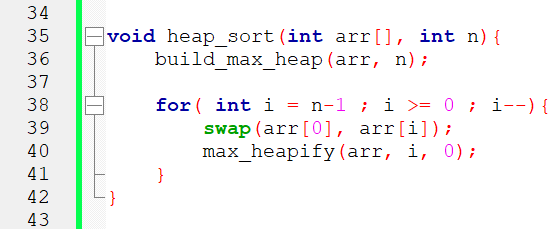
Answer: 2n/3 (worst case: bottom row half full)

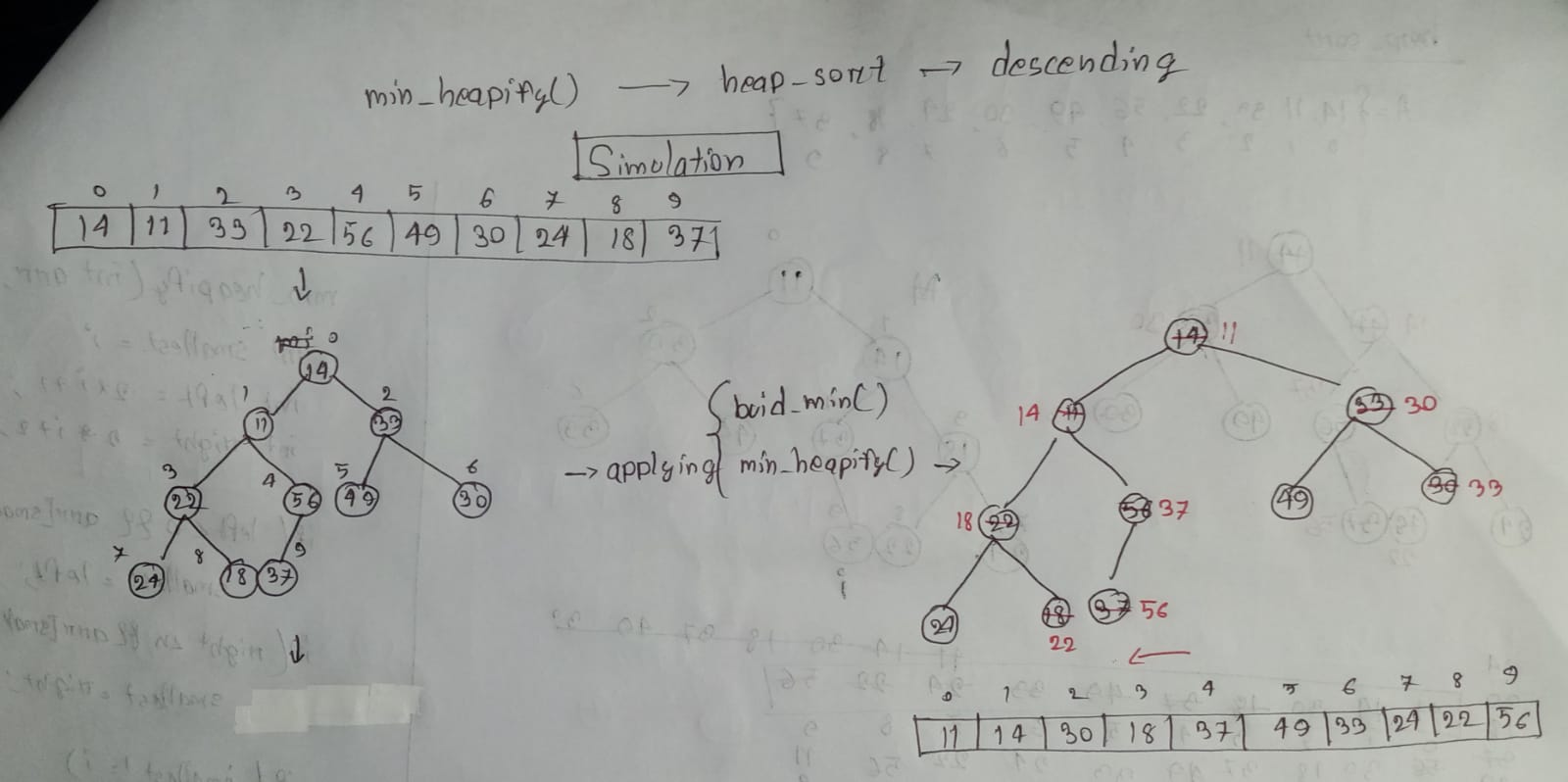
**Heap Sort**

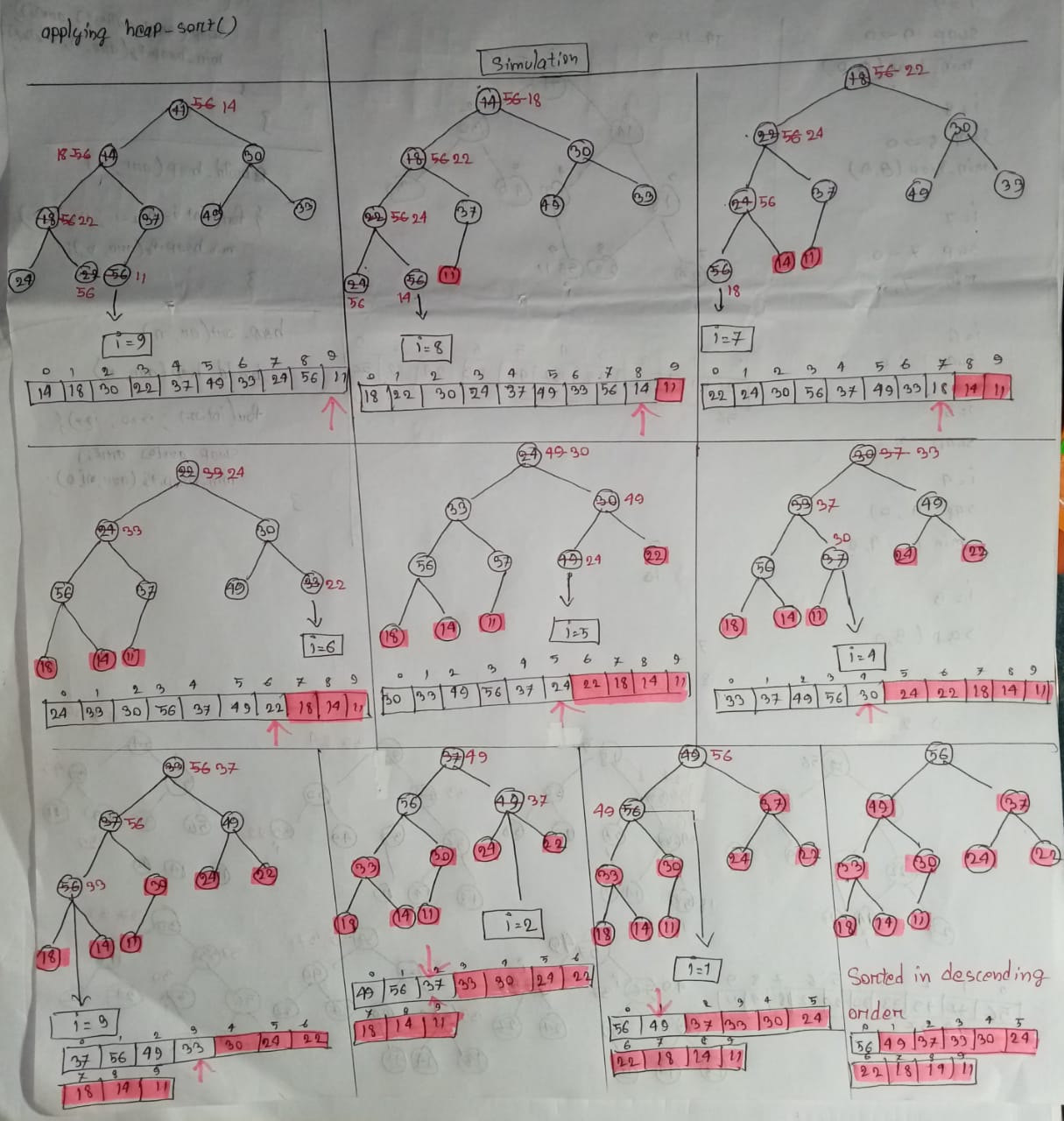
Heap\_Sort()

Using max\_heapify() 🡪 max\_heapify() + build\_max\_heap() + heap\_sort() + print\_array() 🡪 ascending order

Using min\_heapify() 🡪 min\_heapify() + build\_min\_heap() + heap\_sort() + print\_array() 🡪 descending order







**Priority Queue**

Operations of priority queue

1. Increase\_key( S, x, k ) 🡪 Increases the value of element *x*’s key to the new value *k*
2. Insert( S, x ) 🡪 Inserts element *x* into set *S*, according to its priority
3. ExractMax( S ) 🡪 Removes and returns the element of *S* with the largest key
4. Maximum( S ) 🡪 Returns, but does not remove, the element of *S* with the largest key

Pseudocode for the functions

|  |  |  |  |
| --- | --- | --- | --- |
| Increase\_key( S, x, k ) | Insert( S, x ) | ExractMax( S ) | Maximum( S ) |
| **1**    **Prior need**  Swap(int a, int b)  Parent(int i) | **2**    **Prior need**  Swap(int a, int b)  Parent(int i)  Increase\_key(array, size, key) | **3**    **Prior need**  Swap(int a, int b)  Max\_heapify(array, size, i) | **4** |

