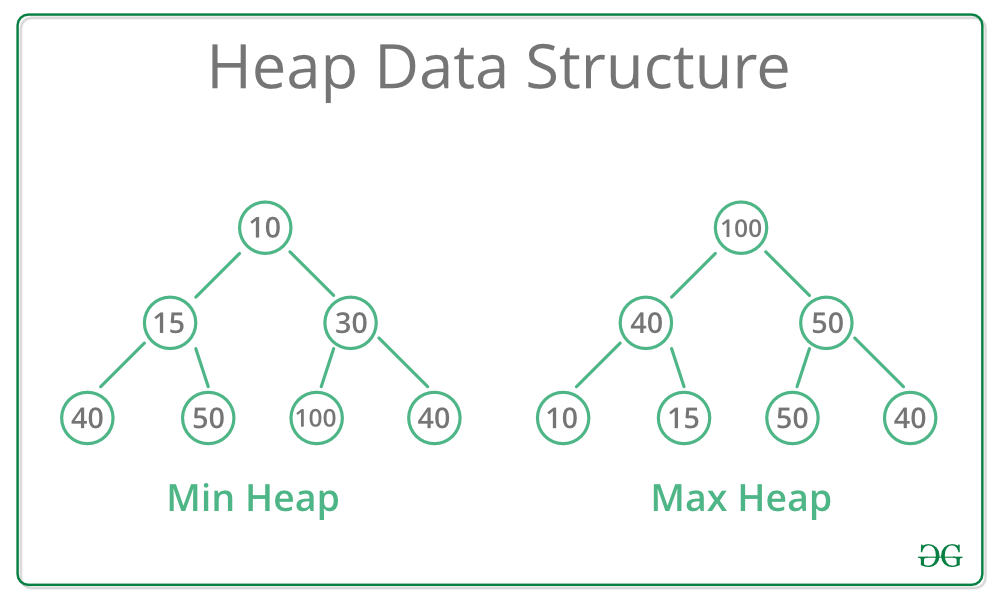
**Topic 1 : Introduction to Heaps**

A Heap is a Tree-based data structure, which satisfies the below properties:

1. A Heap is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

1. A Heap is either Min Heap or Max Heap. In a Min-Heap, the key at root must be minimum among all keys present in the Binary Heap. The same property must be recursively true for all nodes in the Tree. Max Heap is similar to MinHeap.

**Binary Heap**: A Binary Heap is a heap where each node can have at most two children. In other words, a Binary Heap is a complete Binary Tree satisfying the above-mentioned properties.  
  


**Representing Binary Heaps**

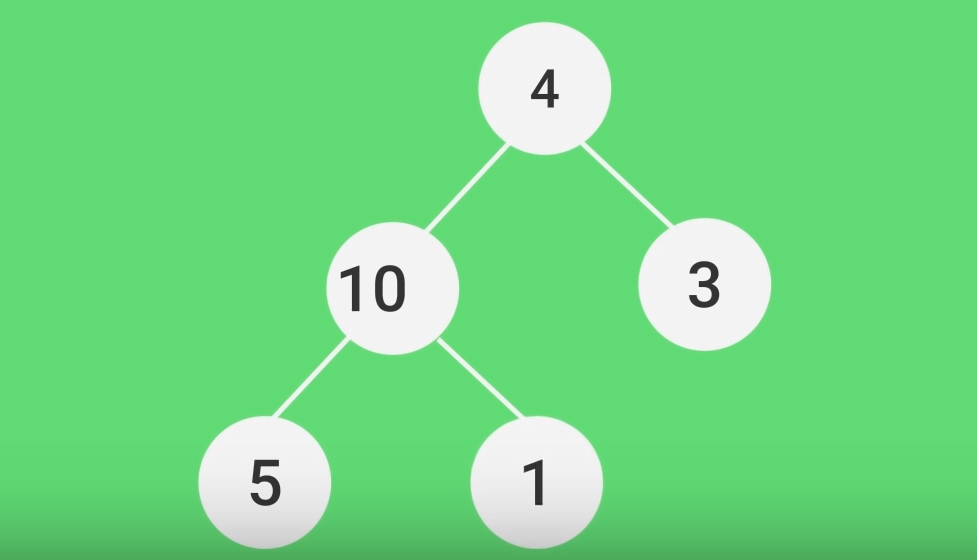
Since a Binary Heap is a complete Binary Tree, it can be easily represented using Arrays.

