Contents

Chapter 1

Adding elements of an array until every element becomes greater than or equal to k

Adding elements of an array until every element becomes greater than or equal to k - GeeksforGeeks

We are given a list of **N** unsorted elements, we need to find minimum number of steps in which the elements of the list can be added to make all the elements greater than or equal to **K**. We are allowed to add two elements together and make them one.

Examples:

```
Input : arr[] = {1 10 12 9 2 3}
          K = 6
Output : 2
First we add (1 + 2), now the new list becomes
3 10 12 9 3, then we add (3 + 3),  now the new
list becomes 6 10 12 9, Now all the elements in
the list are greater than 6. Hence the output is
2 i:e 2 operations are required
to do this.
```

As we can see from above explanation, we need to extract two smallest elements and then add their sum to list. We need to continue this step until all elements are greater than or equal to K.

Method 1 (Brute Force):
We can create a simple array the sort it and then add two minimum elements and keep on storing them back in the array until all the elements become greater than **K**.

Method 2 (Efficient):
If we take a closer look, we can notice that this problem is similar to Huffman coding. We use **Min Heap** as the main operations here are extract min and insert. Both of these operations can be done in O(Log n) time.

C++

```
 // A C++ program to count minimum steps to make all
// elements greater than or equal to k.
#include<bits/stdc++.h>
using namespace std;

// A class for Min Heap
class MinHeap
{
    int *harr;
    int capacity; // maximum size
    int heap_size; // Current count
public:
    // Constructor
    MinHeap(int *arr, int capacity);

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

    int parent(int i)
    {
        return (i-1)/2;
    }

    // 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 extract the root which is the
    // minimum element
    int extractMin();
```

```
    // Returns the minimum key (key at
    // root) from min heap
    int getMin()
    {
        return harr[0];
    }

    int getSize()
    {
        return heap_size;
    }

    // Inserts a new key 'k'
    void insertKey(int k);
};

// Constructor: Builds a heap from
// a given array a[] of given size
MinHeap::MinHeap(int arr[], int n)
{
    heap_size = n;
    capacity = n;
    harr = new int[n];

    for (int i=0; i<n; i++)
        harr[i] = arr[i];

    // building the heap from first
    // non-leaf node by calling max
    // heapify function
    for (int i=n/2-1; i>=0; i--)
        heapify(i);
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(harr[i], harr[parent(i)]);
        i = parent(i);
```

```
    }
}

// Method to remove minimum element
// (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and
    // remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    heapify(0);

    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::heapify(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]);
        heapify(smallest);
    }
}

// Returns count of steps needed to make
// all elements greater than or equal to
// k by adding elements
```

```
int countMinOps(int arr[], int n, int k)
{
    // Build a min heap of array elements
    MinHeap h(arr, n);

    long int res = 0;

    while (h.getMin() < k)
    {
        if (h.getSize() == 1)
            return -1;

        // Extract two minimum elements
        // and insert their sum
        int first = h.extractMin();
        int second = h.extractMin();
        h.insertKey(first + second);

        res++;
    }

    return res;
}

// Driver code
int main()
{
    int arr[] = {1, 10, 12, 9, 2, 3};
    int n  = sizeof(arr)/sizeof(arr[0]);
    int k = 6;
    cout << countMinOps(arr, n, k);
    return 0;
}
```

Java

```
 // A Java program to count minimum steps to make all
// elements greater than or equal to k.
public class Add_Elements {

    // A class for Min Heap
    static class MinHeap
    {
        int[] harr;
        int capacity; // maximum size
        int heap_size; // Current count

        // Constructor: Builds a heap from
```

```
// a given array a[] of given size
MinHeap(int arr[], int n)
{
    heap_size = n;
    capacity = n;
    harr = new int[n];

    for (int i=0; i<n; i++)
        harr[i] = arr[i];

    // building the heap from first
    // non-leaf node by calling max
    // heapify function
    for (int i=n/2-1; i>=0; i--)
        heapify(i);
}

// A recursive method to heapify a subtree
// with root at given index. This method
// assumes that the subtrees are already
// heapified
void heapify(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)
    {
        int temp = harr[i];
        harr[i] = harr[smallest];
        harr[smallest] = temp;
        heapify(smallest);
    }
}

static int parent(int i)
{
    return (i-1)/2;
}

// to get index of left child of
// node at index i
static 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);
}

// Method to remove minimum element
// (or root) from min heap
int extractMin()
{
    if (heap_size <= 0)
        return Integer.MAX_VALUE;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and
    // remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    heapify(0);

    return root;
}

// Returns the minimum key (key at
// root) from min heap
int getMin()
{
    return harr[0];
}

int getSize()
{
    return heap_size;
}

// Inserts a new key 'k'
void insertKey(int k)
{
    // First insert the new key at the end
```

```
        heap_size++;
        int i = heap_size - 1;
        harr[i] = k;

        // Fix the min heap property if it is violated
        while (i != 0 && harr[parent(i)] > harr[i])
        {
             int temp = harr[i];
             harr[i] = harr[parent(i)];
             harr[parent(i)] = temp;
             i = parent(i);
        }
    }
}


// Returns count of steps needed to make
// all elements greater than or equal to
// k by adding elements
static int countMinOps(int arr[], int n, int k)
{
    // Build a min heap of array elements
    MinHeap h = new MinHeap(arr, n);

    int res = 0;

    while (h.getMin() < k)
    {
        if (h.getSize() == 1)
            return -1;

        // Extract two minimum elements
        // and insert their sum
        int first = h.extractMin();
        int second = h.extractMin();
        h.insertKey(first + second);

        res++;
    }

    return res;
}

// Driver code
public static void main(String args[])
{
    int arr[] = {1, 10, 12, 9, 2, 3};
    int n  = arr.length;
```

```
        int k = 6;
        System.out.println(countMinOps(arr, n, k));
    }
}
// This code is contributed by Sumit Ghosh
```

Output:

```
2
```

Source

https://www.geeksforgeeks.org/adding-elements-array-every-element-becomes-greater-k/

Chapter 2

Applications of Heap Data Structure

Applications of Heap Data Structure - GeeksforGeeks

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.

Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in O(logn) time which is a O(n) operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.

Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of this post for details.

References:
http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap07.htm
http://en.wikipedia.org/wiki/Heap_%28data_structure%29

Source

https://www.geeksforgeeks.org/applications-of-heap-data-structure/

Chapter 3

Applications of Priority Queue

Applications of Priority Queue - GeeksforGeeks

A Priority Queue is different from a normal queue, because instead of being a "first-in-first-out", values come out in order by priority. It is an abstract data type that captures the idea of a container whose elements have "priorities" attached to them. An element of highest priority always appears at the front of the queue. If that element is removed, the next highest priority element advances to the front.

A priority queue is typically implemented using Heap data structure.

Applications:

Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

Data compression: It is used in Huffman codes which is used to compresses data.

Artificial Intelligence : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Operating systems: It is also use in Operating System for load balancing (load balancing on server), interrupt handling.

