1. Maximum element in min heap

Given a min heap, find the maximum element present in the heap.

Examples:

```
Input :      10

          /    \

         25     23

       /  \    / \

      45   30  50  40

Output : 50


Input :     20

          /   \

         40    28

Output : 40
```

Brute force approach:

We can check all the nodes in the min heap to get the maximum element. Note that this approach works on any binary tree and does not makes use of any property of the min heap. It has a time and space complexity of O(n). Since min heap is a complete binary tree, we generally use arrays to store them, so we can check all the nodes by simply traversing the array. If the heap is stored using pointers, then we can use recursion to check all the nodes.

Below is the implementation of above approach:

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// Function to find the
// maximum element in a
// min heap
int findMaximumElement(int heap[], int n)
```

```
{
    int maximumElement = heap[0];

    for (int i = 1; i < n; ++i)
        maximumElement = max(maximumElement, heap[i]);

    return maximumElement;
}

// Driver code
int main()
{
    // Number of nodes
    int n = 10;

    // heap represents the following min heap:
    //     10
    //    / \
    //  25     23
    //  / \   / \
    // 45 50 30 35
    // / \ /
    //63 65 81
    int heap[] = { 10, 25, 23, 45, 50, 30, 35, 63, 65, 81 };

    cout << findMaximumElement(heap, n);

    return 0;
}
```

Output:

```
81
```

Efficient approach:

The min heap property requires that the parent node be lesser than its child node(s). Due to this, we can conclude that a non-leaf node cannot be the maximum element as its child node has a higher value. So we can narrow down our search space to only leaf nodes. In a min heap having n elements, there are ceil(n/2) leaf nodes. The time and space complexity remains O(n) as a constant factor of 1/2 does not affect the asymptotic complexity.

Below is the implementation of above approach:

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to find the
// maximumelement in a
// max heap
int findMaximumElement(int heap[], int n)
{
    int maximumElement = heap[n / 2];

    for (int i = 1 + n / 2; i < n; ++i)
        maximumElement = max(maximumElement, heap[i]);

    return maximumElement;
}

// Driver code
int main()
{
    // Number of nodes
    int n = 10;

    // heap represents the following min heap:
    //     10
    //    / \
    //  25     23
    //  / \   / \
    // 45 50 30 35
    // / \ /
    //63 65 81
    int heap[] = { 10, 25, 23, 45, 50, 30, 35, 63, 65, 81 };

    cout << findMaximumElement(heap, n);

    return 0;
}
```

Output:

```
81
```

2. Minimum element in a max heap

Given a max heap, find the minimum element present in the heap.

Examples:

```
Input :     100

          /    \

         75     50

        /  \    / \

      55   10  2  40

Output : 2


Input :     20

          /   \

         4    18

Output : 4
```

Brute force approach:

We can check all the nodes in the max heap to get the minimum element. Note that this approach works on any binary tree and does not makes use of any property of the max heap. It has a time and space complexity of O(n). Since max heap is a complete binary tree, we generally use arrays to store them, so we can check all the nodes by simply traversing the array. If the heap is stored using pointers, then we can use recursion to check all the nodes.

Below is the implementation of above approach:

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// Function to find the
// minimum element in a
// max heap
int findMinimumElement(int heap[], int n)
```

```
{
    int minimumElement = heap[0];

    for (int i = 1; i < n; ++i)
        minimumElement = min(minimumElement, heap[i]);

    return minimumElement;
}

// Driver code
int main()
{
    // Number of nodes
    int n = 10;
    // heap represents the following max heap:
    //          20
    //        /    \
    //       18     10
    //    /     \    /  \
    //    12      9  9   3
    //   /  \   /
    // 5    6 8
    int heap[] = { 20, 18, 10, 12, 9, 9, 3, 5, 6, 8 };

    cout << findMinimumElement(heap, n);

    return 0;
}
```

Output:

```
3
```

Efficient approach:

The max heap property requires that the parent node be greater than its child node(s). Due to this, we can conclude that a non-leaf node cannot be the minimum element as its child node has a lower value. So we can narrow down our search space to only leaf nodes. In a max heap having n elements, there are ceil(n/2) leaf nodes. The time and space complexity remains O(n) as a constant factor of 1/2 does not affect the asymptotic complexity.

Below is the implementation of above approach:

```
// C++ implementation of above approach
#include <bits/stdc++.h>
using namespace std;

// Function to find the
// minimum element in a
// max heap
int findMinimumElement(int heap[], int n)
{
    int minimumElement = heap[n/2];

    for (int i = 1 + n / 2; i < n; ++i)
        minimumElement = min(minimumElement, heap[i]);

    return minimumElement;
}

// Driver code
int main()
{
    // Number of nodes
    int n = 10;
    // heap represents the following max heap:
    //         20
    //        /    \
    //      18      10
    //     /    \    /  \
    //    12      9  9   3
    //   /  \   /
    // 5    6 8
    int heap[] = { 20, 18, 10, 12, 9, 9, 3, 5, 6, 8 };

    cout << findMinimumElement(heap, n);

    return 0;
}
```

Output:

```
3
```

3. 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 Deletion:

Since 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:

     10

   /    \

  5      3

 / \

2   4


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.
     4

   /    \

  5      3

 /

2


Step 2: Heapify root.
Final Heap:

     5

   /    \

  4      3

 /

2
```

Implementation:

```cpp
// 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);
    }
}

// Function to delete the root from Heap
void deleteRoot(int arr[], int& n)
{
    // Get the last element
    int lastElement = arr[n - 1];

    // Replace root with first element
    arr[0] = lastElement;

    // Decrease size of heap by 1
    n = n - 1;

    // heapify the root node
    heapify(arr, n, 0);
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver Code
int main()
{
    // Array representation of Max-Heap
    // 10
    //    /  \
    // 5    3
    //   / \
    // 2   4
    int arr[] = { 10, 5, 3, 2, 4 };

    int n = sizeof(arr) / sizeof(arr[0]);
    deleteRoot(arr, n);

    printArray(arr, n);

    return 0;
}
```

Output:

```
5 4 3 2
```

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:

     10

   /    \

  5      3

 / \

2   4



The new element to be inserted is 15.


Process:
```

```
Step 1: Insert the new element at the end.
     10

   /    \

  5      3

 / \    /

2   4  15


Step 2: Heapify the new element following bottom-up
       approach.

-> 15 is more than its parent 3, swap them.

      10

   /    \

  5      15

 / \    /

2   4  3


-> 15 is again more than its parent 10, swap them.

      15

   /    \

  5      10

 / \    /

2   4  3


Therefore, the final heap after insertion is:

      15

   /    \
   5      10

  / \    /

 2   4  3
```

Implementation:

```cpp
// 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 (arr[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
void insertNode(int arr[], int& n, int Key)
{
    // Increase the size of Heap by 1
    n = n + 1;

    // Insert the element at end of Heap
    arr[n - 1] = Key;

    // Heapify the new node following a
    // Bottom-up approach
    heapify(arr, n, n - 1);
}
```

```
// A utility function to print array of size n
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    cout << "\n";
}

// Driver Code
int main()
{
    // Array representation of Max-Heap
    // 10
    //    /  \
    // 5    3
    //  / \
    // 2   4
    int arr[MAX] = { 10, 5, 3, 2, 4 };

    int n = 5;

    int key = 15;

    insertNode(arr, n, key);

    printArray(arr, n);
    // Final Heap will be:
    // 15
    //    /   \
    // 5      10
    //  / \   /
    // 2   4 3
    return 0;
}
```

Output:

```
15 5 10 2 4 3
```

4. k largest(or smallest) elements in an array | added Min Heap method

Question: Write an efficient program for printing k largest elements in an array. Elements in array can be in any order.

For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Method 1 (Use Bubble k times)

Thanks to Shailendra for suggesting this approach.

1) Modify Bubble Sort to run the outer loop at most k times.

2) Print the last k elements of the array obtained in step 1.

Time Complexity: O(nk)

Like Bubble sort, other sorting algorithms like Selection Sort can also be modified to get the k largest elements.

Method 2 (Use temporary array)

K largest elements from arr[0..n-1]

1) Store the first k elements in a temporary array temp[0..k-1].

2) Find the smallest element in temp[], let the smallest element be min.

3-a) For each element x in arr[k] to arr[n-1]. O(n-k)

If x is greater than the min then remove min from temp[] and insert x.

3-b)Then, determine the new min from temp[]. O(k)

4) Print final k elements of temp[]

Time Complexity: O((n-k)*k). If we want the output sorted then O((n-k)*k + klogk)

Thanks to nesamani1822 for suggesting this method.

Method 3(Use Sorting)

1) Sort the elements in descending order in O(nLogn)

2) Print the first k numbers of the sorted array O(k).

Following is the implementation of above.

```
// C++ code for k largest elements in an array
#include <bits/stdc++.h>
using namespace std;

void kLargest(int arr[], int n, int k)
{
    // Sort the given array arr in reverse
    // order.
    sort(arr, arr + n, greater<int>());

    // Print the first kth largest elements
    for (int i = 0; i < k; i++)
        cout << arr[i] << " ";
}

// driver program
int main()
{
    int arr[] = { 1, 23, 12, 9, 30, 2, 50 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;
    kLargest(arr, n, k);
}

// This article is contributed by Chhavi
```

Output:

```
50 30 23
```

Time complexity: O(nlogn)

Method 4 (Use Max Heap)

1) Build a Max Heap tree in O(n)

2) Use Extract Max k times to get k maximum elements from the Max Heap

O(klogn)

Time complexity: O(n + klogn)

Method 5(Use Oder Statistics)

1) Use order statistic algorithm to find the kth largest element. Please see the topic selection in worst-case linear time O(n)

2) Use QuickSort Partition algorithm to partition around the kth largest number O(n).

3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.

Time complexity: O(n) if we don’t need the sorted output, otherwise O(n+kLogk)

Thanks to Shilpi for suggesting the first two approaches.

Method 6 (Use Min Heap)

This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.

1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.

……a) If the element is greater than the root then make it root and call heapify for MH

……b) Else ignore it.

// The step 2 is O((n-k)*logk)

3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

All of the above methods can also be used to find the kth largest (or smallest) element.

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

// Swap function to interchange
// the value of variables x and y
int swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

// Min Heap Class
// arr holds reference to an integer
// array size indicate the number of
// elements in Min Heap
class MinHeap {

    int size;
    int* arr;

public:
    // Constructor to initialize the size and arr
    MinHeap(int size, int input[]);

    // Min Heapify function, that assumes that
    // 2*i+1 and 2*i+2 are min heap and fix the
    // heap property for i.
    void heapify(int i);

    // Build the min heap, by calling heapify
    // for all non-leaf nodes.
    void buildHeap();
};

// Constructor to initialize data
// members and creating mean heap
MinHeap::MinHeap(int size, int input[])
{
    // Initializing arr and size

    this->size = size;
    this->arr = input;

    // Building the Min Heap
    buildHeap();
}

// Min Heapify function, that assumes
// 2*i+1 and 2*i+2 are min heap and
// fix min heap property for i

void MinHeap::heapify(int i)
{
    // If Leaf Node, Simply return
    if (i >= size / 2)
        return;
    // variable to store the smallest element
    // index out of i, 2*i+1 and 2*i+2
    int smallest;

    // Index of left node
    int left = 2 * i + 1;

    // Index of right node
    int right = 2 * i + 2;

    // Select minimum from left node and
    // current node i, and store the minimum
    // index in smallest variable
    smallest = arr[left] < arr[i] ? left : i;

    // If right child exist, compare and
    // update the smallest variable
    if (right < size)
        smallest = arr[right] < arr[smallest]
                                ? right : smallest;

    // If Node i violates the min heap
    // property, swap  current node i with
    // smallest to fix the min-heap property
    // and recursively call heapify for node smallest.
    if (smallest != i) {
        swap(arr[i], arr[smallest]);
        heapify(smallest);
    }
}

// Build Min Heap
void MinHeap::buildHeap()
{
    // Calling Heapify for all non leaf nodes
    for (int i = size / 2 - 1; i >= 0; i--) {
        heapify(i);
    }
}

void FirstKelements(int arr[],int size,int k){
    // Creating Min Heap for given
    // array with only k elements
    MinHeap* m = new MinHeap(k, arr);

    // Loop For each element in array
    // after the kth element
    for (int i = k; i < size; i++) {

        // if current element is smaller
        // than minimum element, do nothing
        // and continue to next element
        if (arr[0] > arr[i])
            continue;

        // Otherwise Change minimum element to
        // current element, and call heapify to
        // restore the heap property
        else {
            arr[0] = arr[i];
            m->heapify(0);
        }
    }
    // Now min heap contains k maximum
    // elements, Iterate and print
    for (int i = 0; i < k; i++) {
        cout << arr[i] << " ";
    }
}
// Driver Program
int main()
{

    int arr[] = { 11, 3, 2, 1, 15, 5, 4,
                           45, 88, 96, 50, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    // Size of Min Heap
    int k = 3;

    FirstKelements(arr,size,k);

    return 0;
}
// This code is contributed by Ankur Goel
```

Output:

```
50 88 96
```

5. Median in a stream of integers (running integers)

Given that integers are read from a data stream. Find median of elements read so for in efficient way. For simplicity assume there are no duplicates. For example, let us consider the stream 5, 15, 1, 3 …

```
After reading 1st element of stream - 5 -> median - 5

After reading 2nd element of stream - 5, 15 -> median - 10

After reading 3rd element of stream - 5, 15, 1 -> median - 5

After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so
on...
```

Making it clear, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick average of middle two elements in sorted stream.