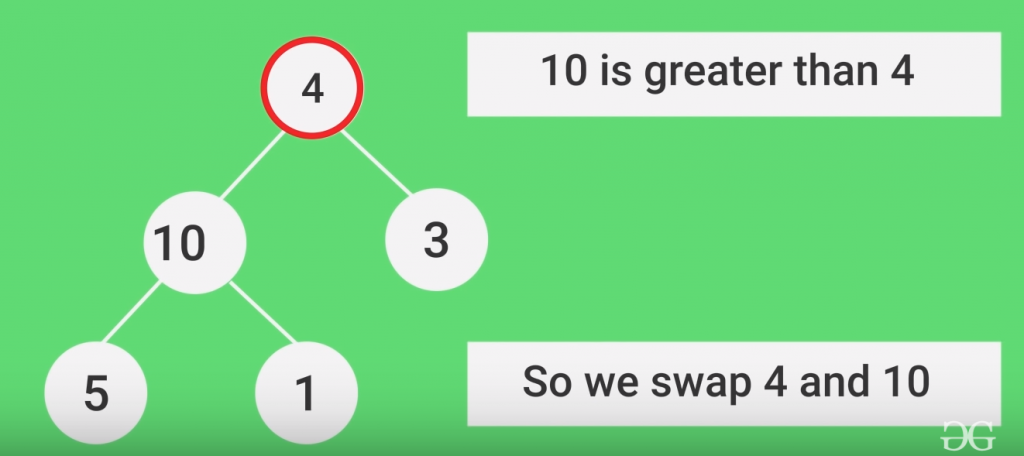
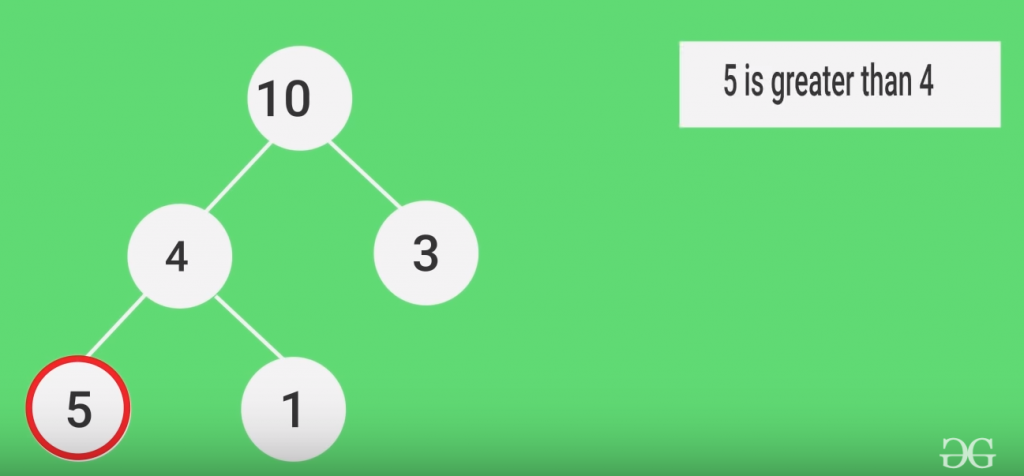
* The root element will be at Arr[0].
* Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

|  |  |
| --- | --- |
| Arr[(i-1)/2] | Returns the parent node |
| Arr[(2\*i)+1] | Returns the left child node |
| Arr[(2\*i)+2] | Returns the right child node |

**Getting Maximum Element**: In a Max-Heap, the maximum element is always present at the root node which is the first element in the array used to represent the Heap. So, the maximum element from a max heap can be simply obtained by returning the root node as Arr[0] in O(1) time complexity.  
  