Source

https://www.geeksforgeeks.org/applications-priority-queue/

Chapter 4

Array Representation Of Binary Heap

Array Representation Of Binary Heap - GeeksforGeeks

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array. The representation is done as:

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

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

Source

https://www.geeksforgeeks.org/array-representation-of-binary-heap/

Chapter 5

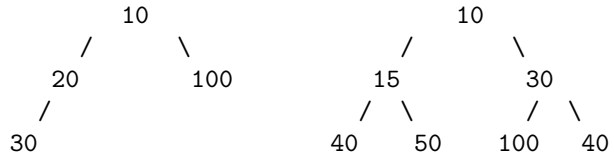
Binary Heap

Binary Heap - GeeksforGeeks

A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

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

Examples of Min Heap:



How is Binary Heap represented?
A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

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

```
2 4 1
```

Coding Practice on Heap
All Articles on Heap
Quiz on Heap
PriorityQueue : Binary Heap Implementation in Java Library

Source

https://www.geeksforgeeks.org/binary-heap/

Chapter 6

Binomial Heap

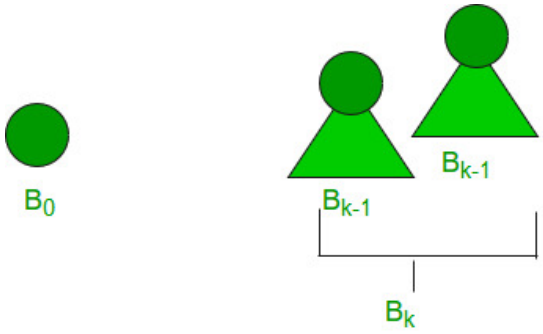
Binomial Heap - GeeksforGeeks

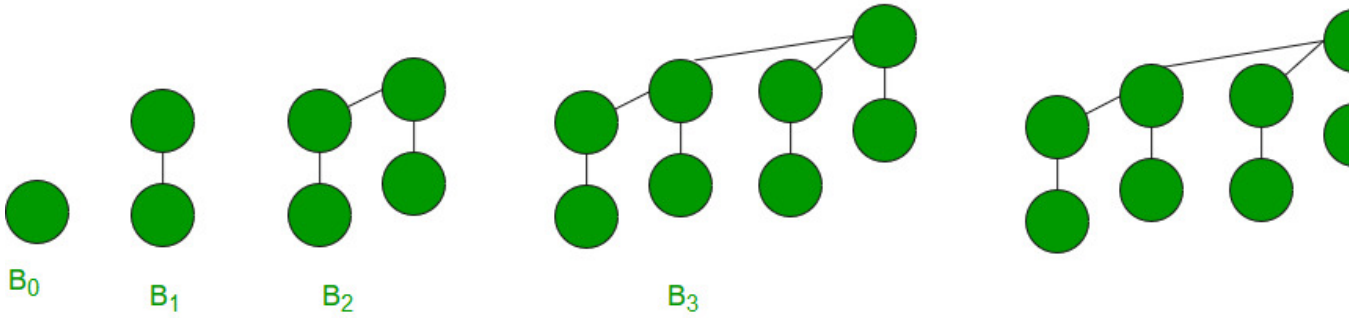
The main application of Binary Heapis as implement priority queue. Binomial Heap is to extension of Binary Heapthat provides faster union or merge operation together with other operations provided by Binary Heap.

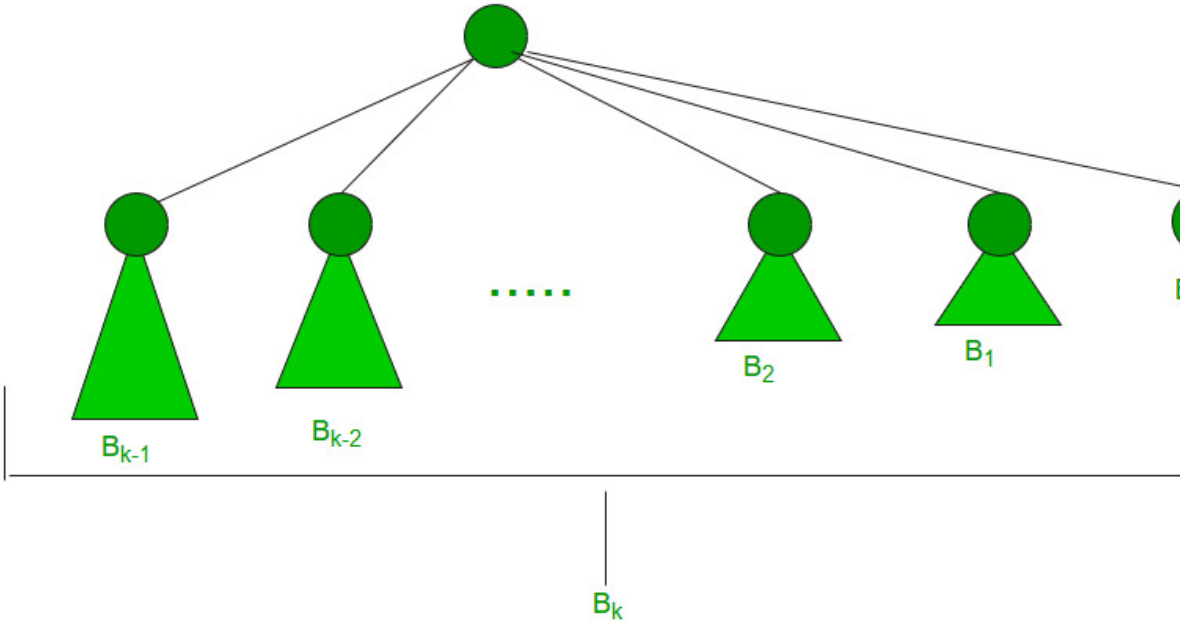
A Binomial Heap is a collection of Binomial Trees

What is a Binomial Tree?
A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other.
A Binomial Tree of order k has following properties.
a) It has exactly 2^k nodes.
b) It has depth as k.
c) There are exactly ${}^kC_i$ nodes at depth i for i = 0, 1, . . . , k.
d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.

The following diagram is referred from 2nd Edition of CLRS book.

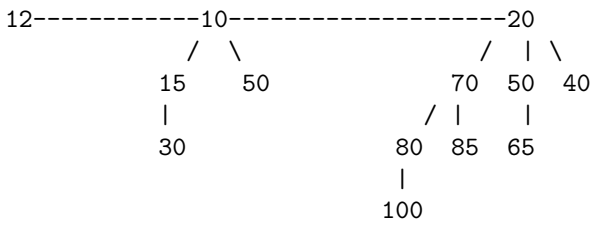

B0
Bk-1
Bk-1
Bk


B0
B1
B2
B3

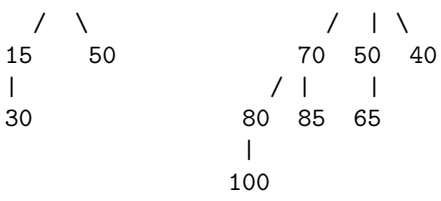

Bk-1
Bk-2
B2
B1
Bk

Binomial Heap:
A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property. And there can be at most one Binomial Tree of any degree.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3
Binomial Trees of orders 0, 2 and 3 from left to right.



A Binomial Heap with 12 nodes. It is a collection of 2
Binomial Trees of orders 2 and 3 from left to right.

Binary Representation of a number and Binomial Heaps
A Binomial Heap with n nodes has the number of Binomial Trees equal to the number of set bits in the Binary representation of n. For example let n be 13, there 3 set bits in the binary representation of n (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are O(Logn) Binomial Trees in a Binomial Heap with 'n' nodes.

Operations of Binomial Heap:
The main operation in Binomial Heap is union(), all other operations mainly use this operation. The union() operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss union later.

1) insert(H, k): Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2) getMin(H): A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.