Note that output is effective median of integers read from the stream so far. Such an algorithm is called online algorithm. Any algorithm that can guarantee output of i-elements after processing i-th element, is said to be online algorithm. Let us discuss three solutions for the above problem.

Method 1: Insertion Sort

If we can sort the data as it appears, we can easily locate median element. Insertion Sort is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting i-th element, the first i elements of array are sorted. The insertion sort doesn’t depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting next element. This is the key part of insertion sort that makes it an online algorithm.

However, insertion sort takes $O(n^2)$ time to sort n elements. Perhaps we can use binary search on insertion sort to find location of next element in O(log n) time. Yet, we can’t do data movement in O(log n) time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.

Interested reader can try implementation of Method 1.

Method 2: Augmented self balanced binary search tree (AVL, RB, etc…)
At every node of BST, maintain number of elements in the subtree rooted at that node. We can use a node as root of simple binary tree, whose left child is self balancing BST with elements less than root and right child is self balancing BST with elements greater than root. The root element always holds effective median.
If left and right subtrees contain same number of elements, root node holds average of left and right subtree root data. Otherwise, root contains same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.

Self balancing BST is costly in managing balancing factor of BST. However, they provide sorted data which we don't need. We need median only. The next method make use of Heaps to trace median.

Method 3: Heaps
Similar to balancing BST in Method 2 above, we can use a max heap on left side to represent elements that are less than effective median, and a min heap on right side to represent elements that are greater than effective median.

After processing an incoming element, the number of elements in heaps differ utmost by 1 element. When both heaps contain same number of elements, we pick average of heaps root data as effective median. When the heaps are not balanced, we select effective median from the root of heap containing more elements.
Given below is implementation of above method. For algorithm to build these heaps, please read the highlighted code.

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

// Heap capacity
```

```cpp
#define MAX_HEAP_SIZE (128)
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

//// Utility functions

// exchange a and b
inline
void Exch(int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}

// Greater and Smaller are used as comparators
bool Greater(int a, int b)
{
    return a > b;
}

bool Smaller(int a, int b)
{
    return a < b;
}

int Average(int a, int b)
{
    return (a + b) / 2;
}

// Signum function
// = 0  if a == b  - heaps are balanced
// = -1 if a < b   - left contains less elements than right
// = 1  if a > b   - left contains more elements than right
int Signum(int a, int b)
{
    if( a == b )
        return 0;

    return a < b ? -1 : 1;
}

// Heap implementation
// The functionality is embedded into
// Heap abstract class to avoid code duplication
class Heap
{
public:
    // Initializes heap array and comparator required
    // in heapification
    Heap(int *b, bool (*c)(int, int)) : A(b), comp(c)
    {
        heapSize = -1;
    }

    // Frees up dynamic memory
    virtual ~Heap()
    {
        if( A )
        {
            delete[] A;
        }
    }

    // We need only these four interfaces of Heap ADT
    virtual bool Insert(int e) = 0;
    virtual int  GetTop() = 0;
    virtual int  ExtractTop() = 0;
    virtual int  GetCount() = 0;

protected:

    // We are also using location 0 of array
    int left(int i)
    {
        return 2 * i + 1;
    }

    int right(int i)
    {
        return 2 * (i + 1);
    }

    int parent(int i)
    {
        if( i <= 0 )
        {
            return -1;
        }

        return (i - 1)/2;
    }

    // Heap array
    int   *A;
    // Comparator
    bool  (*comp)(int, int);
    // Heap size
    int   heapSize;
```

```
// Returns top element of heap data structure
int top(void)
{
    int max = -1;

    if( heapSize >= 0 )
    {
        max = A[0];
    }

    return max;
}

// Returns number of elements in heap
int count()
{
    return heapSize + 1;
}

// Heapification
// Note that, for the current median tracing problem
// we need to heapify only towards root, always
void heapify(int i)
{
    int p = parent(i);

    // comp - differentiate MaxHeap and MinHeap
    // percolates up
    if( p >= 0 && comp(A[i], A[p]) )
    {
        Exch(A[i], A[p]);
        heapify(p);
    }
}

// Deletes root of heap
int deleteTop()
{
    int del = -1;

    if( heapSize > -1)
    {
        del = A[0];

        Exch(A[0], A[heapSize]);
        heapSize--;
        heapify(parent(heapSize+1));
    }
```

```
        return del;
    }

    // Helper to insert key into Heap
    bool insertHelper(int key)
    {
        bool ret = false;

        if( heapSize < MAX_HEAP_SIZE )
        {
            ret = true;
            heapSize++;
            A[heapSize] = key;
            heapify(heapSize);
        }

        return ret;
    }
};

// Specilization of Heap to define MaxHeap
class MaxHeap : public Heap
{
private:

public:
    MaxHeap() : Heap(new int[MAX_HEAP_SIZE], &Greater)  {  }

    ~MaxHeap()  { }

    // Wrapper to return root of Max Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Max Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Max Heap
    int  GetCount()
    {
        return count();
    }
    // Wrapper to insert into Max Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Specilization of Heap to define MinHeap
class MinHeap : public Heap
{
private:

public:

    MinHeap() : Heap(new int[MAX_HEAP_SIZE], &Smaller) { }

    ~MinHeap() { }

    // Wrapper to return root of Min Heap
    int GetTop()
    {
        return top();
    }

    // Wrapper to delete and return root of Min Heap
    int ExtractTop()
    {
        return deleteTop();
    }

    // Wrapper to return # elements of Min Heap
    int  GetCount()
    {
        return count();
    }

    // Wrapper to insert into Min Heap
    bool Insert(int key)
    {
        return insertHelper(key);
    }
};

// Function implementing algorithm to find median so far.
int getMedian(int e, int &m, Heap &l, Heap &r)
{
    // Are heaps balanced? If yes, sig will be 0
    int sig = Signum(l.GetCount(), r.GetCount());
    switch(sig)
    {
    case 1: // There are more elements in left (max) heap

        if( e < m ) // current element fits in left (max) heap
        {
            // Remore top element from left heap and
            // insert into right heap
            r.Insert(l.ExtractTop());

            // current element fits in left (max) heap
            l.Insert(e);
        }
        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());

        break;

    case 0: // The left and right heaps contain same number of
elements

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
            m = l.GetTop();
        }
        else
        {
            // current element fits in right (min) heap
            r.Insert(e);
            m = r.GetTop();
        }

        break;

    case -1: // There are more elements in right (min) heap

        if( e < m ) // current element fits in left (max) heap
        {
            l.Insert(e);
        }
        else
        {
            // Remove top element from right heap and
            // insert into left heap
            l.Insert(r.ExtractTop());

            // current element fits in right (min) heap
            r.Insert(e);
        }

        // Both heaps are balanced
        m = Average(l.GetTop(), r.GetTop());

        break;
    }

    // No need to return, m already updated
    return m;
}

void printMedian(int A[], int size)
{
    int m = 0; // effective median
    Heap *left  = new MaxHeap();
    Heap *right = new MinHeap();

    for(int i = 0; i < size; i++)
    {
        m = getMedian(A[i], m, *left, *right);

        cout << m << endl;
    }

    // C++ more flexible, ensure no leaks
    delete left;
    delete right;
}
// Driver code
int main()
{
    int A[] = {5, 15, 1, 3, 2, 8, 7, 9, 10, 6, 11, 4};
    int size = ARRAY_SIZE(A);

    // In lieu of A, we can also use data read from a stream
    printMedian(A, size);

    return 0;
}
```

Time Complexity: If we omit the way how stream was read, complexity of median finding is O(N log N), as we need to read the stream, and due to heap insertions/deletions.

At first glance the above code may look complex. If you read the code carefully, it is simple algorithm.

6. Median of Stream of Running Integers using STL

Given that integers are being read from a data stream. Find 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 average of middle two elements in 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,median is 5. After reading 10,median is 7.5 After reading 15 ,median is 10.

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.

Algorithm:

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 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 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 average of top of elements of both max and min heap.
    - If the size of both heaps are same. Then check if current is less than previous median or not. If the current element is less than previous median then insert it to max heap and new median will be equal to top element of max heap. If the current element is greater than previous median then insert it to min heap and new median will be equal to top element of min heap.

Below is the implementation of above approach:

```
// C++ program to find med in
// stream of running integers
#include<bits/stdc++.h>
using namespace std;

// function to calculate med of stream
void printMedians(double arr[], int n)
{
    // max heap to store the smaller half elements
    priority_queue<double> s;

    // min heap to store the greater half elements
    priority_queue<double,vector<double>,greater<double> > g;

    double med = arr[0];
    s.push(arr[0]);

    cout << med << endl;
```

```
// 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;
        }

        cout << med << endl;
    }
}

// Driver program to test above functions
int main()
{
    // stream of integers
    double arr[] = {5, 15, 10, 20, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printMedians(arr, n);
    return 0;
}
```

Output:

```
5

10

10

12.5

10
```

Time Complexity: O(n Log n)

Auxiliary Space : O(n

7. Find smallest range containing elements from k lists

Given k sorted lists of integers of size n each, find the smallest range that includes at least element from each of the k lists. If more than one smallest ranges are found, print any one of them.

Example:

```
Input: K = 3
arr1[] : [4, 7, 9, 12, 15]

arr2[] : [0, 8, 10, 14, 20]

arr3[] : [6, 12, 16, 30, 50]

Output:
The smallest range is [6 8]


Explanation: Smallest range is formed by
number 7 from the first list, 8 from second

list and 6 from the third list.


Input: k = 3
arr1[] : [4, 7]

arr2[] : [1, 2]

arr3[] : [20, 40]

Output:
The smallest range is [2 20]


Explanation:The range [2, 20] contains 2, 4, 7, 20
which contains element from all the three arrays.
```

<u>Naive Approach:</u> Given K sorted list, find a range where there is at least one element from every list. The idea to solve the problem is very simple, keep k

pointers which will constitute the elements in the range, by taking the min and max of the k elements the range can be formed. Initially, all the pointers will point to the start of all the k arrays. Store the range max to min. If the range has to be minimised then either the minimum value has to be increased or maximum value has to be decreased. The maximum value cannot be decreased as the array is sorted but the minimum value can be increased. To continue increasing the minimum value, increase the pointer of the list containing the minimum value and update the range until one of the lists exhausts.

- Algorithm:
    1. Create an extra space *ptr* of length k to store the pointers and a variable *minrange* initilized to a maximum value.
    2. Initially the index of every list is 0, therefore initialize every element of *ptr[0..k]* to 0, the array ptr will store the index of the elements in the range.
    3. Repeat the following steps until atleast one list exhausts:
        1. Now find the minimum and maximum value among the current elements of all the list pointed by the ptr[0…k] array.
        2. Now update the minrange if current (max-min) is less than minrange.
        3. increment the pointer pointing to current minimum element.
- Implementation:

```
// C++ program to finds out smallest range that includes
// elements from each of the given sorted lists.
#include <bits/stdc++.h>

using namespace std;

#define N 5

// array for storing the current index of list i
int ptr[501];

// This function takes an k sorted lists in the form of
// 2D array as an argument. It finds out smallest range
// that includes elements from each of the k lists.
void findSmallestRange(int arr[][N], int n, int k)
{
    int i, minval, maxval, minrange, minel, maxel, flag,
minind;

    // initializing to 0 index;
    for (i = 0; i <= k; i++)
        ptr[i] = 0;

    minrange = INT_MAX;

    while (1) {
        // for mainting the index of list containing the
minimum element
        minind = -1;
        minval = INT_MAX;
        maxval = INT_MIN;
        flag = 0;

        // iterating over all the list
        for (i = 0; i < k; i++) {
            // if every element of list[i] is traversed then
break the loop
            if (ptr[i] == n) {
                flag = 1;
                break;
            }
            // find minimum value among all the list elements
pointing by the ptr[] array
            if (ptr[i] < n && arr[i][ptr[i]] < minval) {
                minind = i; // update the index of the list
                minval = arr[i][ptr[i]];
            }
            // find maximum value among all the list elements
pointing by the ptr[] array
            if (ptr[i] < n && arr[i][ptr[i]] > maxval) {
                maxval = arr[i][ptr[i]];
            }
        }

        // if any list exhaust we will not get any better
answer, so break the while loop
        if (flag)
            break;

        ptr[minind]++;

        // updating the minrange
        if ((maxval - minval) < minrange) {
            minel = minval;
            maxel = maxval;
            minrange = maxel - minel;
        }
    }

    printf("The smallest range is [%d, %d]\n", minel, maxel);
}

// Driver program to test above function
int main()
{
    int arr[][N] = {
        { 4, 7, 9, 12, 15 },
        { 0, 8, 10, 14, 20 },
        { 6, 12, 16, 30, 50 }
    };

    int k = sizeof(arr) / sizeof(arr[0]);

    findSmallestRange(arr, N, k);

    return 0;
}
// This code is contributed by Aditya Krishna Namdeo
```

- **Output:**