**Getting Minimum Element**: In a Min-Heap, the minimum element is always present at the root node which is the first element in the array used to represent the Heap. So, the minimum element from a minheap can be simply obtained by returning the root node as Arr[0] in O(1) time complexity.

**Heapifying an Element**

Generally, on inserting a new element onto a Heap, it does not satisfy the property of Heap as stated above on its own. The process of placing the element at the correct location so that it satisfies the Heap property is known as Heapify.  
  
**Heapifying in a Max Heap**: The property of Max Heap says that every node's value must be greater than the values of its children nodes. So, to **heapify**a particular node, swap the value of the node with the maximum value of its children nodes and continue the process until all of the nodes below it satisfies the Heap property.  
  
Consider the below heap as a Max-Heap:  
  
  
In the above heap, node **4**does not follow the Heap property. Let's heapify the root node, ***node 4***.

* **Step 1**: Swap the node 4 with the maximum of its childrens i.e., max(10, 3).  
  
* **Step 2**: Again the node 4 does not follow the heap property. Swap the node 4 with the maximum of its new children's i.e., max(5, 1).  
  
* The Node 4 is now heapified successfully and placed at it's correct position.

**Time Complexity**: The time complexity to heapify a single node is **O(h)**, where h is equal to **log(N)**in a complete binary tree where N is the total number of nodes.

**Topic 2 : Insertion and Deletion in Heaps**

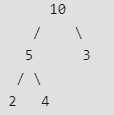
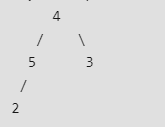
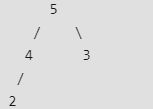
Deletion in Heap

Given a Binary Heap and an element present in the given Heap. The task is to delete an element from this Heap.

The standard deletion operation on Heap is to delete the element present at the root node of the Heap. That is if it is a Max Heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.  
  
**Process of Root Deletion (Or Extract Min in Min Heap)**:  
Deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

* Replace the root or element to be deleted by the last element.
* Delete the last element from the Heap.
* Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, **heapify**the last node placed at the position of root.

**Illustration**:

Suppose the Heap is a Max-Heap as:  
  
  
*The element to be deleted is root, i.e. 10.*  
  
**Process**:  
The last element is 4.  
  
**Step 1:** Replace the last element with root, and delete it.  
  
  
**Step 2**: Heapify root.  
Final Heap:  


**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program for implement deletion in Heaps

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. N is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

Run

Java



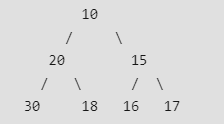
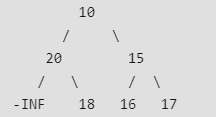
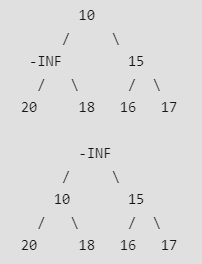
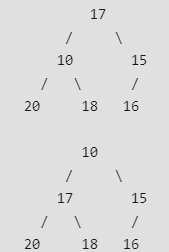
**Output:**

5 4 3 2

**How to Delete Nnn-Root elements?**:  
Below are the steps to delete an element at given index from Min Heap

* Replace the element to be deleted with minus Infinite (INT\_MIN in C++ and Integer.MIN\_VALUE in Java)
* Keep swapping the minus infinite value with parent until it reaches root
* Now delete the root using process discussed above

**Illustration**

Let us try deleting 30 from below Min Heap  
  
  
We replace 30 with -INF  
  
  
We move -INF to root by swapping with parent  
nodes one by one  
  
  
Now Delete Root (We first replace it with last node  
then call heapify)  


Insertion in Heaps

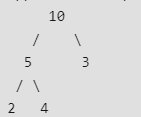
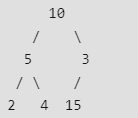
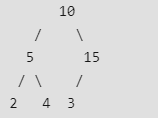
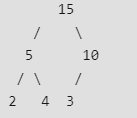
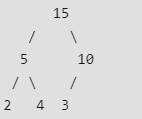
The insertion operation is also similar to that of the deletion process.