3) extractMin(H): This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

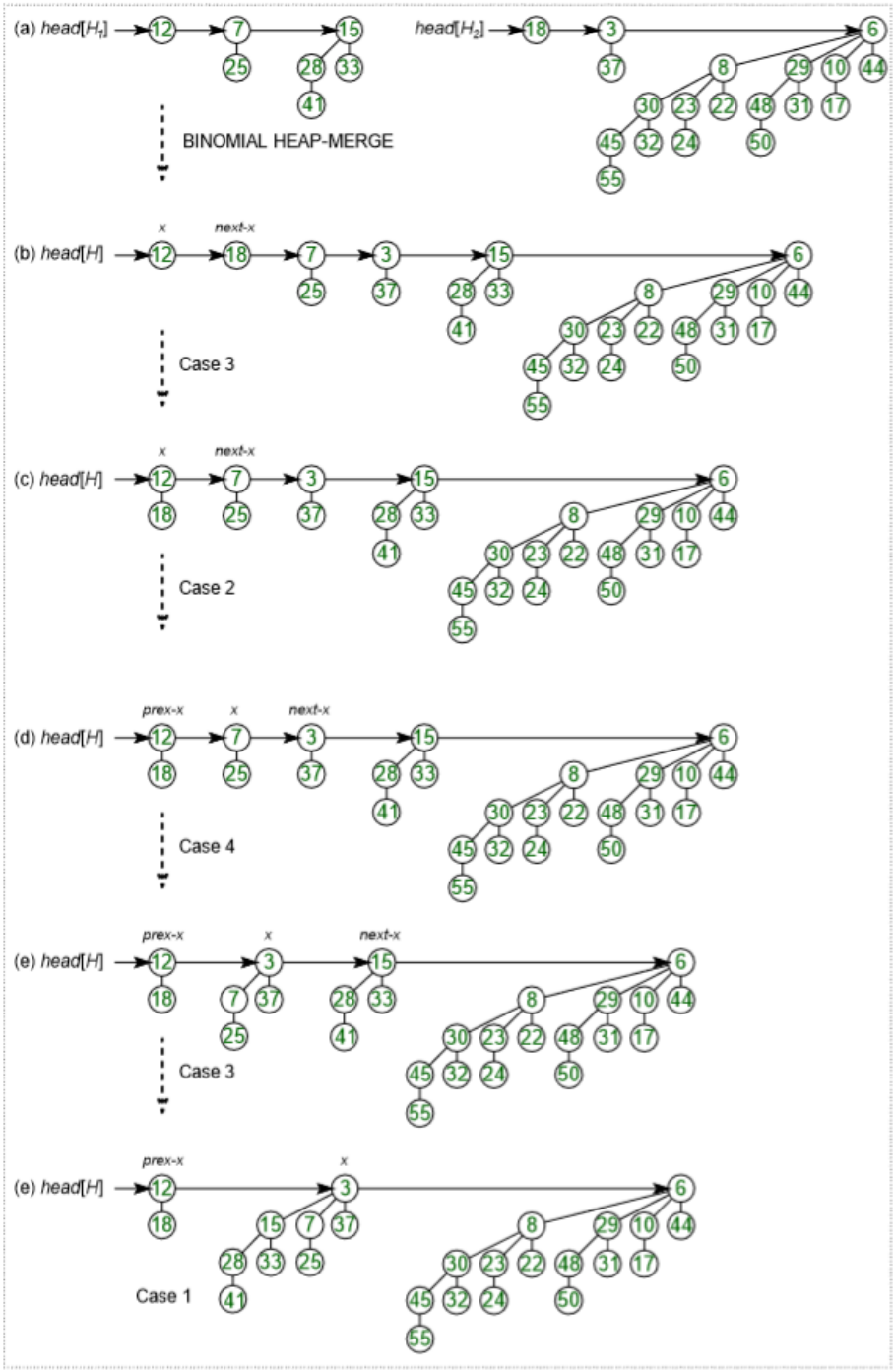
4) delete(H): Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().

5) decreaseKey(H): decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn).

Union operation in Binomial Heap:
Given two Binomial Heaps H1 and H2, union(H1, H2) creates a single Binomial Heap.
1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.

2) After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.
——Case 1: Orders of x and next-x are not same, we simply move ahead.
In following 3 cases orders of x and next-x are same.
——Case 2: If the order of next-next-x is also same, move ahead.
——Case 3: If the key of x is smaller than or equal to the key of next-x, then make next-x as a child of x by linking it with x.
——Case 4: If the key of x is greater, then make x as the child of next.

The following diagram is taken from 2nd Edition of CLRS book.



How to represent Binomial Heap?
A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this

in and extractMin() and delete()). The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

Sources:
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/binomial-heap-2/

Chapter 7

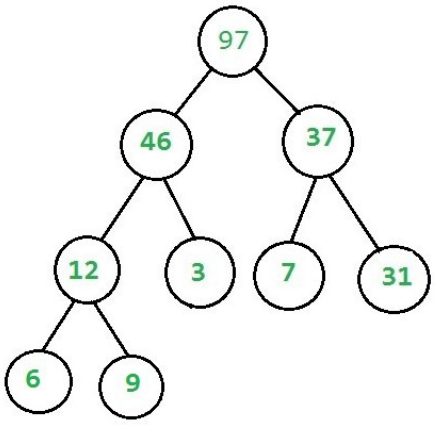
Check if a given Binary Tree is Heap

Check if a given Binary Tree is Heap - GeeksforGeeks

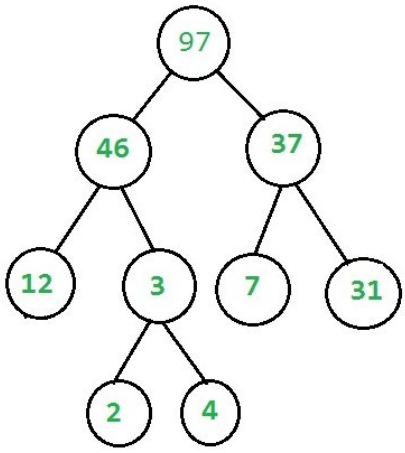
Given a binary tree we need to check it has heap property or not, Binary tree need to fulfill following two conditions for being a heap –

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 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 return 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 Node has both child then check heap property at Node at recur for both subtrees. Complete code.

Implementation

C/C++

```
 /* C program to checks if a binary tree is max heap ot not */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/*  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 = (struct Node*)malloc(sizeof(struct 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))
        printf("Given binary tree is a Heap\n");
    else
        printf("Given binary tree is not a Heap\n");

    return 0;
}
```

Java

```
 /* Java program to checks if a binary tree is max heap ot not */

// A Binary Tree node
class Node
{
    int key;
    Node left, right;

    Node(int k)
    {
        key = k;
        left = right = null;
    }
}

class Is_BinaryTree_MaxHeap
{
    /* This function counts the number of nodes in a binary tree */
    int countNodes(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 */
    boolean isCompleteUtil(Node root, int index, 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.
    boolean isHeapUtil(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.
boolean isHeap(Node root)
{
    if(root == null)
        return true;

    // These two are used in isCompleteUtil()
    int node_count = countNodes(root);

    if(isCompleteUtil(root, 0 , node_count)==true && isHeapUtil(root)==true)
        return true;
    return false;
}

// driver function to test the above functions
public static void main(String args[])
{
    Is_BinaryTree_MaxHeap bt = new Is_BinaryTree_MaxHeap();

    Node root = new Node(10);
    root.left = new Node(9);
    root.right = new Node(8);
    root.left.left = new Node(7);
    root.left.right = new Node(6);
    root.right.left = new Node(5);
    root.right.right = new Node(4);
```

```
        root.left.left.left = new Node(3);
        root.left.left.right = new Node(2);
        root.left.right.left = new Node(1);

        if(bt.isHeap(root) == true)
            System.out.println("Given binary tree is a Heap");
        else
            System.out.println("Given binary tree is not a Heap");
    }
}

// This code has been contributed by Amit Khandelwal
```

Python

```
 # To check if a binary tree
# is a MAX Heap or not
class GFG:
    def __init__(self, value):
        self.key = value
        self.left = None
        self.right = None

    def count_nodes(self, root):
        if root is None:
            return 0
        else:
            return (1 + self.count_nodes(root.left) +
                        self.count_nodes(root.right))

    def heap_propert_util(self, root):

        if (root.left is None and
            root.right is None):
            return True

        if root.right is None:
            return root.key >= root.left.key
        else:
            if (root.key >= root.left.key and
                root.key >= root.right.key):
                return (self.heap_propert_util(root.left) and
                        self.heap_propert_util(root.right))
            else:
                return False

    def complete_tree_util(self, root,
                           index, node_count):
```

```
        if root is None:
            return True
        if index >= node_count:
            return False
        return (self.complete_tree_util(root.left, 2 *
                                         index + 1, node_count) and
                self.complete_tree_util(root.right, 2 *
                                         index + 2, node_count))

    def check_if_heap(self):
        node_count = self.count_nodes(self)
        if (self.complete_tree_util(self, 0, node_count) and
            self.heap_propert_util(self)):
            return True
        else:
            return False

# Driver Code
root = GFG(5)
root.left = GFG(2)
root.right = GFG(3)
root.left.left = GFG(1)

if root.check_if_heap():
    print("Given binary tree is a heap")
else:
    print("Given binary tree is not a Heap")

# This code has been
# contributed by Yash Agrawal
```