```
The smallest range is [6 8]
```

- Complexity Analysis:
  1. Time complexity : O(n * k2), to find the maximum and minimum in an array of length k the time required is O(k), and to traverse all the k arrays of length n (in worst case), the time complexity is O(n*k), then the total time complexity is O(n*k2).
  2. Space complexity: O(k), an extra array is required of length k so the space complexity is O(k)

Efficient approach: The approach remains the same but the time complexity can be reduced by using min-heap or *priority queue*. Min heap can be used to find the maximum and minimum value in logarithmic time or log k time instead of linear time. Rest of the approach remains the same.

- Algorithm:

1. create an Min heap to store k elements, one from each array and a variable *minrange* initilized to a maximum value and also keep a variable *max* to store the maximum integer.
2. Initially put the first element of each element from each list and store the maximum value in *max*.
3. Repeat the following steps until atleast one list exhausts :
    1. To find the minimum value or *min*, use the top or root of the Min heap which is the minimum element.
    2. Now update the minrange if current (max-min) is less than minrange.
    3. remove the top or root element from priority queue and insert the next element from the list which contains the min element and upadate the max with the new element inserted.

- Implementation:

```
// C++ program to finds out smallest range that includes
// elements from each of the given sorted lists.
#include <bits/stdc++.h>
using namespace std;

#define N 5

// A min heap node
struct MinHeapNode {
    // The element to be stored
    int element;

    // index of the list from which the element is taken
    int i;

    // index of the next element to be picked from list
    int j;
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode* x, MinHeapNode* y);

// A class for Min Heap
class MinHeap {

    // pointer to array of elements in heap
    MinHeapNode* harr;
```

```cpp
    // size of min heap
    int heap_size;

public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int);

    // to get index of left child of node at index i
    int left(int i) { return (2 * i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2 * i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x)
    {
        harr[0] = x;
        MinHeapify(0);
    }
};

// Constructor: Builds a heap from a
// given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1) / 2;
    while (i >= 0) {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at
// given index. This method assumes that the subtrees
// are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
```

```
        smallest = l;
    if (r < heap_size && harr[r].element < 
harr[smallest].element) 
        smallest = r;
    if (smallest != i) {
        swap(harr[i], harr[smallest]);
        MinHeapify(smallest);
    }
}

// This function takes an k sorted lists in the form of
// 2D array as an argument. It finds out smallest range
// that includes elements from each of the k lists.
void findSmallestRange(int arr[][N], int k)
{
    // Create a min heap with k heap nodes. Every heap node
    // has first element of an list
    int range = INT_MAX;
    int min = INT_MAX, max = INT_MIN;
    int start, end;

    MinHeapNode* harr = new MinHeapNode[k];
    for (int i = 0; i < k; i++) {
        // Store the first element
        harr[i].element = arr[i][0];

        // index of list
        harr[i].i = i;

        // Index of next element to be stored
        // from list
        harr[i].j = 1;

        // store max element
        if (harr[i].element > max)
            max = harr[i].element;
    }

    // Create the heap
    MinHeap hp(harr, k);

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its list
    while (1) {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();

        // update min
        min = hp.getMin().element;
```

```

            // update range
            if (range > max - min + 1) {
                range = max - min + 1;
                start = min;
                end = max;
            }

            // Find the next element that will replace current
            // root of heap. The next element belongs to same
            // list as the current root.
            if (root.j < N) {
                root.element = arr[root.i][root.j];
                root.j += 1;

                // update max element
                if (root.element > max)
                    max = root.element;
            }

            // break if we have reached end of any list
            else
                break;

            // Replace root with next element of list
            hp.replaceMin(root);
        }

        cout << "The smallest range is "
             << "["
             << start << " " << end << "]" << endl;
        ;
}

// Driver program to test above functions
int main()
{
    int arr[][N] = {
        { 4, 7, 9, 12, 15 },
        { 0, 8, 10, 14, 20 },
        { 6, 12, 16, 30, 50 }
    };
    int k = sizeof(arr) / sizeof(arr[0]);

    findSmallestRange(arr, k);

    return 0;
}
```

- **Output:**

```
The smallest range is [6 8]
```

- Complexity Analysis:
    1. Time complexity : O(n * k *log k).

       To find the maximum and minimum in an Min Heap of length k the time required is O(log k), and to traverse all the k arrays of length n (in worst case), the time complexity is O(n*k), then the total time complexity is O(n * k *log k).
    2. Space complexity: O(k).

       The priority queue will store k elements so the space complexity os O(k

8. Convert min Heap to max Heap

Given array representation of min Heap, convert it to max Heap in O(n) time.

Example :

```
Input: arr[] = [3 5 9 6 8 20 10 12 18 9]

        3

     /     \

    5       9

  /   \    /  \

 6     8  20   10

 /  \   /

12   18 9



Output: arr[] = [20 18 10 12 9 9 3 5 6 8] OR

      [any Max Heap formed from input elements]


        20

      /    \

    18      10

  /    \    /  \

 12     9  9    3

 /  \   /

5    6 8
```

The problem might look complex at first look. But our final goal is to only build the max heap. The idea is very simple – we simply build Max Heap without caring about the input. We start from bottom-most and rightmost internal mode of min Heap and heapify all internal modes in bottom up way to build the Max heap.

Below is its implementation

```cpp
// A C++ program to convert min Heap to max Heap
#include<bits/stdc++.h>
using namespace std;

// to heapify a subtree with root at given index
void MaxHeapify(int arr[], int i, int n)
{
    int l = 2*i + 1;
    int r = 2*i + 2;
    int largest = i;
    if (l < n && arr[l] > arr[i])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        MaxHeapify(arr, largest, n);
    }
}

// This function basically builds max heap
void convertMaxHeap(int arr[], int n)
{
    // Start from bottommost and rightmost
    // internal mode and heapify all internal
    // modes in bottom up way
    for (int i = (n-2)/2; i >= 0; --i)
        MaxHeapify(arr, i, n);
}

// A utility function to print a given array
// of given size
void printArray(int* arr, int size)
{
    for (int i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

// Driver program to test above functions
int main()
```

```
{
    // array representing Min Heap
    int arr[] = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Min Heap array : ");
    printArray(arr, n);

    convertMaxHeap(arr, n);

    printf("\nMax Heap array : ");
    printArray(arr, n);

    return 0;
}
```

Output :

```
Min Heap array : 3 5 9 6 8 20 10 12 18 9

Max Heap array : 20 18 10 12 9 9 3 5 6 8
```

The complexity of above solution might looks like O(nLogn) but it is O(n). Refer this G-Fact for more details.

9. Count pairs (a, b) whose sum of cubes is N (a^3 + b^3 = N)

Given N, count all ‘a’ and ‘b’ that satisfy the condition a^3 + b^3 = N.

Examples:

```
Input : N = 9

Output : 2

1^3 + 2^3 = 9

2^3 + 1^3 = 9


Input : N = 28

Output : 2

 1^3 + 3^3 = 28

 3^3 + 1^3 = 28
```

Note:- (a, b) and (b, a) are to be considered as two different pairs.

Asked in : Adobe

Implementation:

```
Travers numbers from 1 to cube root of N.

  a) Subtract cube of current number from

    N and check if their difference is a

    perfect cube or not.

      i) If perfect cube then increment count.


2- Return count.
```

Below is the implementation of above approach:

```
// C++ program to count pairs whose sum
// cubes is N
#include<bits/stdc++.h>
using namespace std;

// Function to count the pairs satisfying
// a ^ 3 + b ^ 3 = N
int countPairs(int N)
{
    int count = 0;

    // Check for each number 1 to cbrt(N)
    for (int i = 1; i <= cbrt(N); i++)
    {
        // Store cube of a number
        int cb = i*i*i;

        // Subtract the cube from given N
        int diff = N - cb;

        // Check if the difference is also
        // a perfect cube
        int cbrtDiff = cbrt(diff);

        // If yes, then increment count
        if (cbrtDiff*cbrtDiff*cbrtDiff == diff)
            count++;
    }

    // Return count
    return count;
}

// Driver program
int main()
{
    // Loop to Count no. of pairs satisfying
    // a ^ 3 + b ^ 3 = i for N = 1 to 10
    for (int i = 1; i<= 10; i++)
        cout << "For n = " << i << ", "
             << countPairs(i) <<" pair exists\n";

    return 0;
}
```

Output:

```
For n= 1, 1 pair exists

For n= 2, 1 pair exists

For n= 3, 0 pair exists
For n= 4, 0 pair exists

For n= 5, 0 pair exists

For n= 6, 0 pair exists

For n= 7, 0 pair exists

For n= 8, 1 pair exists

For n= 9, 2 pair exists

For n= 10, 0 pair exists
```

10. Convert BST to Max Heap

Given a Binary Search Tree which is also a Complete Binary Tree. The problem is to convert a given BST into a Special Max Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Max Heap.

Examples:

```
Input :          4

             /   \

           2     6

         /  \   /  \

        1   3  5    7


Output :        7

           /   \

         3     6

       /   \  /   \

     1    2 4     5
The given BST has been transformed into a
Max Heap.
All the nodes in the Max Heap satisfies the given

condition, that is, values in the left subtree of

a node should be less than the values in the right

subtree of the node.
```

Pre Requisites: Binary Seach Tree | Heaps

Approach

1. Create an array arr[] of size n, where n is the number of nodes in the given BST.

2. Perform the inorder traversal of the BST and copy the node values in the arr[] in sorted order.

3. Now perform the postorder traversal of the tree.

4. While traversing the root during the postorder traversal, one by one copy the values from the array arr[] to the nodes.

```
// C++ implementation to convert a given
// BST to Max Heap
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* getNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function prototype for postorder traversal
// of the given tree
void postorderTraversal(Node*);

// Function for the inorder traversal of the tree
// so as to store the node values in 'arr' in
// sorted order
void inorderTraversal(Node* root, vector<int>& arr)
{
    if (root == NULL)
```

```
        return;

    // first recur on left subtree
    inorderTraversal(root->left, arr);

    // then copy the data of the node
    arr.push_back(root->data);

    // now recur for right subtree
    inorderTraversal(root->right, arr);
}

void BSTToMaxHeap(Node* root, vector<int> arr, int* i)
{
    if (root == NULL)
        return;

    // recur on left subtree
    BSTToMaxHeap(root->left, arr, i);

    // recur on right subtree
    BSTToMaxHeap(root->right, arr, i);

    // copy data at index 'i' of 'arr' to
    // the node
    root->data = arr[++*i];
}

// Utility function to convert the given BST to
// MAX HEAP
void convertToMaxHeapUtil(Node* root)
{
    // vector to store the data of all the
    // nodes of the BST
    vector<int> arr;
    int i = -1;

    // inorder traversal to populate 'arr'
    inorderTraversal(root, arr);

    // BST to MAX HEAP conversion
    BSTToMaxHeap(root, arr, &i);
}

// Function to Print Postorder Traversal of the tree
void postorderTraversal(Node* root)
{
    if (!root)
        return;
    // recur on left subtree
    postorderTraversal(root->left);

    // then recur on right subtree
    postorderTraversal(root->right);

    // print the root's data
    cout << root->data << " ";
}

// Driver Code
int main()
{
    // BST formation
    struct Node* root = getNode(4);
    root->left = getNode(2);
    root->right = getNode(6);
    root->left->left = getNode(1);
    root->left->right = getNode(3);
    root->right->left = getNode(5);
    root->right->right = getNode(7);

    convertToMaxHeapUtil(root);
    cout << "Postorder Traversal of Tree:" << endl;
    postorderTraversal(root);

    return 0;
}
```

Output:

```
Postorder Traversal of Tree:

1 2 3 4 5 6 7
```

Time Complexity: O(n)

Auxiliary Space: O(n)

where, n is the number of nodes in the tree

11. Convert BST to Min Heap

Given a binary search tree which is also a complete binary tree. The problem is to convert the given BST into a Min Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Min Heap.

Examples:

```
Input :          4

             /   \

            2     6

          /  \   /  \

         1   3  5    7


Output :        1

            /   \

           2     5

         /  \   /  \

        3   4  6    7


The given BST has been transformed into a
Min Heap.
All the nodes in the Min Heap satisfies the given

condition, that is, values in the left subtree of

a node should be less than the values in the right

subtree of the node.
```

1. Create an array **arr[]** of size **n**, where n is the number of nodes in the given BST.
2. Perform the inorder traversal of the BST and copy the node values in the **arr[]** in sorted order.
3. Now perform the preorder traversal of the tree.
4. While traversing the root during the preorder traversal, one by one copy the values from the array **arr[]** to the nodes.