***Given a Binary Heap and a new element to be added to this Heap. The task is to insert the new element to the Heap maintaining the properties of Heap.***

**Process of Insertion**: Elements can be inserted to the heap following a similar approach as discussed above for deletion. The idea is to:

* First increase the heap size by 1, so that it can store the new element.
* Insert the new element at the end of the Heap.
* This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, **heapify**this newly inserted element following a bottom-up approach.

**Illustration**:

Suppose the Heap is a Max-Heap as:  
  
  
*The new element to be inserted is 15.*  
  
***Process***:  
**Step 1:** Insert the new element at the end.  
  
  
**Step 2**: Heapify the new element following bottom-up   
 approach.  
-> 15 is less than its parent 3, swap them.  
  
  
-> 15 is again less than its parent 10, swap them.  
  
  
Therefore, the final heap after insertion is:  
-> 15 is less than its parent 3, swap them.  


**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to insert new element to Heap

#include <iostream>

using namespace std;

#define MAX 1000 // Max size of Heap

// Function to heapify ith node in a Heap

// of size n following a Bottom-up approach

void heapify(int arr[], int n, int i)

{

// Find parent

int parent = (i - 1) / 2;

if (parent >= 0) {

// For Max-Heap

// If current node is greater than its parent

// Swap both of them and call heapify again

// for the parent

if (arr[i] > arr[parent]) {

swap(arr[i], arr[parent]);

// Recursively heapify the parent node

heapify(arr, n, parent);

}

}

}

// Function to insert a new node to the Heap

Run

Java



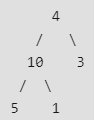
**Output:**

15 5 10 2 4 3

**Topic 3 : Building Heaps**

***Given N elements. The task is to build a Binary Heap from the given array. The heap can be either Max Heap or Min Heap.***

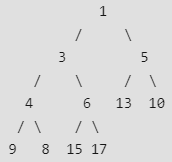
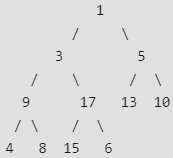
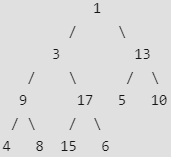
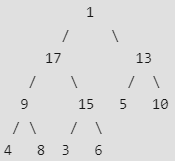
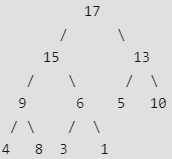
Suppose the given input elements are: 4, 10, 3, 5, 1.  
  
The corresponding complete binary tree for this array of elements [4, 10, 3, 5, 1] will be:

  
  
Root is at index 0 in array.  
Left child of i-th node is at (2\*i + 1)th index.  
Left child of i-th node is at (2\*i + 2)th index.  
Parent of the i-th node is at (i-1)/2 index.

Suppose, we need to build a Max-Heap from the above-given array elements. It can be clearly seen that the above complete binary tree formed does not follow the Heap property. So, the idea is to heapify the complete binary tree formed from the array in reverse level order following top-down approach.  
  
That is first heapify, the last node in level order traversal of the tree, then heapify the second last node and so on.  
  
  
**Time Complexity:** Heapify a single node takes O(Log N) time complexity where N is the total number of Nodes. Therefore, building the entire Heap will take N heapify operations and the total time complexity will be **O(N\*logN)**.  
  
**Optimized Approach**: The above approach can be optimized by observing the fact that the leaf nodes need not to be *heapified*as they already follow the heap property. Also, the array representation of the complete binary tree contains the level order traversal of the tree.  
  
So the idea is to find the position of the last non-leaf node and perform the **heapify**operation of each non-leaf node in reverse level order.

**Last non-leaf node** = parent of last-node.  
or, Last non-leaf node = parent of node at (n-1)th index.  
or, Last non-leaf node = Node at index ((n-1) - 1)/2.  
 = (n - 2)/2.