Output:

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

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : scorncer17

Source

https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-heap/

Chapter 8

Connect n ropes with minimum cost

Connect n ropes with minimum cost - GeeksforGeeks

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 C++ implementation of above algorithm.

```
 // C++ program for connecting n ropes with minimum cost
#include <iostream>

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.

C++

```
 #include<iostream>
#include<queue>

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;
}
```

Java

```
 // Java program to connect n
// ropes with minimum cost
import java.util.*;

class ConnectRopes
{
static int minCost(int arr[], int n)
{
    // Create a priority queue
    PriorityQueue<Integer> pq =
                        new PriorityQueue<Integer>();

    // Adding items to the pQueue
    for(int i=0;i<n;i++)
    {
        pq.add(arr[i]);
    }

    // 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.poll();
        int second = pq.poll();

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

    return res;
}
```

```
// Driver program to test above function
public static void main(String args[])
{
    int len[] = {4, 3, 2, 6};
    int size = len.length;
    System.out.println("Total cost for connecting"+
                " ropes is " + minCost(len, size));

}
}
// This code is contributed by yash_pec
```

Output:

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

This article is compiled by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : JAGRITIBANSAL, AbhijeetSrivastava

Source

https://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/

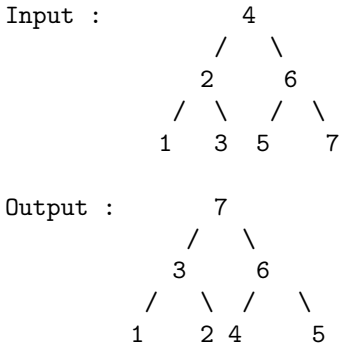
Chapter 9

Convert BST to Max Heap

Convert BST to Max Heap - GeeksforGeeks

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:




```
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

Source

https://www.geeksforgeeks.org/convert-bst-to-max-heap/

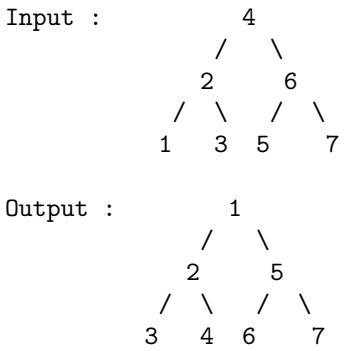
Chapter 10

Convert BST to Min Heap

Convert BST to Min Heap - GeeksforGeeks

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:




```
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)

Source

https://www.geeksforgeeks.org/convert-bst-min-heap/

Chapter 11

Convert min Heap to max Heap

Convert min Heap to max Heap - GeeksforGeeks

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

C++

```
 // 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;
}
```

Java

```
 // Java program to convert min Heap to max Heap

class GFG
{
    // To heapify a subtree with root at given index
    static 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] and arr[largest]
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            MaxHeapify(arr, largest, n);
        }
    }

    // This function basically builds max heap
    static 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
    static void printArray(int arr[], int size)
    {
        for (int i = 0; i < size; ++i)
            System.out.print(arr[i]+" ");
    }

    // driver program
    public static void main (String[] args)
    {
        // array representing Min Heap
        int arr[] = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};
        int n = arr.length;

        System.out.print("Min Heap array : ");
        printArray(arr, n);

        convertMaxHeap(arr, n);

        System.out.print("\nMax Heap array : ");
        printArray(arr, n);
    }
}

// Contributed by Pramod Kumar
```

C#

```
 // C# program to convert
// min Heap to max Heap
using System;

class GFG
{
    // To heapify a subtree with
    // root at given index
    static 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] and arr[largest]
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;
            MaxHeapify(arr, largest, n);
        }
    }

    // This function basically
    // builds max heap
    static 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
    static void printArray(int []arr,
                           int size)
    {
        for (int i = 0; i < size; ++i)
            Console.Write(arr[i]+" ");
    }

    // Driver Code
    public static void Main ()
    {
        // array representing Min Heap
        int []arr = {3, 5, 9, 6, 8,
                     20, 10, 12, 18, 9};
        int n = arr.Length;

        Console.Write("Min Heap array : ");
        printArray(arr, n);

        convertMaxHeap(arr, n);

        Console.Write("\nMax Heap array : ");
        printArray(arr, n);
    }
}
```

```
// This code is contributed by nitin mittal.
```

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.

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/convert-min-heap-to-max-heap/

Chapter 12

Design an efficient data structure for given operations

Design an efficient data structure for given operations - GeeksforGeeks

Design a Data Structure for the following operations. The data structure should be efficient enough to accommodate the operations according to their frequency.

```
1) findMin() : Returns the minimum item.
   Frequency: Most frequent

2) findMax() : Returns the maximum item.
    Frequency: Most frequent

3) deleteMin() : Delete the minimum item.
    Frequency: Moderate frequent

4) deleteMax() : Delete the maximum item.
    Frequency: Moderate frequent

5) Insert() : Inserts an item.
    Frequency: Least frequent

6) Delete() : Deletes an item.
    Frequency: Least frequent.
```

A **simple solution** is to maintain a sorted array where smallest element is at first position and largest element is at last. The time complexity of findMin(), findMAx() and deleteMax() is O(1). But time complexities of deleteMin(), insert() and delete() will be O(n).

Can we do the most frequent two operations in O(1) and other operations in O(Logn) time?.

The idea is to use two binary heaps (one max and one min heap). The main challenge is, while deleting an item, we need to delete from both min-heap and max-heap. So, we need some kind of mutual data structure. In the following design, we have used doubly linked list as a mutual data structure. The doubly linked list contains all input items and indexes of corresponding min and max heap nodes. The nodes of min and max heaps store addresses of nodes of doubly linked list. The root node of min heap stores the address of minimum item in doubly linked list. Similarly, root of max heap stores address of maximum item in doubly linked list. Following are the details of operations.

1) findMax(): We get the address of maximum value node from root of Max Heap. So this is a O(1) operation.

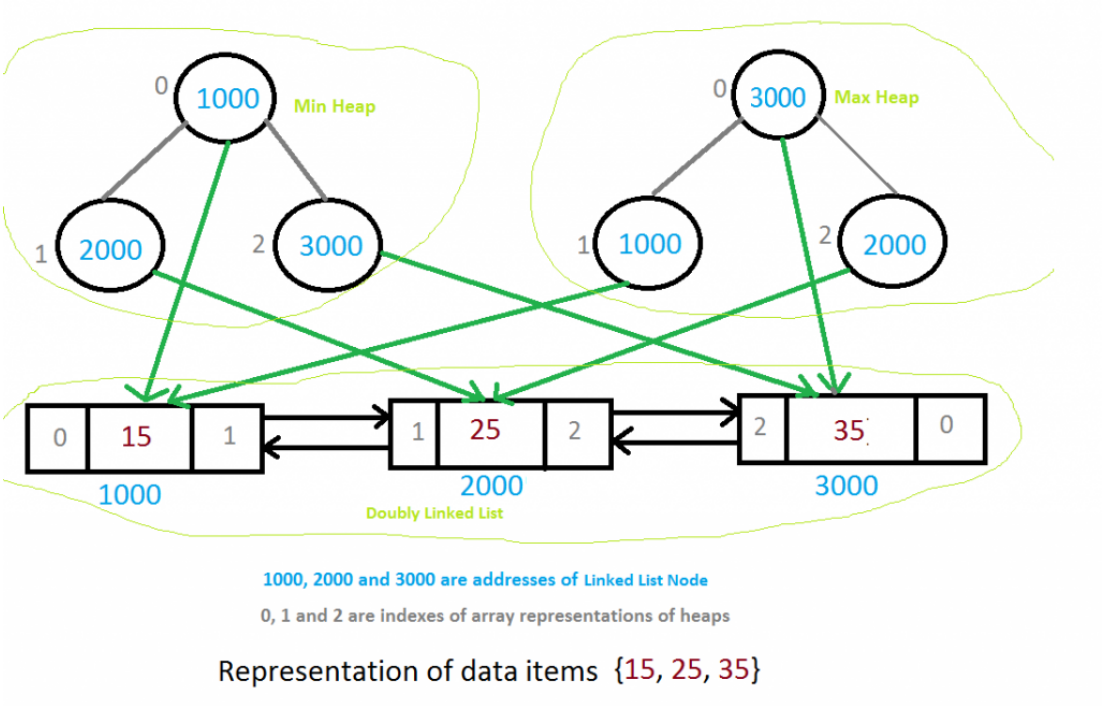
1) findMin(): We get the address of minimum value node from root of Min Heap. So this is a O(1) operation.

3) deleteMin(): We get the address of minimum value node from root of Min Heap. We use this address to find the node in doubly linked list. From the doubly linked list, we get node of Max Heap. We delete node from all three. We can delete a node from doubly linked list in O(1) time. delete() operations for max and min heaps take O(Logn) time.

4) deleteMax(): is similar to deleteMin()

5) Insert() We always insert at the beginning of linked list in O(1) time. Inserting the address in Max and Min Heaps take O(Logn) time. So overall complexity is O(Logn)

6) Delete() We first search the item in Linked List. Once the item is found in O(n) time, we delete it from linked list. Then using the indexes stored in linked list, we delete it from Min Heap and Max Heaps in O(Logn) time. *So overall complexity of this operation is O(n). The Delete operation can be optimized to O(Logn) by using a balanced binary search tree instead of doubly linked list as a mutual data structure. Use of balanced binary search will not effect time complexity of other operations as it will act as a mutual data structure like doubly Linked List.*



Following is C implementation of the above data structure.