```
// C++ implementation to convert the given
// BST to Min Heap
#include <bits/stdc++.h>
using namespace std;

// structure of a node of BST
struct Node
{
    int data;
    Node *left, *right;
};

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers. */
struct Node* getNode(int data)
{
    struct Node *newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// function prototype for preorder traversal
// of the given tree
void preorderTraversal(Node*);

// function for the inorder traversal of the tree
// so as to store the node values in 'arr' in
// sorted order
void inorderTraversal(Node *root, vector<int>& arr)
{
    if (root == NULL)
        return;

    // first recur on left subtree
    inorderTraversal(root->left, arr);

    // then copy the data of the node
    arr.push_back(root->data);

    // now recur for right subtree
    inorderTraversal(root->right, arr);
}

// function to convert the given BST to MIN HEAP
// performs preorder traversal of the tree
void BSTToMinHeap(Node *root, vector<int> arr, int *i)
{
    if (root == NULL)
        return;

    // first copy data at index 'i' of 'arr' to
    // the node
    root->data = arr[++*i];

    // then recur on left subtree
    BSTToMinHeap(root->left, arr, i);

    // now recur on right subtree
    BSTToMinHeap(root->right, arr, i);
}

// utility function to convert the given BST to
// MIN HEAP
void convertToMinHeapUtil(Node *root)
{
    // vector to store the data of all the
    // nodes of the BST
    vector<int> arr;
    int i = -1;

    // inorder traversal to populate 'arr'
    inorderTraversal(root, arr);

    // BST to MIN HEAP conversion
    BSTToMinHeap(root, arr, &i);
}

// function for the preorder traversal of the tree
void preorderTraversal(Node *root)
{
    if (!root)
        return;

    // first print the root's data
    cout << root->data << " ";

    // then recur on left subtree
    preorderTraversal(root->left);

    // now recur on right subtree
    preorderTraversal(root->right);
}

// Driver program to test above
int main()
{
    // BST formation
    struct Node *root = getNode(4);
    root->left = getNode(2);
    root->right = getNode(6);
    root->left->left = getNode(1);
    root->left->right = getNode(3);
    root->right->left = getNode(5);
    root->right->right = getNode(7);

    convertToMinHeapUtil(root);
    cout << "Preorder Traversal:" << endl;
    preorderTraversal(root);

    return 0;
}
```

Output:

```
Preorder Traversal:

1 2 3 4 5 6 7
```

Time Complexity: O(n)

Auxiliary Space: O(n)

12. K’th largest element in a stream

Given an infinite stream of integers, find the k’th largest element at any point of time.

Example:

```
Input:

stream[] = {10, 20, 11, 70, 50, 40, 100, 5, ...}

k = 3


Output:    {_,   _, 10, 11, 20, 40, 50,  50, ...}
```

Extra space allowed is O(k).

We have discussed different approaches to find k’th largest element in an array in the following posts.

K’th Smallest/Largest Element in Unsorted Array | Set 1

K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

K’th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

Here we have a stream instead of whole array and we are allowed to store only k elements.

A Simple Solution is to keep an array of size k. The idea is to keep the array sorted so that the k’th largest element can be found in O(1) time (we just need to return first element of array if array is sorted in increasing order)

How to process a new element of stream?

For every new element in stream, check if the new element is smaller than current k’th largest element. If yes, then ignore it. If no, then remove the smallest element from array and insert new element in sorted order. Time complexity of processing a new element is O(k).

A Better Solution is to use a Self Balancing Binary Search Tree of size k. The k’th largest element can be found in O(Logk) time.

How to process a new element of stream?

For every new element in stream, check if the new element is smaller than current k’th largest element. If yes, then ignore it. If no, then remove the smallest element from the tree and insert new element. Time complexity of processing a new element is O(Logk).

An Efficient Solution is to use Min Heap of size k to store k largest elements of stream. The k’th largest element is always at root and can be found in O(1) time.

How to process a new element of stream?

Compare the new element with root of heap. If new element is smaller, then ignore it. Otherwise replace root with new element and call heapify for the root of modified heap. Time complexity of finding the k’th largest element is O(Logk).

```
// A C++ program to find k'th smallest element in a stream
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void buildHeap();
    void MinHeapify(int i);  //To minheapify subtree rooted with
index i
    int parent(int i)  { return (i-1)/2;  }
    int left(int i)    { return (2*i + 1);  }
    int right(int i)   { return (2*i + 2);  }
    int extractMin();  // extracts root (minimum) element
    int getMin()       {  return harr[0]; }
    // to replace root with new node x and heapify() new root
    void replaceMin(int x) { harr[0] = x; MinHeapify(0); }
};

MinHeap::MinHeap(int a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
}

void MinHeap::buildHeap()
{
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::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 MinHeap::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);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element from input stream
void kthLargest(int k)
{
    // count is total no. of elements in stream seen so far
    int count = 0, x;  // x is for new element

    // Create a min heap of size k
    int *arr = new int[k];
    MinHeap mh(arr, k);


    while (1)
    {
        // Take next element from stream
        cout << "Enter next element of stream ";
        cin >> x;

        // Nothing much to do for first k-1 elements
        if (count < k-1)
        {
            arr[count] = x;
            count++;
        }

        else
        {
          // If this is k'th element, then store it
          // and build the heap created above
          if (count == k-1)
          {
              arr[count] = x;
              mh.buildHeap();
          }

          else
          {
               // If next element is greater than
               // k'th largest, then replace the root
               if (x > mh.getMin())
                  mh.replaceMin(x); // replaceMin calls
                                    // heapify()
          }

          // Root of heap is k'th largest element
          cout << "K'th largest element is "
               << mh.getMin() << endl;
          count++;
        }
    }
}

// Driver program to test above methods
int main()
{
    int k = 3;
    cout << "K is " << k << endl;
    kthLargest(k);
    return 0;
}
```
Output

```
K is 3

Enter next element of stream 23

Enter next element of stream 10

Enter next element of stream 15

K'th largest element is 10

Enter next element of stream 70

K'th largest element is 15

Enter next element of stream 5

K'th largest element is 15

Enter next element of stream 80
```

```
K'th largest element is 23

Enter next element of stream 100

K'th largest element is 70

Enter next element of stream

CTRL + C pressed
```

13. K’th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k’th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

> *Input: arr[] = {7, 10, 4, 3, 20, 15}*
>
> *k = 3*
>
> *Output: 7*
>
> *Input: arr[] = {7, 10, 4, 3, 20, 15}*
>
> *k = 4*
>
> *Output: 10*

We have discussed a similar problem to print k largest elements.

Method 1 (Simple Solution)

A simple solution is to sort the given array using a O(N log N) sorting algorithm like Merge Sort, Heap Sort, etc and return the element at index k-1 in the sorted array.

Time Complexity of this solution is O(N Log N)

```
// Simple C++ program to find k'th smallest element
#include <algorithm>
#include <iostream>
using namespace std;

// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Sort the given array
    sort(arr, arr + n);
    // Return k'th element in the sorted array
    return arr[k - 1];
}

// Driver program to test above methods
int main()
{
    int arr[] = { 12, 3, 5, 7, 19 };
    int n = sizeof(arr) / sizeof(arr[0]), k = 2;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Method 2 (Using Min Heap – HeapSelect)

We can find k’th smallest element in time complexity better than O(N Log N). A simple optomization is to create a Min Heap of the given n elements and call extractMin() k times.

The following is C++ implementation of above method.

```
// A C++ program to find k'th smallest element using min heap
#include <climits>
#include <iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int* x, int* y);

// A class for Min Heap
class MinHeap {
    int* harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MinHeap(int a[], int size); // Constructor
    void MinHeapify(int i); // To minheapify subtree rooted with
index i
    int parent(int i) { return (i - 1) / 2; }
    int left(int i) { return (2 * i + 1); }
    int right(int i) { return (2 * i + 2); }
    int extractMin(); // extracts root (minimum) element
    int getMin() { return harr[0]; } // Returns minimum
};

MinHeap::MinHeap(int a[], int 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 MinHeap::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 MinHeap::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);
    }
}

// A utility function to swap two elements
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th smallest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of n elements: O(n) time
    MinHeap mh(arr, n);

    // Do extract min (k-1) times
    for (int i = 0; i < k - 1; i++)
        mh.extractMin();

    // Return root
    return mh.getMin();
}

// Driver program to test above methods
int main()
{
    int arr[] = { 12, 3, 5, 7, 19 };
    int n = sizeof(arr) / sizeof(arr[0]), k = 2;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time complexity of this solution is O(n + kLogn).

Method 3 (Using Max-Heap)

We can also use Max Heap for finding the k'th smallest element. Following is algorithm.

1) Build a Max-Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the k’th element (arr[k] to arr[n-1]), compare it with root of MH.

……a) If the element is less than the root then make it root and call heapify for MH

……b) Else ignore it.

// The step 2 is O((n-k)*logk)

3) Finally, root of the MH is the kth smallest element.

Time complexity of this solution is O(k + (n-k)*Logk)

The following is C++ implementation of above algorithm

```cpp
// A C++ program to find k'th smallest element using max heap
#include <climits>
#include <iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int* x, int* y);

// A class for Max Heap
class MaxHeap {
    int* harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of max heap
    int heap_size; // Current number of elements in max heap
public:
    MaxHeap(int a[], int size); // Constructor
    void maxHeapify(int i); // To maxHeapify subtree rooted with
index i
    int parent(int i) { return (i - 1) / 2; }
    int left(int i) { return (2 * i + 1); }
    int right(int i) { return (2 * i + 2); }

    int extractMax(); // extracts root (maximum) element
    int getMax() { return harr[0]; } // Returns maximum

    // to replace root with new node x and heapify() new root
    void replaceMax(int x)
    {
        harr[0] = x;
        maxHeapify(0);
    }
};
```

```
MaxHeap::MaxHeap(int a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1) / 2;
    while (i >= 0) {
        maxHeapify(i);
        i--;
    }
}

// Method to remove maximum element (or root) from max heap
int MaxHeap::extractMax()
{
    if (heap_size == 0)
        return INT_MAX;

    // Store the maximum 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];
        maxHeapify(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 MaxHeap::maxHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int largest = i;
    if (l < heap_size && harr[l] > harr[i])
        largest = l;
    if (r < heap_size && harr[r] > harr[largest])
        largest = r;
    if (largest != i) {
        swap(&harr[i], &harr[largest]);
        maxHeapify(largest);
    }
}
```

```
// A utility function to swap two elements
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Function to return k'th largest element in a given array
int kthSmallest(int arr[], int n, int k)
{
    // Build a heap of first k elements: O(k) time
    MaxHeap mh(arr, k);

    // Process remaining n-k elements.  If current element is
    // smaller than root, replace root with current element
    for (int i = k; i < n; i++)
        if (arr[i] < mh.getMax())
            mh.replaceMax(arr[i]);

    // Return root
    return mh.getMax();
}

// Driver program to test above methods
int main()
{
    int arr[] = { 12, 3, 5, 7, 19 };
    int n = sizeof(arr) / sizeof(arr[0]), k = 4;
    cout << "K'th smallest element is " << kthSmallest(arr, n, k);
    return 0;
}
```

Output:

```
K'th smallest element is 12
```

Method 4 (QuickSelect)

This is an optimization over method 1 if QuickSort is used as a sorting algorithm in first step. In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the array around it. The idea is, not to do complete quicksort, but stop at the point where pivot itself is k'th smallest element. Also, not to recur for both left and right sides of pivot, but recur for one of them according to the position of pivot. The worst case time complexity of this method is O(n2), but it works in O(n) on average.

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

int partition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE
DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1) {
        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);

        // If position is same as k
        if (pos - l == k - 1)
            return arr[pos];
        if (pos - l > k - 1) // If position is more, recur for
left subarray
            return kthSmallest(arr, l, pos - 1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort().  It considers the
last
// element as pivot and moves all smaller element to left of it
// and greater elements to right
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++) {
        if (arr[j] <= x) {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = { 12, 3, 5, 7, 4, 19, 26 };
    int n = sizeof(arr) / sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n -
1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

There are two more solutions which are better than above discussed ones: One solution is to do randomized version of quickSelect() and other solution is worst case linear time algorithm

14. K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend reading the following post as a prerequisite of this post.

K’th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than the size of the array, we need to find the k’th smallest element in the given array. It is given that all array elements are distinct.

Examples:

```
Input: arr[] = {7, 10, 4, 3, 20, 15}

      k = 3

Output: 7


Input: arr[] = {7, 10, 4, 3, 20, 15}

      k = 4

Output: 10
```

We have discussed three different solutions here.

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the previous post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, rand() to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is an implementation of the above Randomized QuickSelect.

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
```

```
using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE
DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than the number of elements in the array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort().  It considers the
last
// element as pivot and moves all smaller element to left of it
and
// greater elements to right. This function is used by
randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-
1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time Complexity:

The worst case time complexity of the above solution is still O(n2). In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is O(n), see CLRS book or MIT video lecture for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range

15. K’th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend reading following posts as a prerequisite of this post.

K’th Smallest/Largest Element in Unsorted Array | Set 1

K’th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

Given an array and a number k where k is smaller than the size of the array, we need to find the k’th smallest element in the given array. It is given that all array elements are distinct.

Examples:

```
Input: arr[] = {7, 10, 4, 3, 20, 15}

      k = 3

Output: 7


Input: arr[] = {7, 10, 4, 3, 20, 15}

      k = 4

Output: 10
```

In previous post, we discussed an expected linear time algorithm. In this post, a worst-case linear time method is discussed. The idea in this new method is similar to quickSelect(), we get worst-case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on another side). After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of the pivot.

Following is complete algorithm.

> *kthSmallest(arr[0..n-1], k)*
>
> *1) Divide arr[] into ⌈n/5⌉ groups where size of each group is 5 except possibly the last group which may have less than 5 elements.*

```
2) Sort the above created [n/5] groups and find median of all groups.
Create an auxiliary array 'median[]' and store medians of all [n/5]
groups in this median array.
// Recursively call this method to find median of median[0..[n/5]-1]
3) medOfMed = kthSmallest(median[0..[n/5]-1], [n/10])
4) Partition arr[] around medOfMed and obtain its position.
pos = partition(arr, n, medOfMed)
5) If pos == k return medOfMed
6) If pos > k return kthSmallest(arr[l..pos-1], k)
7) If pos < k return kthSmallest(arr[pos+1..r], k-pos+l-1)
```

In above algorithm, last 3 steps are same as algorithm in previous post. The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

Following is the implementation of above algorithm.