**Illustration**:

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}  
  
Corresponding Complete Binary Tree is:  
  
  
***The task is to build a Max-Heap from the above array***.  
  
Total Nodes = 11.  
Last Non-leaf node index = (11/2) - 1 = 4.  
Therefore, the last non-leaf node = 6.  
  
To build the heap, heapify only the nodes:  
[1, 3, 5, 4, 6] in reverse order.  
  
**Heapify 6**: Swap 6 and 17.  
  
  
**Heapify 4**: Swap 4 and 9.  
  
  
**Heapify 5**: Swap 13 and 5.  
  
  
**Heapify 3**: First Swap 3 and 17, again swap 3 and 15.  
  
  
**Heapify 1**: First Swap 1 and 17, again swap 1 and 15,   
 finally swap 1 and 6.  


**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program for building Heap from Array

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. N is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

Run

Java



**Output:**

Array representation of Heap is:  
17 15 13 9 6 5 10 4 8 3 1

**Topic 4 : Implementing Heaps using inbuilt classes in**

**C++ and Java**

Heaps in C++

In C++ the priority\_queue container can be used to implement Heaps. The priority\_queue in C++ is a built-in container class which is used to implement priority queues easily. This container uses max heap internally to implement a priority queue efficiently. Therefore, it can be also used in-place of max heaps.  
  
This container can also be modified by passing some additional parameters to be used as a Min Heap.  
  
**Syntax**:

* For implementing Max Heap:

priority\_queue< *type\_of\_data* > *name\_of\_heap*;

* For implementing Min Heap:

priority\_queue< *type*, vector<*type*>, greater<*type*> > *name\_of\_heap*;

**Methods of priority queue:**

* **priority\_queue::empty()**: The empty() function returns whether the queue is empty.
* **priority\_queue::size()**: The size() function returns the size of the queue.
* **priority\_queue::top()**: This function returns a reference to the top most element of the queue.
* **priority\_queue::push()**: The push(g) function adds the element ‘g’ at the end of the queue.
* **priority\_queue::pop()**: The pop() function deletes the first element of the queue.
* **priority\_queue::swap()**: This function is used to swap the contents of one priority queue with another priority queue of the same type and size.

**Note**: The above functions are applicable in case of both Max-Heap and Min-heap implementation of priority\_queue container.  
  
**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to implement Max Heap and Min Heap

// using priority\_queue in C++ STL

#include <iostream>

#include <queue>

using namespace std;

int main ()

{

// DECLARING MAX HEAP

priority\_queue <int> max\_heap;

// Add elements to the Max Heap

// in any order

max\_heap.push(10);

max\_heap.push(30);

max\_heap.push(20);

max\_heap.push(5);

max\_heap.push(1);

// Print element at top of the heap

// every time and remove it until the

// heap is not empty

cout<<"Element at top of Max Heap at every step:\n";

while(!max\_heap.empty())

{

// Print Top Element

cout<<max\_heap.top()<<" ";

Run

**Output:**

Element at top of Max Heap at every step:  
30 20 10 5 1   
  
Element at top of Min Heap at every step:  
1 5 10 20 30

Heaps in Java

Java also provides us with a built-in class named PriorityQueue which can be used to implement both Max heap and Min heap easily and efficiently.  
  
By default, the PriorityQueue class implements a Min-Heap. However, it can also be modified by using a comparator function to implement Max-Heap as well as shown in the below syntax.  
  
**Syntax**:

* Implementing Max Heap:

Queue<Integer> max\_heap = new PriorityQueue<>(   
 Collections.reverseOrder());

* Implementing Min Heap:

Queue<Integer> min\_heap = new PriorityQueue<>();

**Some useful method of PriorityQueue class in Java:**

* **boolean add(E element)**: This method inserts the specified element into this priority queue.
* **public remove()**: This method removes a single instance of the specified element from this queue, if it is present.
* **public poll()**: This method retrieves and removes the head of this queue, or returns null if this queue is empty.
* **public peek()**: This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
* **void clear()**: This method is used to remove all of the contents of the priority queue.
* **int size()**: The method is used to return the number of elements present in the set.