```
 // C program for efficient data structure
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A node of doubly linked list
struct LNode
{
    int data;
    int minHeapIndex;
    int maxHeapIndex;
    struct LNode *next, *prev;
};

// Structure for a doubly linked list
struct List
{
    struct LNode *head;
};

// Structure for min heap
struct MinHeap
{
```

```
    int size;
    int capacity;
    struct LNode* *array;
};

// Structure for max heap
struct MaxHeap
{
    int size;
    int capacity;
    struct LNode* *array;
};

// The required data structure
struct MyDS
{
    struct MinHeap* minHeap;
    struct MaxHeap* maxHeap;
    struct List* list;
};

// Function to swap two integers
void swapData(int* a, int* b)
{ int t = *a;   *a = *b;   *b = t; }

// Function to swap two List nodes
void swapLNode(struct LNode** a, struct LNode** b)
{ struct LNode* t = *a; *a = *b; *b = t; }

// A utility function to create a new List node
struct LNode* newLNode(int data)
{
    struct LNode* node =
        (struct LNode*) malloc(sizeof(struct LNode));
    node->minHeapIndex = node->maxHeapIndex = -1;
    node->data = data;
    node->prev = node->next = NULL;
    return node;
}

// Utility function to create a max heap of given capacity
struct MaxHeap* createMaxHeap(int capacity)
{
    struct MaxHeap* maxHeap =
     (struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = 0;
    maxHeap->capacity = capacity;
    maxHeap->array =
```

```
     (struct LNode**) malloc(maxHeap->capacity * sizeof(struct LNode*));
    return maxHeap;
}

// Utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
       (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
       (struct LNode**) malloc(minHeap->capacity * sizeof(struct LNode*));
    return minHeap;
}

// Utility function to create a List
struct List* createList()
{
    struct List* list =
      (struct List*) malloc(sizeof(struct List));
    list->head = NULL;
    return list;
}

// Utility function to create the main data structure
// with given capacity
struct MyDS* createMyDS(int capacity)
{
    struct MyDS* myDS =
        (struct MyDS*) malloc(sizeof(struct MyDS));
    myDS->minHeap = createMinHeap(capacity);
    myDS->maxHeap = createMaxHeap(capacity);
    myDS->list = createList();
    return myDS;
}

// Some basic operations for heaps and List
int isMaxHeapEmpty(struct MaxHeap* heap)
{  return (heap->size == 0); }

int isMinHeapEmpty(struct MinHeap* heap)
{  return heap->size == 0; }

int isMaxHeapFull(struct MaxHeap* heap)
{  return heap->size == heap->capacity; }

int isMinHeapFull(struct MinHeap* heap)
```

```
{  return heap->size == heap->capacity; }

int isListEmpty(struct List* list)
{  return !list->head;   }

int hasOnlyOneLNode(struct List* list)
{    return !list->head->next && !list->head->prev; }


// The standard minheapify function.  The only thing it does extra
// is swapping indexes of heaps inside the List
void minHeapify(struct MinHeap* minHeap, int index)
{
    int smallest, left, right;
    smallest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( minHeap->array[left] &&
         left < minHeap->size &&
         minHeap->array[left]->data < minHeap->array[smallest]->data
       )
        smallest = left;

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

    if (smallest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&(minHeap->array[smallest]->minHeapIndex),
                 &(minHeap->array[index]->minHeapIndex));

        // Now swap pointers to List nodes
        swapLNode(&minHeap->array[smallest],
                  &minHeap->array[index]);

        // Fix the heap downward
        minHeapify(minHeap, smallest);
    }
}

// The standard maxHeapify function.  The only thing it does extra
// is swapping indexes of heaps inside the List
```

```
void maxHeapify(struct MaxHeap* maxHeap, int index)
{
    int largest, left, right;
    largest = index;
    left = 2 * index + 1;
    right = 2 * index + 2;

    if ( maxHeap->array[left] &&
         left < maxHeap->size &&
         maxHeap->array[left]->data > maxHeap->array[largest]->data
       )
        largest = left;

    if ( maxHeap->array[right] &&
         right < maxHeap->size &&
         maxHeap->array[right]->data > maxHeap->array[largest]->data
       )
        largest = right;

    if (largest != index)
    {
        // First swap indexes inside the List using address
        // of List nodes
        swapData(&maxHeap->array[largest]->maxHeapIndex,
                 &maxHeap->array[index]->maxHeapIndex);

        // Now swap pointers to List nodes
        swapLNode(&maxHeap->array[largest],
                  &maxHeap->array[index]);

        // Fix the heap downward
        maxHeapify(maxHeap, largest);
    }
}

// Standard function to insert an item in Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct LNode* temp)
{
    if (isMinHeapFull(minHeap))
        return;

    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && temp->data < minHeap->array[(i - 1) / 2]->data )
    {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        minHeap->array[i]->minHeapIndex = i;
        i = (i - 1) / 2;
```

```
    }

    minHeap->array[i] = temp;
    minHeap->array[i]->minHeapIndex = i;
}

// Standard function to insert an item in Max Heap
void insertMaxHeap(struct MaxHeap* maxHeap, struct LNode* temp)
{
    if (isMaxHeapFull(maxHeap))
        return;