```cpp
// C++ implementation of worst case linear time algorithm
// to find k'th smallest element
#include<iostream>
#include<algorithm>
#include<climits>

using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[].  This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n);  // Sort the array
    return arr[n/2];   // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
        // of every group and store it in median[] array.
        int i, median[(n+4)/5]; // There will be floor((n+4)/5)
groups;
        for (i=0; i<n/5; i++)
            median[i] = findMedian(arr+l+i*5, 5);
        if (i*5 < n) //For last group with less than 5 elements
        {
            median[i] = findMedian(arr+l+i*5, n%5);
            i++;
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
                                 kthSmallest(median, 0, i-1, i/2);

        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, medOfMed);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is "
         << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

Time Complexity:

The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.

The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.

The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.

The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.

How many elements are greater than medOfMed and how many are smaller? At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Similarly, the number of elements that are less than medOfMed is at least 3n/10 – 6. In the worst case, the function recurs for at most n – (3n/10 – 6) which is 7n/10 + 6 elements.

Note that 7n/10 + 6 20 20 and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 80 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + O(n) & \text{if } n > 90 \end{cases}$$

We show that the running time is linear by substitution. Assume that T(n) cn for some constant c and all n > 80. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

```
T(n)  <= cn/5 + c(7n/10 + 6) + O(n)

     <= cn/5 + c + 7cn/10 + 6c + O(n)

    <= 9cn/10 + 7c + O(n)

    <= cn,
```

since we can pick c large enough so that c(n/10 – 7) is larger than the function described by the O(n) term for all n > 80. The worst-case running time of is

therefore linear
(Source: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm).
Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. Therefore, this algorithm doesn’t work well in practical situations, randomized quickSelect works much better and preferred.

16. Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

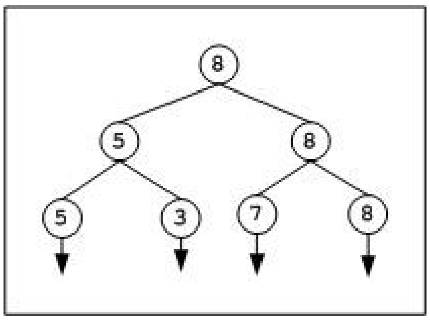
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in (N + $\log_2 N$ – 2) comparisons.

The following diagram displays a tournament tree (winner tree) as a max heap. Note that the concept of loser tree is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

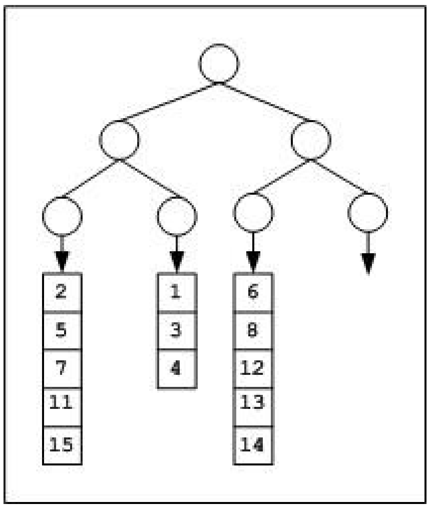
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height CEIL ($\log_2 M$) to have atleast M external nodes.

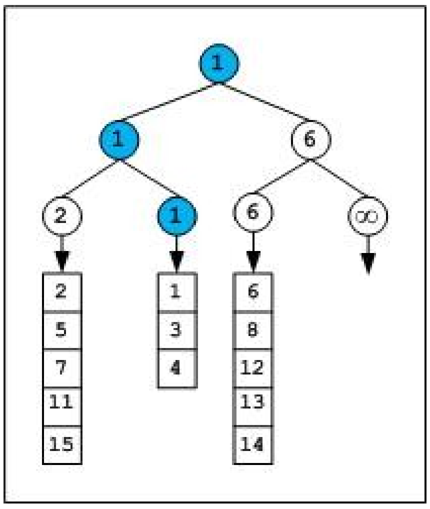
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1

{1, 3, 4} ---- Array2

{6, 8, 12, 13, 14} ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3$.= 1.585 = 2 rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



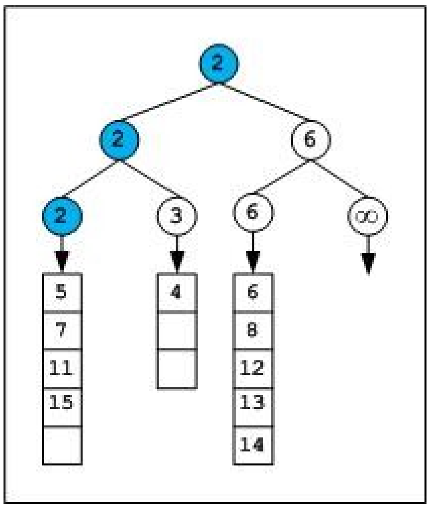
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of $O((L_1 + L_2 + … + L_m) * \log M)$ to merge all the arrays, and $O(m*\log M)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files.

Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

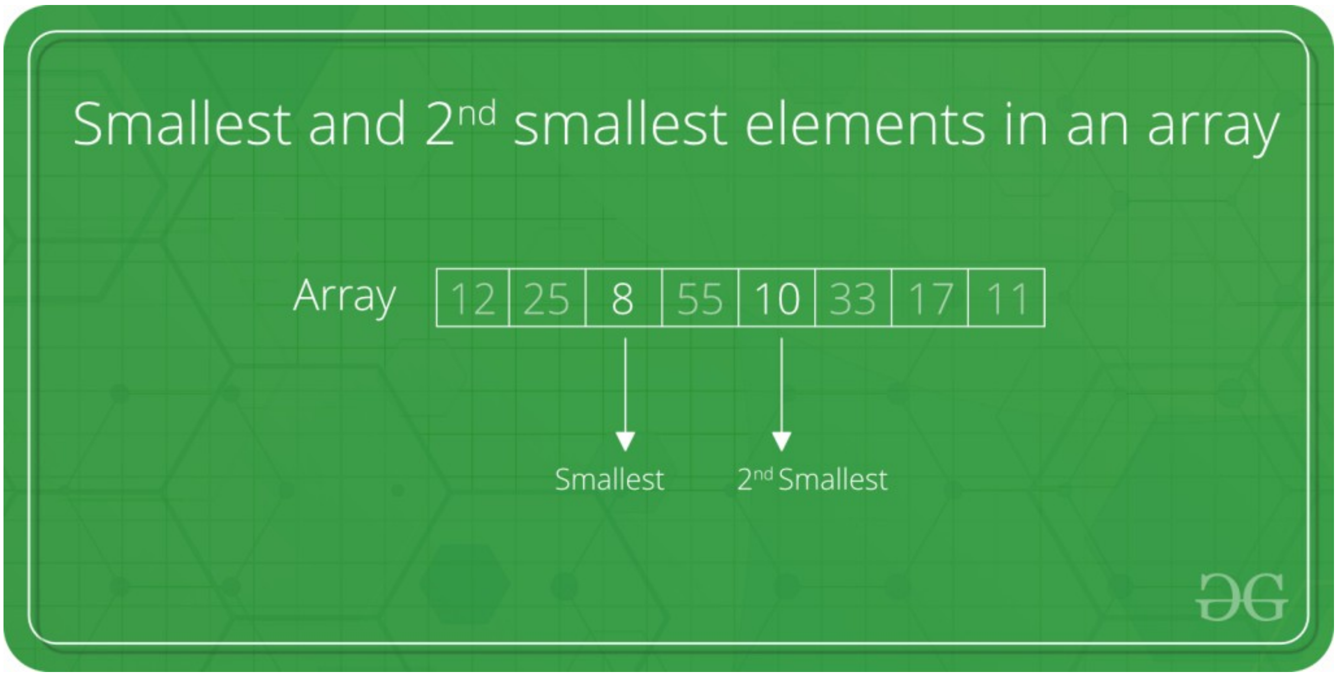
Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to 2^k – 1 where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. Implementation is discussed in below code

17. Find the smallest and second smallest elements in an array

Write an efficient C program to find smallest and second smallest element in an array.



Example:

```
Input:  arr[] = {12, 13, 1, 10, 34, 1}

Output: The smallest element is 1 and

       second Smallest element is 10
```

A Simple Solution is to sort the array in increasing order. The first two elements in sorted array would be two smallest elements. Time complexity of this solution is O(n Log n).

A Better Solution is to scan the array twice. In first traversal find the minimum element. Let this element be x. In second traversal, find the smallest element greater than x. Time complexity of this solution is O(n).

The above solution requires two traversals of input array.

An Efficient Solution can find the minimum two elements in one traversal.

Below is complete algorithm.

Algorithm:

```
1) Initialize both first and second smallest as INT_MAX

   first = second = INT_MAX

2) Loop through all the elements.

   a) If the current element is smaller than first, then update first

       and second.

   b) Else if the current element is smaller than second then update

    second
```

Implementation:

```
// C++ program to find smallest and
// second smallest elements
#include <bits/stdc++.h>
using namespace std; /* For INT_MAX */

void print2Smallest(int arr[], int arr_size)
{
    int i, first, second;

    /* There should be atleast two elements */
    if (arr_size < 2)
    {
        cout<<" Invalid Input ";
        return;
    }

    first = second = INT_MAX;
    for (i = 0; i < arr_size ; i ++)
    {
        /* If current element is smaller than first
        then update both first and second */
        if (arr[i] < first)
        {
            second = first;
            first = arr[i];
        }

        /* If arr[i] is in between first and second
        then update second */
        else if (arr[i] < second && arr[i] != first)
            second = arr[i];
    }
    if (second == INT_MAX)
        cout << "There is no second smallest element\n";
    else
        cout << "The smallest element is " << first << " and
second "
            "Smallest element is " << second << endl;
}

/* Driver code */
int main()
{
    int arr[] = {12, 13, 1, 10, 34, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    print2Smallest(arr, n);
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output :

```
The smallest element is 1 and second Smallest element is 10
```

The same approach can be used to find the largest and second largest elements in an array.

Time Complexity: O(n)

18. Second minimum element using minimum comparisons

Given an array of integers, find the minimum (or maximum) element and the element just greater (or smaller) than that in less than 2n comparisons. The given array is not necessarily sorted. Extra space is allowed.

Examples:

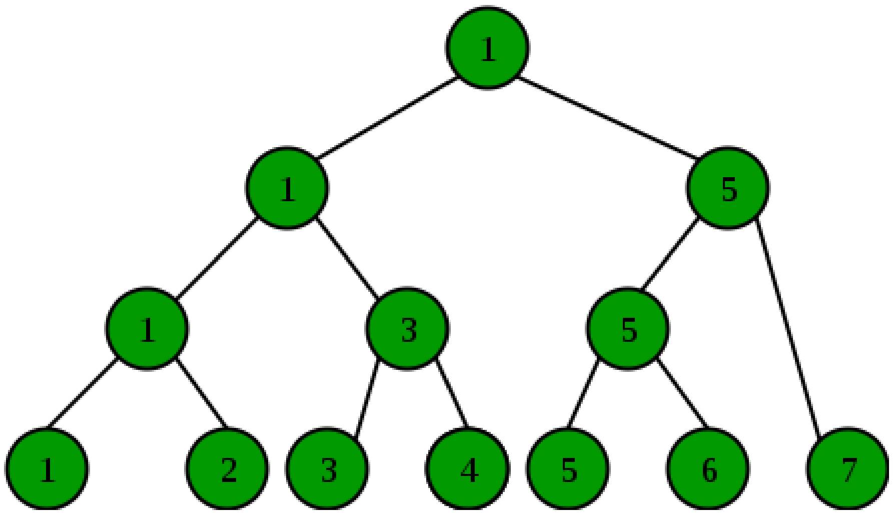
```
Input: {3, 6, 100, 9, 10, 12, 7, -1, 10}

Output: Minimum: -1, Second minimum: 3
```

We have already discussed an approach in below post.

Find the smallest and second smallest element in an array

Comparisons of array elements can be costly if array elements are of large size, for example large strings. We can minimize the number of comparisons used in above approach.

The idea is based on tournament tree. Consider each element in the given array as a leaf node.



First, we find the minimum element as explained by building a tournament tee. To build the tree, we compare all adjacent pairs of elements (leaf nodes) with each other. Now we have n/2 elements which are smaller than their counterparts (from each pair, the smaller element forms the level above that of the leaves). Again, find the smaller elements in each of the n/4 pairs. Continue this process until the root of the tree is created. The root is the minimum. Next, we traverse the tree and while doing so, we discard the sub trees whose root is greater than the smallest element. Before discarding, we update the second smallest element, which will hold our result. The point to understand here is that the tree is height balanced and we have to spend only logn time to traverse all the elements that were ever compared to the minimum in step 1. Thus, the overall time complexity is n + logn. Auxiliary space needed is O(2n), as the number of leaf nodes will be approximately equal to the number of internal nodes.