**Note**: The above methods are applicable in case of both Max-Heap and Min-heap implementation of PriorityQueue class.  
  
**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Java Program to implement Max heap and Min heap

// using the PriorityQueue class

import java.util.\*;

import java.io.\*;

import java.lang.\*;

class GFG

{

public static void main (String[] args) {

// DECLARING MAX HEAP

Queue<Integer> max\_heap = new PriorityQueue<>(

Collections.reverseOrder());

// Add elements to the Max Heap

// in any order

max\_heap.add(10);

max\_heap.add(30);

max\_heap.add(20);

max\_heap.add(5);

max\_heap.add(1);

// Print element at top of the heap

// every time and remove it until the

// heap is not empty

System.out.print("Element at top of Max Heap at"

+ " every step:\n");

while(!max\_heap.isEmpty())

{

Run

**Output:**

Some error occured or ide is down,please try again

**Topic 5 : Sample Problems on Binary Heaps**

**Problem #1 : Heap Sort - Ascending Order**

**Description -**Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining element.  
**Solution -**Heap Sort Algorithm for sorting in increasing order-

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat the above steps while the size of the heap is greater than 1.

**Pseudo Code**

// To heapify a subtree rooted with node i which is  
// an index in arr[]. n is size of heap  
void heapify( arr[], n, i)  
{  
 largest = i // Initialize largest as root  
 l = 2\*i + 1 // left = 2\*i + 1  
 r = 2\*i + 2 // right = 2\*i + 2  
  
 // If left child is larger than root  
 if (l < n && arr[l] > arr[largest])  
 largest = l  
  
 // If right child is larger than largest so far  
 if (r < n && arr[r] > arr[largest])  
 largest = r  
  
 // If largest is not root  
 if (largest != i)  
 {  
 swap(arr[i], arr[largest])  
  
 // Recursively heapify the affected sub-tree  
 heapify(arr, n, largest)  
 }  
}  
  
// main function to do heap sort  
void heapSort( arr[], n)  
{  
 // Build heap (rearrange array)  
 for (int i = n / 2 - 1; i >= 0; i--)  
 heapify(arr, n, i)  
  
 // One by one extract an element from heap  
 for (int i=n-1; i>=0; i--)  
 {  
 // Move current root to end  
 swap(arr[0], arr[i])  
  
 // call max heapify on the reduced heap  
 heapify(arr, i, 0)  
 }  
}

**Problem #2 : K Th Smallest/Largest Element in Unsorted Array**

**Description -** Given an array and a number k where k is smaller than size of array, we need to find the kth smallest element in the given array. It is given that all array elements are distinct.  
**Solution -**The idea is to create Min Heap and extract k elements from the element, kth extracted element will be the kth smallest element in the array.  
**Pseudo Code**

void MinHeap( a[], size)   
{   
 heap\_size = size  
 harr = a // store address of array   
 int i = (heap\_size - 1)/2  
 while (i >= 0)   
 {   
 MinHeapify(i)  
 i--  
 }   
}   
   
// Method to remove minimum element (or root) from min heap   
int extractMin()   
{   
 if (heap\_size == 0)   
 return INT\_MAX  
   
 // Store the minimum vakue.   
 int root = harr[0]  
   
 // If there are more than 1 items, move the last item to root   
 // and call heapify.   
 if (heap\_size > 1)   
 {   
 harr[0] = harr[heap\_size-1]  
 MinHeapify(0)  
 }   
 heap\_size--  
   
 return root  
}   
// A recursive method to heapify a subtree with root at given index   
// This method assumes that the subtrees are already heapified   
void MinHeapify(int i)   
{   
 int l = left(i)  
 int r = right(i)  
 int smallest = i;   
 if (l < heap\_size && harr[l] < harr[i])   
 smallest = l  
 if (r < heap\_size && harr[r] < harr[smallest])   
 smallest = r  
 if (smallest != i)   
 {   
 swap(& harr[i], & harr[smallest])  
 MinHeapify(smallest)  
 }   
}   
// Returns Minimum Element   
int getMin()   
{  
 return harr[0]  
}  
// Function to return k'th smallest element in a given array   
int kthSmallest( arr[], n, k)   
{   
 // Build a heap of n elements: O(n) time   
 MinHeap(arr, n)  
   