    ++maxHeap->size;
    int i = maxHeap->size - 1;
    while (i && temp->data > maxHeap->array[(i - 1) / 2]->data )
    {
        maxHeap->array[i] = maxHeap->array[(i - 1) / 2];
        maxHeap->array[i]->maxHeapIndex = i;
        i = (i - 1) / 2;
    }

    maxHeap->array[i] = temp;
    maxHeap->array[i]->maxHeapIndex = i;
}


// Function to find minimum value stored in the main data structure
int findMin(struct MyDS* myDS)
{
    if (isMinHeapEmpty(myDS->minHeap))
        return INT_MAX;

    return myDS->minHeap->array[0]->data;
}

// Function to find maximum value stored in the main data structure
int findMax(struct MyDS* myDS)
{
    if (isMaxHeapEmpty(myDS->maxHeap))
        return INT_MIN;

    return myDS->maxHeap->array[0]->data;
}

// A utility function to remove an item from linked list
void removeLNode(struct List* list, struct LNode** temp)
{
    if (hasOnlyOneLNode(list))
```

```
        list->head = NULL;

    else if (!(*temp)->prev) // first node
    {
        list->head = (*temp)->next;
        (*temp)->next->prev = NULL;
    }
    // any other node including last
    else
    {
        (*temp)->prev->next = (*temp)->next;
        // last node
        if ((*temp)->next)
            (*temp)->next->prev = (*temp)->prev;
    }
    free(*temp);
    *temp = NULL;
}

// Function to delete maximum value stored in the main data structure
void deleteMax(struct MyDS* myDS)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMaxHeapEmpty(maxHeap))
        return;
    struct LNode* temp = maxHeap->array[0];

    // delete the maximum item from maxHeap
    maxHeap->array[0] =
        maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[0]->maxHeapIndex = 0;
    maxHeapify(maxHeap, 0);

    // remove the item from minHeap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to delete minimum value stored in the main data structure
void deleteMin(struct MyDS* myDS)
```

```
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isMinHeapEmpty(minHeap))
        return;
    struct LNode* temp = minHeap->array[0];

    // delete the minimum item from minHeap
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[0]->minHeapIndex = 0;
    minHeapify(minHeap, 0);

    // remove the item from maxHeap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove the node from List
    removeLNode(myDS->list, &temp);
}

// Function to enList an item to List
void insertAtHead(struct List* list, struct LNode* temp)
{
    if (isListEmpty(list))
        list->head = temp;

    else
    {
        temp->next = list->head;
        list->head->prev = temp;
        list->head = temp;
    }
}

// Function to delete an item from List. The function also
// removes item from min and max heaps
void Delete(struct MyDS* myDS, int item)
{
    MinHeap *minHeap = myDS->minHeap;
    MaxHeap *maxHeap = myDS->maxHeap;

    if (isListEmpty(myDS->list))
        return;
```

```
    // search the node in List
    struct LNode* temp = myDS->list->head;
    while (temp && temp->data != item)
        temp = temp->next;

    // if item not found
    if (!temp || temp && temp->data != item)
        return;

    // remove item from min heap
    minHeap->array[temp->minHeapIndex] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeap->array[temp->minHeapIndex]->minHeapIndex = temp->minHeapIndex;
    minHeapify(minHeap, temp->minHeapIndex);

    // remove item from max heap
    maxHeap->array[temp->maxHeapIndex] = maxHeap->array[maxHeap->size - 1];
    --maxHeap->size;
    maxHeap->array[temp->maxHeapIndex]->maxHeapIndex = temp->maxHeapIndex;
    maxHeapify(maxHeap, temp->maxHeapIndex);

    // remove node from List
    removeLNode(myDS->list, &temp);
}

// insert operation for main data structure
void Insert(struct MyDS* myDS, int data)
{
    struct LNode* temp = newLNode(data);

    // insert the item in List
    insertAtHead(myDS->list, temp);

    // insert the item in min heap
    insertMinHeap(myDS->minHeap, temp);

    // insert the item in max heap
    insertMaxHeap(myDS->maxHeap, temp);
}

// Driver program to test above functions
int main()
{
    struct MyDS *myDS = createMyDS(10);
    // Test Case #1
    /*Insert(myDS, 10);
    Insert(myDS, 2);
    Insert(myDS, 32);
```

```
    Insert(myDS, 40);
    Insert(myDS, 5);*/

    // Test Case #2
    Insert(myDS, 10);
    Insert(myDS, 20);
    Insert(myDS, 30);
    Insert(myDS, 40);
    Insert(myDS, 50);

    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMax(myDS);  // 50 is deleted
    printf("After deleteMax()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    deleteMin(myDS); // 10 is deleted
    printf("After deleteMin()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n\n", findMin(myDS));

    Delete(myDS, 40); // 40 is deleted
    printf("After Delete()\n");
    printf("Maximum = %d \n", findMax(myDS));
    printf("Minimum = %d \n", findMin(myDS));

    return 0;
}
```

Output:

```
Maximum = 50
Minimum = 10

After deleteMax()
Maximum = 40
Minimum = 10

After deleteMin()
Maximum = 40
Minimum = 20

After Delete()
Maximum = 30
Minimum = 20
```

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/a-data-structure-question/

Chapter 13

Fibonacci Heap | Set 1 (Introduction)

Fibonacci Heap | Set 1 (Introduction) - GeeksforGeeks

Heaps are mainly used for implementing priority queue. We have discussed below heaps in previous posts.

Binary Heap
Binomial Heap

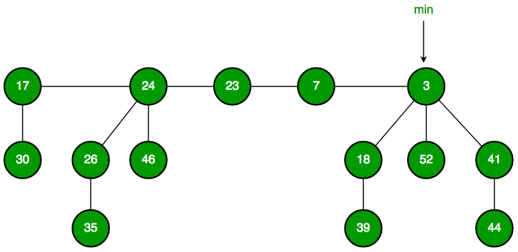
In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are amortized time complexities of **Fibonacci Heap**.

```
1) Find Min:      Θ(1)     [Same as both Binary and Binomial]
2) Delete Min:    O(Log n) [Θ(Log n) in both Binary and Binomial]
3) Insert:        Θ(1)     [Θ(Log n) in Binary and Θ(1) in Binomial]
4) Decrease-Key:  Θ(1)     [Θ(Log n) in both Binary and Binomial]
5) Merge:         Θ(1)     [Θ(m Log n) or Θ(m+n) in Binary and
                            Θ(Log n) in Binomial]
```

Like Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

Below is an example Fibonacci Heap taken from here.



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is O(VLogV + ELogV). If Fibonacci Heap is used, then time complexity is improved to O(VLogV + E)
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source Wiki).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number.

We will soon be discussing Fibonacci Heap operations in detail.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/

Chapter 14

Find k numbers with most occurrences in the given array

Find k numbers with most occurrences in the given array - GeeksforGeeks

Given an array of **n** numbers and a positive integer **k**. The problem is to find **k** numbers with most occurrences, i.e., the top **k** numbers having the maximum frequency. If two numbers have same frequency then the larger number should be given preference. The numbers should be displayed in decreasing order of their frequencies. It is assumed that the array consists of **k** numbers with most occurrences.

Examples:

```
Input : arr[] = {3, 1, 4, 4, 5, 2, 6, 1},
             k = 2
Output : 4 1
Frequency of 4 = 2
Frequency of 1 = 2
These two have the maximum frequency and
4 is larger than 1.

Input : arr[] = {7, 10, 11, 5, 2, 5, 5, 7, 11, 8, 9},
            k = 4
Output : 5 11 7 10
```

Asked in Amazon Interview

Method 1: Using hash table, we create a frequency table which stores the frequency of occurrence of each number in the given array. In the hash table we define **(x, y)** tuple, where **x** is the key(number) and **y** is its frequency in the array. Now we traverse this hash table and create an array **freq_arr[]** which stores these (number, frequency) tuples. Sort this **freq_arr[]** on the basis of the conditions defined in the problem statement. Now, print the first **k** numbers of this **freq_arr[]**.