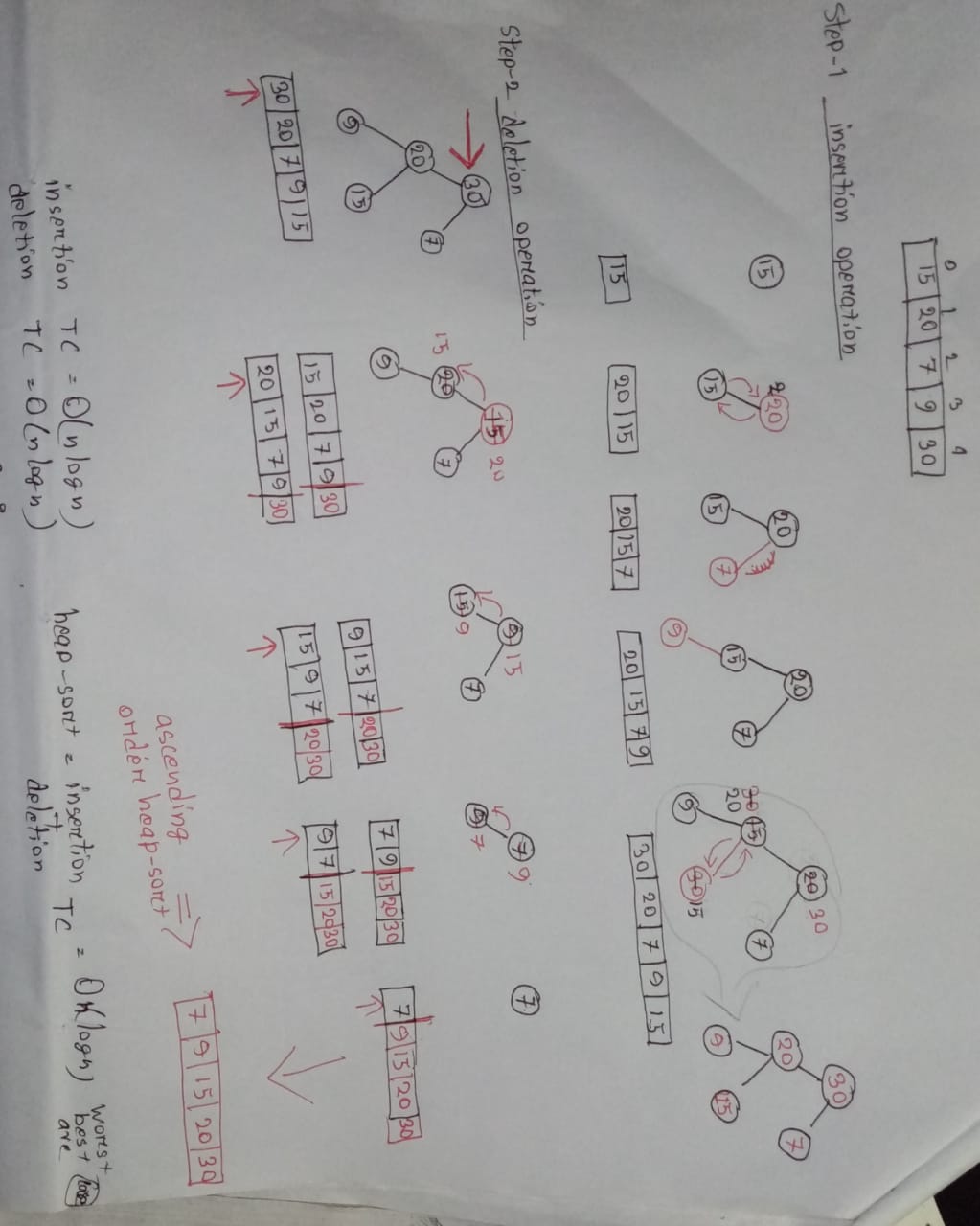
|  |
| --- |
| **Max\_increase\_key(int array[], int i, int key)** |
|  |
| **Max\_heap\_insert(int arr[], int size, int key)** |
|  |
| **Exract\_max(int arr[], int &n)** |
|  |
| **Maximum(int arr[])** |
|  |

**7.9 Heap Sort | Heapify Method | Build Max Heap Algorithm | Sorting Algorithms**

<https://youtu.be/Q_eia3jC9Ts?si=8WSsuY_ZyoGYz4yW>

After every insertion I have to check if it is still satisfying max heap property or not

After every deletion I have to check if it is still satisfying max heap property or not



Heap sort has two approach

1. Insertion + deletion
2. Build heap + Heapify + heap sort