```
// C++ program to find minimum and second minimum
// using minimum number of comparisons
#include <bits/stdc++.h>
using namespace std;

// Tournament Tree node
struct Node
{
    int idx;
    Node *left, *right;
};

// Utility function to create a tournament tree node
Node *createNode(int idx)
{
    Node *t = new Node;
    t->left = t->right = NULL;
    t->idx = idx;
    return t;
}

// This function traverses tree across height to
// find second smallest element in tournament tree.
// Note that root is smallest element of tournament
// tree.
void traverseHeight(Node *root, int arr[], int &res)
{
    // Base case
    if (root == NULL || (root->left == NULL &&
                         root->right == NULL))
```

```
        return;

    // If left child is smaller than current result,
    // update result and recur for left subarray.
    if (res > arr[root->left->idx] &&
       root->left->idx != root->idx)
    {
        res = arr[root->left->idx];
        traverseHeight(root->right, arr, res);
    }

    // If right child is smaller than current result,
    // update result and recur for left subarray.
    else if (res > arr[root->right->idx] &&
             root->right->idx != root->idx)
    {
        res = arr[root->right->idx];
        traverseHeight(root->left, arr, res);
    }
}

// Prints minimum and second minimum in arr[0..n-1]
void findSecondMin(int arr[], int n)
{
    // Create a list to store nodes of current
    // level
    list<Node *> li;

    Node *root = NULL;
    for (int i = 0; i < n; i += 2)
    {
        Node *t1 = createNode(i);
        Node *t2 = NULL;
        if (i + 1 < n)
        {
            // Make a node for next element
            t2 = createNode(i + 1);

            // Make smaller of two as root
            root = (arr[i] < arr[i + 1])? createNode(i) :
                                           createNode(i + 1);

            // Make two nodes as children of smaller
            root->left = t1;
            root->right = t2;

            // Add root
            li.push_back(root);
        }
        else
            li.push_back(t1);
    }

    int lsize = li.size();

    // Construct the complete tournament tree from above
    // prepared list of winners in first round.
    while (lsize != 1)
    {
        // Find index of last pair
        int last = (lsize & 1)? (lsize - 2) : (lsize - 1);

        // Process current list items in pair
        for (int i = 0; i < last; i += 2)
        {
            // Extract two nodes from list, make a new
            // node for winner of two
            Node *f1 = li.front();
            li.pop_front();

            Node *f2 = li.front();
            li.pop_front();
            root = (arr[f1->idx] < arr[f2->idx])?
                createNode(f1->idx) : createNode(f2->idx);

            // Make winner as parent of two
            root->left = f1;
            root->right = f2;

            // Add winner to list of next level
            li.push_back(root);
        }
        if (lsize & 1)
        {
            li.push_back(li.front());
            li.pop_front();
        }
        lsize = li.size();
    }

    // Traverse tree from root to find second minimum
    // Note that minimum is already known and root of
    // tournament tree.
    int res = INT_MAX;
    traverseHeight(root, arr, res);
    cout << "Minimum: " << arr[root->idx]
        << ", Second minimum: " << res << endl;
}
```

```
// Driver code
int main()
{
    int arr[] = {61, 6, 100, 9, 10, 12, 17};
    int n = sizeof(arr)/sizeof(arr[0]);
    findSecondMin(arr, n);
    return 0;
}
```

Output:

```
Minimum: 6, Second minimum: 9
```

We can optimize above code by avoid creation of leaf nodes, as that information is stored in the array itself (and we know that the elements were compared in pairs).

19. Print all elements in sorted order from row and column wise sorted matrix

Given an n x n matrix, where every row and column is sorted in non-decreasing order. Print all elements of matrix in sorted order.

Example:

```
Input: mat[][]  =  { {10, 20, 30, 40},

                     {15, 25, 35, 45},

                     {27, 29, 37, 48},

                     {32, 33, 39, 50},

                   };


Output:

Elements of matrix in sorted order

10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

We can use Young Tableau to solve the above problem. The idea is to consider given 2D array as Young Tableau and call extract minimum O(N)

```
// A C++ program to Print all elements in sorted order from row
and
// column wise sorted matrix
#include<iostream>
#include<climits>
using namespace std;

#define INF INT_MAX
#define N 4

// A utility function to youngify a Young Tableau.  This is
different
// from standard youngify.  It assumes that the value at mat[0][0]
is
// infinite.
void youngify(int mat[][N], int i, int j)
{
    // Find the values at down and right sides of mat[i][j]
    int downVal  = (i+1 < N)? mat[i+1][j]: INF;
    int rightVal = (j+1 < N)? mat[i][j+1]: INF;

    // If mat[i][j] is the down right corner element, return
    if (downVal==INF && rightVal==INF)
        return;

    // Move the smaller of two values (downVal and rightVal) to
    // mat[i][j] and recur for smaller value
    if (downVal < rightVal)
    {
        mat[i][j] = downVal;
        mat[i+1][j] = INF;
        youngify(mat, i+1, j);
    }
    else
    {
        mat[i][j] = rightVal;
        mat[i][j+1] = INF;
        youngify(mat, i, j+1);
    }
}

// A utility function to extract minimum element from Young
tableau
int extractMin(int mat[][N])
{
    int ret = mat[0][0];
    mat[0][0] = INF;
    youngify(mat, 0, 0);
    return ret;
}

// This function uses extractMin() to print elements in sorted
order
void printSorted(int mat[][N])
{
   cout << "Elements of matrix in sorted order n";
   for (int i=0; i<N*N; i++)
     cout << extractMin(mat) << " ";
}

// driver program to test above function
int main()
{
  int mat[N][N] = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {27, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
  printSorted(mat);
  return 0;
}
```

Output:

```
Elements of matrix in sorted order

10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

Time complexity of extract minimum is O(N) and it is called O(N^2) times. Therefore the overall time complexity is O(N^3).

A better solution is to use the approach used for merging k sorted arrays. The idea is to use a Min Heap of size N which stores elements of first column. The do extract minimum. In extract minimum, replace the minimum element with the next element of the row from which the element is extracted. Time complexity of this solution is O(N^2LogN).

```
// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<climits>
using namespace std;

#define N 4

// A min heap node
struct MinHeapNode
{
    int element; // The element to be stored
    int i; // index of the row from which the element is taken
    int j; // index of the next element to be picked from row
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );
    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x;
MinHeapify(0); }
};

// This function prints elements of a given matrix in non-
decreasing
//  order. It assumes that ma[][] is sorted row wise sorted.
void printSorted(int mat[][N])
{
    // Create a min heap with k heap nodes.  Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[N];
    for (int i = 0; i < N; i++)
    {
        harr[i].element = mat[i][0]; // Store the first element
        harr[i].i = i;  // index of row
        harr[i].j = 1;  // Index of next element to be stored from
row
    }
    MinHeap hp(harr, N); // Create the min heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < N*N; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();

        cout << root.element << " ";

        // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < N)
        {
            root.element = mat[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element =  INT_MAX; //INT_MAX is for infinite

        // Replace root with next element of array
        hp.replaceMin(root);
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at given
index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x;  *x = *y;  *y = temp;
}

// driver program to test above function
int main()
```

```
{
  int mat[N][N] = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {27, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
  printSorted(mat);
  return 0;
}
```
Output:

```
10 15 20 25 27 29 30 32 33 35 37 39 40 45 48 50
```

Exercise:

Above solutions work for a square matrix. Extend the above solutions to work for an M*N rectangular matrix.

20. Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Examples:

```
Input : arr[] = {6, 5, 3, 2, 8, 10, 9}

          k = 3

Output : arr[] = {2, 3, 5, 6, 8, 9, 10}


Input : arr[] = {10, 9, 8, 7, 4, 70, 60, 50}

        k = 4

Output : arr[] = {4, 7, 8, 9, 10, 50, 60, 70}
```

We can use Insertion Sort to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```
/* Function to sort an array using insertion sort*/
void insertionSort(int A[], int size)
{
   int i, key, j;
   for (i = 1; i < size; i++)
   {
       key = A[i];
       j = i-1;

       /* Move elements of A[0..i-1], that are greater than key,
to one
          position ahead of their current position.
          This loop will run at most k times */
       while (j >= 0 && A[j] > key)
       {
           A[j+1] = A[j];
           j = j-1;
```

```
        }
        A[j+1] = key;
    }
}
```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be O(nk)
We can sort such arrays more efficiently with the help of Heap data structure.

Following is the detailed process that uses Heap.

1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k)

time (See this GFact)

2) One by one remove min element from heap, put it in result array, and add a

new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take Logk

time. So overall complexity will be O(k) + O((n-k)*logK)

```
// A STL based C++ program to sort a nearly sorted array.
#include <bits/stdc++.h>
using namespace std;

// Given an array of size n, where every element
// is k away from its target position, sorts the
// array in O(nLogk) time.
int sortK(int arr[], int n, int k)
{
    // Insert first k+1 items in a priority queue (or min heap)
    //(A O(k) operation). We assume, k < n.
    priority_queue<int, vector<int>, greater<int> > pq(arr, arr +
k + 1);

    // i is index for remaining elements in arr[] and index
    // is target index of for current minimum element in
    // Min Heapm 'hp'.
    int index = 0;
    for (int i = k + 1; i < n; i++) {
        arr[index++] = pq.top();
        pq.pop();
        pq.push(arr[i]);
    }

    while (pq.empty() == false) {
        arr[index++] = pq.top();
        pq.pop();
    }
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = { 2, 6, 3, 12, 56, 8 };
    int n = sizeof(arr) / sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array" << endl;
    printArray(arr, n);

    return 0;
}
```

Output:

```
Following is sorted array

2 3 6 8 12 56
```

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also use a Balanced Binary Search Tree instead of Heap to store K+1 elements. The insert and delete operations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap bassed method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn’t need extra space for left and right pointers.

21. Connect n ropes with minimum cost

There are given n ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. We need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.
1) First connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6 and 5.
2) Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
3) Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is 5 + 9 + 15 = 29. This is the optimized cost for connecting ropes. Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three strings of 3, 2 and 10), then connect 10 and 3 (we get two strings of 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.

If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This approach is similar to Huffman Coding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.
Following is complete algorithm for finding the minimum cost for connecting n ropes.
Let there be n ropes of lengths stored in an array len[0..n-1]
1) Create a min heap and insert all lengths into the min heap.
2) Do following while number of elements in min heap is not one.

……a) Extract the minimum and second minimum from min heap

……b) Add the above two extracted values and insert the added value to the min-heap.

……c) Maintain a variable for total cost and keep incrementing it by the sum of extracted values.

3) Return the value of this total cost.

Following is the implementation of above algorithm.

```
// C++ program for connecting n ropes with minimum cost
#include <bits/stdc++.h>

using namespace std;

// A Min Heap:  Collection of min heap nodes
struct MinHeap {
    unsigned size; // Current size of min heap
    unsigned capacity; // capacity of min heap
    int* harr; // Attay of minheap nodes
};

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap = new MinHeap;
    minHeap->size = 0; // current size is 0
    minHeap->capacity = capacity;
    minHeap->harr = new int[capacity];
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if (left < minHeap->size && minHeap->harr[left] < minHeap-
>harr[smallest])
        smallest = left;

    if (right < minHeap->size && minHeap->harr[right] < minHeap-
>harr[smallest])
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->harr[smallest], &minHeap-
>harr[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract minimum value node from heap
int extractMin(struct MinHeap* minHeap)
{
    int temp = minHeap->harr[0];
    minHeap->harr[0] = minHeap->harr[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, int val)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && (val < minHeap->harr[(i - 1) / 2])) {
        minHeap->harr[i] = minHeap->harr[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->harr[i] = val;
}

// A standard funvtion to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
```

```
}

// Creates a min heap of capacity equal to size and inserts all
values
// from len[] in it. Initially size of min heap is equal to
capacity
struct MinHeap* createAndBuildMinHeap(int len[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->harr[i] = len[i];
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that returns the minimum cost to connect n
ropes of
// lengths stored in len[0..n-1]
int minCost(int len[], int n)
{
    int cost = 0; // Initialize result

    // Create a min heap of capacity equal to n and put all ropes
in it
    struct MinHeap* minHeap = createAndBuildMinHeap(len, n);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {
        // Extract two minimum length ropes from min heap
        int min = extractMin(minHeap);
        int sec_min = extractMin(minHeap);

        cost += (min + sec_min); // Update total cost

        // Insert a new rope in min heap with length equal to sum
        // of two extracted minimum lengths
        insertMinHeap(minHeap, min + sec_min);
    }

    // Finally return total minimum cost for connecting all ropes
    return cost;
}

// Driver program to test above functions
int main()
{
    int len[] = { 4, 3, 2, 6 };
    int size = sizeof(len) / sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len,
size);
    return 0;
}
```

Output:

```
Total cost for connecting ropes is 29
```

Time Complexity: Time complexity of the algorithm is O(nLogn) assuming that we use a O(nLogn) sorting algorithm. Note that heap operations like insert and extract take O(Logn) time.