```
 // C++ implementation to find k numbers with most
// occurrences in the given array
#include <bits/stdc++.h>

using namespace std;

// comparison function to sort the 'freq_arr[]'
bool compare(pair<int, int> p1, pair<int, int> p2)
{
    // if frequencies of two elements are same
    // then the larger number should come first
    if (p1.second == p2.second)
        return p1.first > p2.first;

    // sort on the basis of decreasing order
    // of frequencies
    return p1.second > p2.second;
}

// funnction to print the k numbers with most occurrences
void print_N_mostFrequentNumber(int arr[], int n, int k)
{
    // unordered_map 'um' implemented as frequency hash table
    unordered_map<int, int> um;
    for (int i = 0; i<n; i++)
        um[arr[i]]++;

    // store the elements of 'um' in the vector 'freq_arr'
    vector<pair<int, int> > freq_arr(um.begin(), um.end());

    // sort the vector 'freq_arr' on the basis of the
    // 'compare' function
    sort(freq_arr.begin(), freq_arr.end(), compare);

    // display the the top k numbers
    cout << k << " numbers with most occurrences are:\n";
    for (int i = 0; i<k; i++)
        cout << freq_arr[i].first << " ";
}

// Driver program to test above
int main()
{
    int arr[] = {3, 1, 4, 4, 5, 2, 6, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;
    print_N_mostFrequentNumber(arr, n, k);
    return 0;
```

```
}
```

Output:

```
2 numbers with most occurrences are:
4 1
```

Time Complexity: O(dlogd), where **d** is the count of distinct elements in the array.
Auxiliary Space: O(d), where **d** is the count of distinct elements in the array.

Method 2: Create the array **freq_arr**[] as described in **Method 1** of this post. Now, build the max heap using elements of this **freq_arr**[]. The root of the max heap should be the most frequent number and in case of conflicts the larger number gets the preference. Now remove the top **k** numbers of this max heap. C++ STL priority_queue has been used as max heap.

```
 // C++ implementation to find k numbers with most
// occurrences in the given array
#include <bits/stdc++.h>

using namespace std;

// comparison function defined for the priority queue
struct compare
{
    bool operator()(pair<int, int> p1, pair<int, int> p2)
    {
        // if frequencies of two elements are same
        // then the larger number should come first
        if (p1.second == p2.second)
            return p1.first < p2.first;

        // insert elements in the priority queue on the basis of
        // decreasing order of frequencies
        return p1.second < p2.second;
    }
};

// funnction to print the k numbers with most occurrences
void print_N_mostFrequentNumber(int arr[], int n, int k)
{
    // unordered_map 'um' implemented as frequency hash table
    unordered_map<int, int> um;
    for (int i = 0; i<n; i++)
        um[arr[i]]++;

    // store the elements of 'um' in the vector 'freq_arr'
```

```
    vector<pair<int, int> > freq_arr(um.begin(), um.end());

    // priority queue 'pq' implemented as max heap on the basis
    // of the comparison operator 'compare'
    // element with the highest frequency is the root of 'pq'
    // in case of conflicts, larger element is the root
    priority_queue<pair<int, int>, vector<pair<int, int> >,
                           compare> pq(um.begin(), um.end());

    // display the the top k numbers
    cout << k << " numbers with most occurrences are:\n";
    for (int i = 1; i<= k; i++)
    {
        cout << pq.top().first << " ";
        pq.pop();
    }
}

// Driver program to test above
int main()
{
    int arr[] = {3, 1, 4, 4, 5, 2, 6, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 2;
    print_N_mostFrequentNumber(arr, n, k);
    return 0;
}
```

Output:

```
2 numbers with most occurrences are:
4 1
```

Time Complexity: O(klogd), where **d** is the count of distinct elements in the array.
Auxiliary Space: O(d), where **d** is the count of distinct elements in the array.

References: https://www.careercup.com/question?id=5082885552865280

Source

https://www.geeksforgeeks.org/find-k-numbers-occurrences-given-array/

Chapter 15

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

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

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.

C++

```
 // 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;
}
```

Java

```
 // Java program to check if a given tree is
// Binary Heap or not
import java.io.*;
import java.util.*;

public class detheap
{
    // Returns true if given level order traversal
    // is Min Heap.
    static boolean isMinHeap(int []level)
    {
        int n = level.length - 1;

        // 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
    public static void main(String[] args)
                              throws IOException
    {
        // Level order traversal
        int[] level = new int[]{10, 15, 14, 25, 30};

        if  (isMinHeap(level))
            System.out.println("True");
        else
```

```
            System.out.println("False");
    }
}
```

Output:

```
True
```

These algorithms run with worse case **O(n)** complexity

Source

https://www.geeksforgeeks.org/given-level-order-traversal-binary-tree-check-tree-min-heap/

Chapter 16

Heap Sort for decreasing order using min heap

Heap Sort for decreasing order using min heap - GeeksforGeeks

Given an array of elements, sort the array in decreasing order using min heap.

```
Input : arr[] = {5, 3, 10, 1}
Output : arr[] = {10, 5, 3, 1}

Input : arr[] = {1, 50, 100, 25}
Output : arr[] = {100, 50, 25, 1}
```

Prerequisite : Heap sort using min heap.

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

Note : Heap Sort using min heap sorts in descending order where as max heap sorts in ascending order

C++

```
 // C++ program for implementation of Heap Sort
#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 smallest = i; // Initialize smalles as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is smaller than root
    if (l < n && arr[l] < arr[smallest])
        smallest = l;

    // If right child is smaller than smallest so far
    if (r < n && arr[r] < arr[smallest])
        smallest = r;

    // If smallest is not root
    if (smallest != i) {
        swap(arr[i], arr[smallest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, smallest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* 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 program
int main()
{
    int arr[] = { 4, 6, 3, 2, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Java

```
 // Java program for implementation of Heap Sort

import java.io.*;

class GFG {

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    static void heapify(int arr[], int n, int i)
    {
        int smallest = i; // Initialize smalles as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2

        // If left child is smaller than root
        if (l < n && arr[l] < arr[smallest])
            smallest = l;

        // If right child is smaller than smallest so far
        if (r < n && arr[r] < arr[smallest])
            smallest = r;

        // If smallest is not root
        if (smallest != i) {
            int temp = arr[i];
            arr[i] = arr[smallest];
            arr[smallest] = temp;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, smallest);
        }
    }

    // main function to do heap sort
```

```
    static void heapSort(int arr[], int n)
    {
        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i = n - 1; i >= 0; i--) {

            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

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

    // Driver program
    public static void main(String[] args)
    {
        int arr[] = { 4, 6, 3, 2, 9 };
        int n = arr.length;

        heapSort(arr, n);

        System.out.println("Sorted array is ");
        printArray(arr, n);
    }
}

// This code is contributed by vt_m.
```

C#

```
 // C# program for implementation of Heap Sort
using System;

class GFG {
```

```
// To heapify a subtree rooted with
// node i which is an index in arr[],
// n is size of heap
static void heapify(int[] arr, int n, int i)
{
    int smallest = i; // Initialize smalles as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is smaller than root
    if (l < n && arr[l] < arr[smallest])
        smallest = l;

    // If right child is smaller than smallest so far
    if (r < n && arr[r] < arr[smallest])
        smallest = r;

    // If smallest is not root
    if (smallest != i) {
        int temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, smallest);
    }
}

// main function to do heap sort
static void heapSort(int[] arr, int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {

        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

```
    /* A utility function to print array of size n */
    static void printArray(int[] arr, int n)
    {
        for (int i = 0; i < n; ++i)
            Console.Write(arr[i] + " ");
        Console.WriteLine();
    }

    // Driver program
    public static void Main()
    {
        int[] arr = { 4, 6, 3, 2, 9 };
        int n = arr.Length;

        heapSort(arr, n);

        Console.WriteLine("Sorted array is ");
        printArray(arr, n);
    }
}

// This code is contributed by vt_m.
```

Output:

```
Sorted array is
9 6 4 3 2
```

Time complexity: It takes **O(logn)** for **heapify** and **O(n)** for **constructing a heap**. Hence, the overall time complexity of **heap sort** using **min heap** or **max heap** is **O(nlogn)**

Source

https://www.geeksforgeeks.org/heap-sort-for-decreasing-order-using-min-heap/

Chapter 17

Heap in C++ STL | make_heap(), push_heap(), pop_heap(), sort_heap(), is_heap, is_heap_until()

Heap in C++ STL | make_heap(), push_heap(), pop_heap(), sort_heap(), is_heap, is_heap_until() - GeeksforGeeks

Heap data structure can be implemented in a range using STL which allows faster input into heap and retrieval of a number always results in the largest number i.e. largest number of the remaining numbers is popped out each time. Other numbers of the heap are arranged depending upon the implementation.

Operations on heap :

1. make_heap() :- This function is used to **convert a range** in a container **to a heap.**

2. front() :- This function displays the **first element** of heap which is the **maximum number**.