 // Do extract min (k-1) times   
 for (int i=0; i < k-1; i++)   
 extractMin()  
   
 // Return root   
 return getMin()  
}

**Problem #3 : Median of Stream of Running Integers using STL**

**Description -**Given that integers are being read from a data stream. Find the median of all the elements read so far starting from the first integer till the last integer. This is also called Median of Running Integers. The data stream can be any source of data, example: a file, an array of integers, input stream etc.

**What is Median?**

Median can be defined as the element in the data set which separates the higher half of the data sample from the lower half. In other words we can get the median element as, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick the average of the middle two elements in the sorted stream.  
**Example**:

Input: 5 10 15  
Output: 5  
 7.5  
 10

**Explanation**: Given the input stream as an array of integers [5,10,15]. We will now read integers one by one and print the median correspondingly. So, after reading first element 5, the median is 5. After reading 10,median is 7.5 After reading 15 ,median is 10.  
**Solution -** The idea is to use max heap and min heap to store the elements of higher half and lower half. Max heap and min heap can be implemented using priority\_queue in C++ STL. Below is the step by step algorithm to solve this problem.

1. Create two heaps. One max heap to maintain elements of lower half and one min heap to maintain elements of higher half at any point of time..
2. Take the initial value of median as 0.
3. For every newly read element, insert it into either max heap or min heap and calculate the median based on the following conditions:  
   * If the size of max heap is greater than size of min heap and the element is less than previous median then pop the top element from max heap and insert into min heap and insert the new element to max heap else insert the new element to min heap. Calculate the new median as the average of top of elements of both max and min heap.
   * If the size of max heap is less than size of min heap and the element is greater than previous median then pop the top element from min heap and insert into max heap and insert the new element to min heap else insert the new element to max heap. Calculate the new median as the average of top of elements of both max and min heap.
   * If the size of both heaps are the same. Then check if the current is less than the previous median or not. If the current element is less than the previous median then insert it to the max heap and the new median will be equal to the top element of max heap. If the current element is greater than the previous median then insert it to the min heap and the new median will be equal to the top element of the min heap.

**Pseudo Code**

// function to calculate median of stream  
void printMedian(double arr[], int n)  
{  
 // max heap to store the smaller half elements  
 priority\_queue s;  
   
 // min heap to store the greater half elements  
 priority\_queue<double,vector,greater > g;  
   
 double med = arr[0];  
 s.push(arr[0]);  
   
 print(med)  
   
 // reading elements of stream one by one  
 /\* At any time we try to make heaps balanced and  
 their sizes differ by at-most 1. If heaps are  
 balanced,then we declare median as average of  
 min\_heap\_right.top() and max\_heap\_left.top()  
 If heaps are unbalanced,then median is defined  
 as the top element of heap of larger size \*/  
 for (int i=1; i < n; i++)  
 {  
 double x = arr[i];  
   
 // case1(left side heap has more elements)  
 if (s.size() > g.size())  
 {  
 if (x < med)  
 {  
 g.push(s.top());  
 s.pop();  
 s.push(x);  
 }  
 else  
 g.push(x);  
   
 med = (s.top() + g.top())/2.0;  
 }  
   
 // case2(both heaps are balanced)  
 else if (s.size()==g.size())  
 {  
 if (x < med)  
 {  
 s.push(x);  
 med = (double)s.top();  
 }  
 else  
 {  
 g.push(x);  
 med = (double)g.top();  
 }  
 }  
   
 // case3(right side heap has more elements)  
 else  
 {  
 if (x > med)  
 {  
 s.push(g.top());  
 g.pop();  
 g.push(x);  
 }  
 else  
 s.push(x);  
   
 med = (s.top() + g.top())/2.0;  
 }  
   
 print(med)  
 }  
}  
</double,vector

**Time Complexity:** O(n Log n)  
**Auxiliary Space :** O(n)