Algorithmic Paradigm: Greedy Algorithm

A simple implementation with STL in C++

Following is a simple implementation that uses priority_queue available in STL. Thanks to Pango89 for providing below code.

```
#include <bits/stdc++.h>

using namespace std;

int minCost(int arr[], int n)
{
    // Create a priority queue ( http://
www.cplusplus.com/reference/queue/priority_queue/ )
    // By default 'less' is used which is for decreasing order
    // and 'greater' is used for increasing order
    priority_queue<int, vector<int>, greater<int> > pq(arr, arr +
n);

    // Initialize result
    int res = 0;

    // While size of priority queue is more than 1
    while (pq.size() > 1) {
        // Extract shortest two ropes from pq
        int first = pq.top();
        pq.pop();
        int second = pq.top();
        pq.pop();

        // Connect the ropes: update result and
        // insert the new rope to pq
        res += first + second;
        pq.push(first + second);
    }

    return res;
}

// Driver program to test above function
int main()
{
    int len[] = { 4, 3, 2, 6 };
    int size = sizeof(len) / sizeof(len[0]);
    cout << "Total cost for connecting ropes is " << minCost(len,
size);
    return 0;
}
```

Output:

```
Total cost for connecting ropes is 29
```

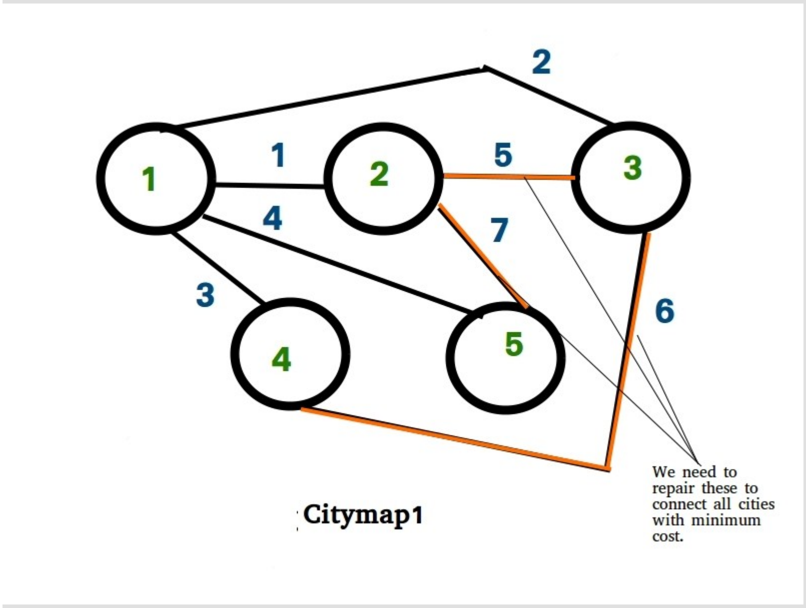
22. Minimum cost to connect all cities

There are n cities and there are roads in between some of the cities. Somehow all the roads are damaged simultaneously. We have to repair the roads to connect the cities again. There is a fixed cost to repair a particular road. Find out the minimum cost to connect all the cities by repairing roads. Input is in matrix(city) form, if city[i][j] = 0 then there is not any road between city i and city j, if city[i][j] = a > 0 then the cost to rebuild the path between city i and city j is a. Print out the minimum cost to connect all the cities.

It is sure that all the cities were connected before the roads were damaged.

Examples:

```
Input : {{0, 1, 2, 3, 4},

        {1, 0, 5, 0, 7},

        {2, 5, 0, 6, 0},

        {3, 0, 6, 0, 0},

        {4, 7, 0, 0, 0}};

Output : 10
```


2
1
5
4
7
3
6
1
2
3
4
5
We need to repair these to connect all cities with minimum cost.
Citymap1

```
Input : {{0, 1, 1, 100, 0, 0},
        {1, 0, 1, 0, 0, 0},

        {1, 1, 0, 0, 0, 0},

        {100, 0, 0, 0, 2, 2},

        {0, 0, 0, 2, 0, 2},

        {0, 0, 0, 2, 2, 0}};

Output : 106
```

Method: Here we have to connect all the cities by path which will cost us least. The way to do that is to find out the Minimum Spanning Tree(MST) of the map of the cities(i.e. each city is a node of the graph and all the damaged roads between cities are edges). And the total cost is the addition of the path edge values in the Minimum Spanning Tree.

Prerequisite: MST Prim's Algorithm

```
// C++ code to find out minimum cost
// path to connect all the cities
#include <bits/stdc++.h>

using namespace std;

// Function to find out minimum valued node
// among the nodes which are not yet included in MST
int minnode(int n, int keyval[], bool mstset[]) {
  int mini = numeric_limits<int>::max();
  int mini_index;

  // Loop through all the values of the nodes
  // which are not yet included in MST and find
  // the minimum valued one.
  for (int i = 0; i < n; i++) {
    if (mstset[i] == false && keyval[i] < mini) {
      mini = keyval[i], mini_index = i;
    }
  }
  return mini_index;
}

// Function to find out the MST and
// the cost of the MST.
void findcost(int n, vector<vector<int>> city) {
```

```cpp
// Array to store the parent node of a
// particular node.
int parent[n];

// Array to store key value of each node.
int keyval[n];

// Boolean Array to hold bool values whether
// a node is included in MST or not.
bool mstset[n];

// Set all the key values to infinite and
// none of the nodes is included in MST.
for (int i = 0; i < n; i++) {
  keyval[i] = numeric_limits<int>::max();
  mstset[i] = false;
}

// Start to find the MST from node 0.
// Parent of node 0 is none so set -1.
// key value or minimum cost to reach
// 0th node from 0th node is 0.
parent[0] = -1;
keyval[0] = 0;

// Find the rest n-1 nodes of MST.
for (int i = 0; i < n - 1; i++) {

  // First find out the minimum node
  // among the nodes which are not yet
  // included in MST.
  int u = minnode(n, keyval, mstset);

  // Now the uth node is included in MST.
  mstset[u] = true;

  // Update the values of neighbor
  // nodes of u which are not yet
  // included in MST.
  for (int v = 0; v < n; v++) {

    if (city[u][v] && mstset[v] == false &&
        city[u][v] < keyval[v]) {
      keyval[v] = city[u][v];
      parent[v] = u;
    }
  }
}
```

```
  // Find out the cost by adding
  // the edge values of MST.
  int cost = 0;
  for (int i = 1; i < n; i++)
    cost += city[parent[i]][i];
  cout << cost << endl;
}

// Utility Program:
int main() {

  // Input 1
  int n1 = 5;
  vector<vector<int>> city1 = {{0, 1, 2, 3, 4},
                               {1, 0, 5, 0, 7},
                               {2, 5, 0, 6, 0},
                               {3, 0, 6, 0, 0},
                               {4, 7, 0, 0, 0}};
  findcost(n1, city1);

  // Input 2
  int n2 = 6;
  vector<vector<int>> city2 = {{0, 1, 1, 100, 0, 0},
                               {1, 0, 1, 0, 0, 0},
                               {1, 1, 0, 0, 0, 0},
                               {100, 0, 0, 0, 2, 2},
                               {0, 0, 0, 2, 0, 2},
                               {0, 0, 0, 2, 2, 0}};
  findcost(n2, city2);

  return 0;
}
```
Output:
```
10

106
```

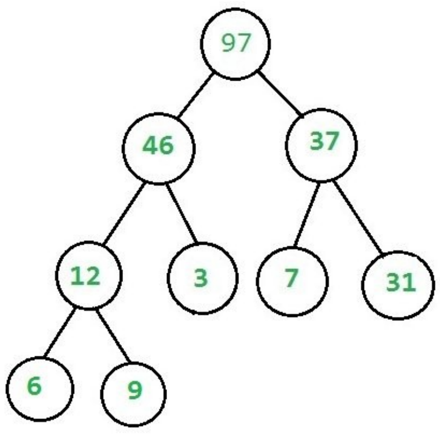
Complexity: The outer loop(i.e. the loop to add new node to MST) runs n times and in each iteration of the loop it takes O(n) time to find the minnode and O(n) time to update the neighboring nodes of u-th node. Hence the overall complexity is O(n_2)

23. Check if a given Binary Tree is Heap

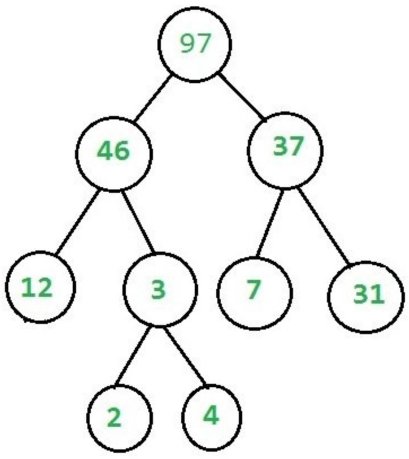
Given a binary tree, we need to check it has heap property or not, Binary tree need to fulfill the following two conditions for being a heap –

1. It should be a complete tree (i.e. all levels except last should be full).
2. Every node’s value should be greater than or equal to its child node (considering max-heap).

For example this tree contains heap property –



While this doesn’t –



We check each of the above condition separately, for checking completeness isComplete and for checking heap isHeapUtil function are written.

Detail about isComplete function can be found here.

isHeapUtil function is written considering the following things –

1. Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).
2. If Node has No child then it’s a leaf node and returns true (Base case)
3. If Node has one child (it must be left child because it is a complete tree) then we need to compare this node with its single child only.
4. If the Node has both child then check heap property at Node at recur for both subtrees.

   Complete code.

Implementation

```
/* C++ program to checks if a binary tree is max heap or not */
#include <bits/stdc++.h>

using namespace std;

/*  Tree node structure */
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
};

/* Helper function that allocates a new node */
struct Node *newNode(int k)
{
    struct Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function counts the number of nodes in a binary tree */
unsigned int countNodes(struct Node* root)
{
    if (root == NULL)
        return (0);
    return (1 + countNodes(root->left) + countNodes(root->right));
}

/* This function checks if the binary tree is complete or not */
bool isCompleteUtil (struct Node* root, unsigned int index,
                     unsigned int number_nodes)
{
    // An empty tree is complete
    if (root == NULL)
        return (true);

    // If index assigned to current node is more than
    // number of nodes in tree, then tree is not complete
    if (index >= number_nodes)
        return (false);

    // Recur for left and right subtrees
    return (isCompleteUtil(root->left, 2*index + 1, number_nodes)
&&
            isCompleteUtil(root->right, 2*index + 2,
number_nodes));
}

// This Function checks the heap property in the tree.
bool isHeapUtil(struct Node* root)
{
    //  Base case : single node satisfies property
    if (root->left == NULL && root->right == NULL)
        return (true);

    //  node will be in second last level
    if (root->right == NULL)
    {
        //  check heap property at Node
        //  No recursive call , because no need to check last
level
        return (root->key >= root->left->key);
    }
    else
    {
        //  Check heap property at Node and
        //  Recursive check heap property at left and right
subtree
        if (root->key >= root->left->key &&
            root->key >= root->right->key)
            return ((isHeapUtil(root->left)) &&
                    (isHeapUtil(root->right)));
        else
            return (false);
    }
}

//  Function to check binary tree is a Heap or Not.
bool isHeap(struct Node* root)
{
    // These two are used in isCompleteUtil()
    unsigned int node_count = countNodes(root);
    unsigned int index = 0;

    if (isCompleteUtil(root, index, node_count) &&
isHeapUtil(root))
        return true;
    return false;
}

// Driver program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(9);
    root->right = newNode(8);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    root->left->left->left = newNode(3);
    root->left->left->right = newNode(2);
    root->left->right->left = newNode(1);

    if (isHeap(root))
        cout << "Given binary tree is a Heap\n";
    else
        cout << "Given binary tree is not a Heap\n";

    return 0;
}

// This code is contributed by shubhamsingh10
```

Output:

```
Given binary tree is a Heap
```

24. Given level order traversal of a Binary Tree, check if the Tree is a Min-Heap

Given the level order traversal of a Complete Binary Tree, determine whether the Binary Tree is a valid Min-Heap

Examples:

```
Input : level = [10, 15, 14, 25, 30]

Output : True

The tree of the given level order traversal is

                  10

                 /  \

                15   14

               /  \

              25   30

We see that each parent has a value less than

its child, and hence satisfies the min-heap

property


Input : level = [30, 56, 22, 49, 30, 51, 2, 67]

Output : False

The tree of the given level order traversal is

                      30

                   /      \

                 56         22

             /      \     /   \

           49        30  51    2

          /
```

```
           67

We observe that at level 0, 30 > 22, and hence

min-heap property is not satisfied
```

We need to check whether each non-leaf node (parent) satisfies the heap property. For this, we check whether each parent (at index i) is smaller than its children (at indices 2*i+1 and 2*i+2, if the parent has two children). If only one child, we only check the parent against index 2*i+1.

```cpp
// C++ program to check if a given tree is
// Binary Heap or not
#include <bits/stdc++.h>
using namespace std;

// Returns true if given level order traversal
// is Min Heap.
bool isMinHeap(int level[], int n)
{
    // First non leaf node is at index (n/2-1).
    // Check whether each parent is greater than child
    for (int i=(n/2-1) ; i>=0 ; i--)
    {
        // Left child will be at index 2*i+1
        // Right child will be at index 2*i+2
        if (level[i] > level[2 * i + 1])
            return false;

        if (2*i + 2 < n)
        {
            // If parent is greater than right child
            if (level[i] > level[2 * i + 2])
                return false;
        }
    }
    return true;
}

// Driver code
int main()
{
    int level[] = {10, 15, 14, 25, 30};
    int n = sizeof(level)/sizeof(level[0]);
    if  (isMinHeap(level, n))
        cout << "True";
    else
```

```
        cout << "False";
    return 0;
}
```

Output:

```
True
```

These algorithms run with worse case O(n) complexity

25. Merge k sorted arrays | Set 1

Given k sorted arrays of size n each, merge them and print the sorted output.

Example:

> *Input:*
>
> *k = 3, n = 4*
>
> *arr[][] = { {1, 3, 5, 7},*
>
> *{2, 4, 6, 8},*
>
> *{0, 9, 10, 11}} ;*

> *Output: 0 1 2 3 4 5 6 7 8 9 10 11*
>
> *Explanation: The output array is a sorted array that contains all the elements of the input matrix.*
>
> *Input:*
>
> *k = 3, n = 4*
>
> *arr[][] = { {1, 5, 6, 8},*
>
> *{2, 4, 10, 12},*
>
> *{3, 7, 9, 11},*
>
> *{13, 14, 15, 16}} ;*
>
> *Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16*
>
> *Explanation: The output array is a sorted array that contains all the elements of the input matrix.*

Naive Approach: The very naive method is to create an output array of size n * k and then copy all the elements into the output array followed by sorting.

- Algorithm:
    1. Create a output array of size n * k.

2. Traverse the matrix from start to end and insert all the elements in output array.
3. Sort and print the output array.

- Implementation:

```
// C++ program to merge k sorted arrays of size n each.
#include<bits/stdc++.h>
using namespace std;
#define n 4


// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
}

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
together
// and prints the final sorted output.
void mergeKArrays(int arr[][n], int a, int output[])
{
    int c=0;

    //traverse the matrix
    for(int i=0; i<a; i++)
    {
        for(int j=0; j<n ;j++)
            output[c++]=arr[i][j];
    }

    //sort the array
    sort(output,output + n*a);

}


// Driver program to test above functions
int main()
{
    // Change n at the top to change number of elements
    // in an array
    int arr[][n] =  {{2, 6, 12, 34},
                     {1, 9, 20, 1000},
                     {23, 34, 90, 2000}};
    int k = sizeof(arr)/sizeof(arr[0]);
    int output[n*k];

    mergeKArrays(arr, 3, output);

    cout << "Merged array is " << endl;
    printArray(output, n*k);

    return 0;
}
```

Output:

```
Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000
```

- Complexity Analysis:
    1. Time Complexity :O(n*k).

       Traversing the matrix requires O(n*k) time.
    2. Space Complexity :O(n*k), The output array is of size n*k.

Efficient Approach The process might begin with merging arrays into groups of two. After the first merge, we have k/2 arrays. Again merge arrays in groups, now we have k/4 arrays. This is similar to merge sort. Divide k arrays into two halves containing an equal number of arrays until there are two arrays in a group. This is followed by merging the arrays in a bottom-up manner.

- Algorithm:
    1. Create a recursive function which takes k arrays and returns the output array.
    2. In the recursive function, if the value of k is 1 then return the array else if the value of k is 2 then merge the two arrays in linear time and return the array.
    3. If the value of k is greater than 2 then divide the group of k elements into two equal halves and recursively call the function, i.e 0 to k/2 array in one recursive function and k/2 to k array in another recursive function.
    4. Print the output array.
- Implementation:

```
// C++ program to merge k sorted arrays of size n each.
#include<bits/stdc++.h>
using namespace std;
```

```
#define n 4

// Merge arr1[0..n1-1] and arr2[0..n2-1] into
// arr3[0..n1+n2-1]
void mergeArrays(int arr1[], int arr2[], int n1,
                             int n2, int arr3[])
{
    int i = 0, j = 0, k = 0;

    // Traverse both array
    while (i<n1 && j <n2)
    {
        // Check if current element of first
        // array is smaller than current element
        // of second array. If yes, store first
        // array element and increment first array
        // index. Otherwise do same with second array
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }

    // Store remaining elements of first array
    while (i < n1)
        arr3[k++] = arr1[i++];

    // Store remaining elements of second array
    while (j < n2)
        arr3[k++] = arr2[j++];
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
}

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
together
// and prints the final sorted output.
void mergeKArrays(int arr[][n],int i,int j, int output[])
{
    //if one array is in range
    if(i==j)
    {
        for(int p=0; p < n; p++)
        output[p]=arr[i][p];
        return;
    }

    //if only two arrays are left them merge them
    if(j-i==1)
    {
        mergeArrays(arr[i],arr[j],n,n,output);
        return;
    }

    //output arrays
    int out1[n*(((i+j)/2)-i+1)],out2[n*(j-((i+j)/2))];

    //divide the array into halves
    mergeKArrays(arr,i,(i+j)/2,out1);
    mergeKArrays(arr,(i+j)/2+1,j,out2);

    //merge the output array
    mergeArrays(out1,out2,n*(((i+j)/2)-i+1),n*(j-((i+j)/
2)),output);

}


// Driver program to test above functions
int main()
{
    // Change n at the top to change number of elements
    // in an array
    int arr[][n] =  {{2, 6, 12, 34},
                     {1, 9, 20, 1000},
                     {23, 34, 90, 2000}};
    int k = sizeof(arr)/sizeof(arr[0]);
    int output[n*k];
    mergeKArrays(arr,0,2, output);

    cout << "Merged array is " << endl;
    printArray(output, n*k);

    return 0;
}
```
Output:

```
Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000
```

- Complexity Analysis:
    1. Time Complexity: O(n * k * log k).
       There are log k levels as in each level the k arrays are divided in half and at each level the k arrays are traversed. So time Complexity is O(n * k).
    2. Space Complexity: O(n * k * log k).
       In each level O(n*k) space is required So Space Complexity is O(n * k * log k).

Alternative Efficient Approach: The idea is to use Min Heap. This MinHeap based solution has the same time complexity which is O(NK log K). But for a different and particular sized array, this solution works much better. The process must start with creating a MinHeap and inserting the first element of all the k arrays. Remove the root element of Minheap and put it in the output array and insert the next element from the array of removed element. To get the result the step must continue until there is no element in the MinHeap left. MinHeap: A Min-Heap is a complete binary tree in which the value in each internal node is smaller than or equal to the values in the children of that node. Mapping the elements of a heap into an array is trivial: if a node is stored at index k, then its left child is stored at index 2k + 1 and its right child at index 2k + 2.

- Algorithm:
    1. Create a min Heap and insert the first element of all k arrays.
    2. Run a loop until the size of MinHeap is greater than zero.
    3. Remove the top element of the MinHeap and print the element.
    4. Now insert the next element from the same array in which the removed element belonged.
    5. If the array doesn’t have any more elements, then replace root with infinite.After replacing the root, heapify the tree.
- Implementation:
-

```cpp
// C++ program to merge k sorted
// arrays of size n each.
#include<bits/stdc++.h>
using namespace std;

#define n 4

// A min-heap node
struct MinHeapNode
{
// The element to be stored
    int element;

// index of the array from which the element is taken
    int i;

// index of the next element to be picked from the array
    int j;
};

// Prototype of a utility function to swap two min-heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{

// pointer to array of elements in heap
    MinHeapNode *harr;

// size of min heap
    int heap_size;
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x;
```

```
MinHeapify(0); }
};

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
together
// and prints the final sorted output.
int *mergeKArrays(int arr[][n], int k)
{

// To store output array
    int *output = new int[n*k];

    // Create a min heap with k heap nodes.
    // Every heap node has first element of an array
    MinHeapNode *harr = new MinHeapNode[k];
    for (int i = 0; i < k; i++)
    {

// Store the first element
        harr[i].element = arr[i][0];

// index of array
        harr[i].i = i;

 // Index of next element to be stored from the array
        harr[i].j = 1;
    }

// Create the heap
    MinHeap hp(harr, k);

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < n*k; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();
        output[count] = root.element;

        // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < n)
        {
            root.element = arr[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
// INT_MAX is for infinite
```

```
else root.element =  INT_MAX;

        // Replace root with next element of array
        hp.replaceMin(root);
    }

    return output;
}

// FOLLOWING ARE IMPLEMENTATIONS OF
// STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given
// array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a
// subtree with root at given index.
// This method assumes that the subtrees
// are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element <
harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x;  *x = *y;  *y = temp;
```

```
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Change n at the top to change number of elements
    // in an array
    int arr[][n] =  {{2, 6, 12, 34},
                     {1, 9, 20, 1000},
                     {23, 34, 90, 2000}};
    int k = sizeof(arr)/sizeof(arr[0]);

    int *output = mergeKArrays(arr, k);

    cout << "Merged array is " << endl;
    printArray(output, n*k);

    return 0;
}
```

Output:

```
Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000
```

- Complexity Analysis:
  1. Time Complexity :O(n * k * log k), Insertion and deletion in a Min Heap requires log k time. So the Overall time compelxity is O(n * k * log k)
  2. Space Complexity :O(k), If Output is not stored then the only space required is the Min-Heap of k elements. So space Comeplexity is O(k).

26. Merge k sorted arrays | Set 2 (Different Sized Arrays)

Given k sorted arrays of possibly different sizes, merge them and print the sorted output.

Examples:

```
Input: k = 3

     arr[][] = { {1, 3},

                {2, 4, 6},

                {0, 9, 10, 11}} ;

Output: 0 1 2 3 4 6 9 10 11


Input: k = 2

     arr[][] = { {1, 3, 20},

                {2, 4, 6}} ;

Output: 1 2 3 4 6 20
```

We have discussed a solution that works for all arrays of same size in Merge k sorted arrays | Set 1.

A simple solution is to create an output array and and one by one copy all arrays to it. Finally, sort the output array using. This approach takes O(N Logn N) time where N is count of all elements.
An efficient solution is to use heap data structure. The time complexity of heap based solution is O(N Log k).
1. Create an output array.
2. Create a min heap of size k and insert 1st element in all the arrays into the heap

3. Repeat following steps while priority queue is not empty.

…..a) Remove minimum element from heap (minimum is always at root) and store it in output array.

…..b) Insert next element from the array from which the element is extracted. If the array doesn’t have any more elements, then do nothing.

```cpp
// C++ program to merge k sorted arrays
// of size n each.
#include <bits/stdc++.h>
using namespace std;

// A pair of pairs, first element is going to
// store value, second element index of array
// and third element index in the array.
typedef pair<int, pair<int, int> > ppi;

// This function takes an array of arrays as an
// argument and all arrays are assumed to be
// sorted. It merges them together and prints
// the final sorted output.
vector<int> mergeKArrays(vector<vector<int> > arr)
{
    vector<int> output;

    // Create a min heap with k heap nodes. Every
    // heap node has first element of an array
    priority_queue<ppi, vector<ppi>, greater<ppi> > pq;

    for (int i = 0; i < arr.size(); i++)
        pq.push({ arr[i][0], { i, 0 } });

    // Now one by one get the minimum element
    // from min heap and replace it with next
    // element of its array
    while (pq.empty() == false) {
        ppi curr = pq.top();
        pq.pop();

        // i ==> Array Number
        // j ==> Index in the array number
        int i = curr.second.first;
        int j = curr.second.second;

        output.push_back(curr.first);

        // The next element belongs to same array as
        // current.
        if (j + 1 < arr[i].size())
            pq.push({ arr[i][j + 1], { i, j + 1 } });
    }

    return output;
}

// Driver program to test above functions
int main()
{
    // Change n at the top to change number
    // of elements in an array
    vector<vector<int> > arr{ { 2, 6, 12 },
                              { 1, 9 },
                              { 23, 34, 90, 2000 } };

    vector<int> output = mergeKArrays(arr);

    cout << "Merged array is " << endl;
    for (auto x : output)
        cout << x << " ";

    return 0;
}
```

Output:

```
Merged array is

1 2 6 9 12 23 34 90 2000
```