```
 // C++ code to demonstrate the working of
// make_heap(), front()
#include<iostream>
#include<algorithm> // for heap operations
using namespace std;
int main()
{

    // Initializing a vector
    vector<int> v1 = {20, 30, 40, 25, 15};
```

```
    // Converting vector into a heap
    // using make_heap()
    make_heap(v1.begin(), v1.end());

    // Displaying the maximum element of heap
    // using front()
    cout << "The maximum element of heap is : ";
    cout << v1.front() << endl;

    return 0;
}
```

Output:

```
The maximum element of heap is : 40
```

3. push_heap() :- This function is used to **insert** elements into heap. The size of the heap is increased by 1. New element is placed appropriately in the heap.

4. pop_heap() :- This function is used to **delete the maximum element** of the heap. The size of heap is decreased by 1. The heap elements are reorganised accordingly after this operation.

```
 // C++ code to demonstrate the working of
// push_heap() and pop_heap()
#include<iostream>
#include<algorithm> // for heap operations
using namespace std;
int main()
{

    // Initializing a vector
    vector<int> v1 = {20, 30, 40, 25, 15};

    // Converting vector into a heap
    // using make_heap()
    make_heap(v1.begin(), v1.end());

    // Displaying the maximum element of heap
    // using front()
    cout << "The maximum element of heap is : ";
    cout << v1.front() << endl;

    // using push_back() to enter element
    // in vector
    v1.push_back(50);
```

```
    // using push_heap() to reorder elements
    push_heap(v1.begin(), v1.end());

    // Displaying the maximum element of heap
    // using front()
    cout << "The maximum element of heap after push is : ";
    cout << v1.front() << endl;

     // using pop_heap() to delete maximum element
    pop_heap(v1.begin(), v1.end());
    v1.pop_back();

    // Displaying the maximum element of heap
    // using front()
    cout << "The maximum element of heap after pop is : ";
    cout << v1.front() << endl;

    return 0;
}
```

Output:

```
The maximum element of heap is : 40
The maximum element of heap after push is : 50
The maximum element of heap after pop is : 40
```

5. sort_heap() :- This function is used to **sort** the heap. After this operation, the container is **no longer a heap**.

```
 // C++ code to demonstrate the working of
// sort_heap()
#include<iostream>
#include<algorithm> // for heap operations
using namespace std;
int main()
{

    // Initializing a vector
    vector<int> v1 = {20, 30, 40, 25, 15};

    // Converting vector into a heap
    // using make_heap()
    make_heap(v1.begin(), v1.end());

    // Displaying heap elements
    cout << "The heap elements are : ";
```

```
    for (int &x : v1)
       cout << x << " ";
    cout << endl;

    // sorting heap using sort_heap()
    sort_heap(v1.begin(), v1.end());

     // Displaying heap elements
    cout << "The heap elements after sorting are : ";
    for (int &x : v1)
       cout << x << " ";

    return 0;
}
```

Output:

```
The heap elements are : 40 30 20 25 15
The heap elements after sorting are : 15 20 25 30 40
```

6. is_heap() :- This function is used to **check** whether the container is **heap** or not. Generally, in most implementations, the **reverse sorted container** is considered as heap. Returns true if container is heap else returns false.

6. is_heap_until() :- This function returns the iterator to the position **till the container is the heap.** Generally, in most implementations, the **reverse sorted container** is considered as heap.

```
 // C++ code to demonstrate the working of
// is_heap() and is_heap_until()
#include<iostream>
#include<algorithm> // for heap operations
using namespace std;
int main()
{

    // Initializing a vector
    vector<int> v1 = {40, 30, 25, 35, 15};

    // Declaring heap iterator
    vector<int>::iterator it1;

    // Checking if container is heap
    // using is_heap()
    is_heap(v1.begin(), v1.end())?
    cout << "The container is heap ":
```

```
    cout << "The container is not heap";
    cout << endl;

    // using is_heap_until() to check position
    // till which container is heap
    auto it = is_heap_until(v1.begin(), v1.end());

    // Displaying heap range elements
    cout << "The heap elements in container are : ";
    for (it1=v1.begin(); it1!=it; it1++)
       cout << *it1 << " ";

    return 0;
}
```

Output:

```
The container is not heap
The heap elements in container are : 40 30 25
```

Improved By : SamirKhan

Source

https://www.geeksforgeeks.org/heap-using-stl-c/

Chapter 18

HeapSort

HeapSort - GeeksforGeeks

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

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

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

C++

```
 // C++ program for implementation of Heap Sort
#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);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* 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 program
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Java

```
 // Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // 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)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
```

```
            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Python

```
 # Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
```

```
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver code to test above
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
# This code is contributed by Mohit Kumra
```

C#

```
 // C# program for implementation of Heap Sort
using System;

public class HeapSort
{
    public void sort(int[] arr)
    {
        int n = arr.Length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
```

```
    {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// 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)
    {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

/* A utility function to print array of size n */
static void printArray(int[] arr)
{
    int n = arr.Length;
    for (int i=0; i<n; ++i)
        Console.Write(arr[i]+" ");
    Console.Read();
}
```

```
    // Driver program
    public static void Main()
    {
        int[] arr = {12, 11, 13, 5, 6, 7};
        int n = arr.Length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        Console.WriteLine("Sorted array is");
        printArray(arr);
    }
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

PHP

```
 <?php

// Php program for implementation of Heap Sort

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
function heapify(&$arr, $n, $i)
{
    $largest = $i; // Initialize largest as root
    $l = 2*$i + 1; // left = 2*i + 1
    $r = 2*$i + 2; // right = 2*i + 2

    // If left child is larger than root
    if ($l < $n && $arr[$l] > $arr[$largest])
        $largest = $l;

    // If right child is larger than largest so far
    if ($r < $n && $arr[$r] > $arr[$largest])
        $largest = $r;

    // If largest is not root
    if ($largest != $i)
    {
        $swap = $arr[$i];
        $arr[$i] = $arr[$largest];
        $arr[$largest] = $swap;

        // Recursively heapify the affected sub-tree
        heapify($arr, $n, $largest);
```

```
    }
}

// main function to do heap sort
function heapSort(&$arr, $n)
{
    // Build heap (rearrange array)
    for ($i = $n / 2 - 1; $i >= 0; $i--)
        heapify($arr, $n, $i);

    // One by one extract an element from heap
    for ($i = $n-1; $i >= 0; $i--)
    {
        // Move current root to end
        $temp = $arr[0];
            $arr[0] = $arr[$i];
            $arr[$i] = $temp;

        // call max heapify on the reduced heap
        heapify($arr, $i, 0);
    }
}

/* A utility function to print array of size n */
function printArray(&$arr, $n)
{
    for ($i = 0; $i < $n; ++$i)
        echo ($arr[$i]." ") ;

}

// Driver program
    $arr = array(12, 11, 13, 5, 6, 7);
    $n = sizeof($arr)/sizeof($arr[0]);

    heapSort($arr, $n);

    echo 'Sorted array is ' . "\n";

    printArray($arr , $n);

// This code is contributed by Shivi_Aggarwal
?>
```

Output:

```
Sorted array is
```

```
5 6 7 11 12 13
```

Here is previous C code for reference.

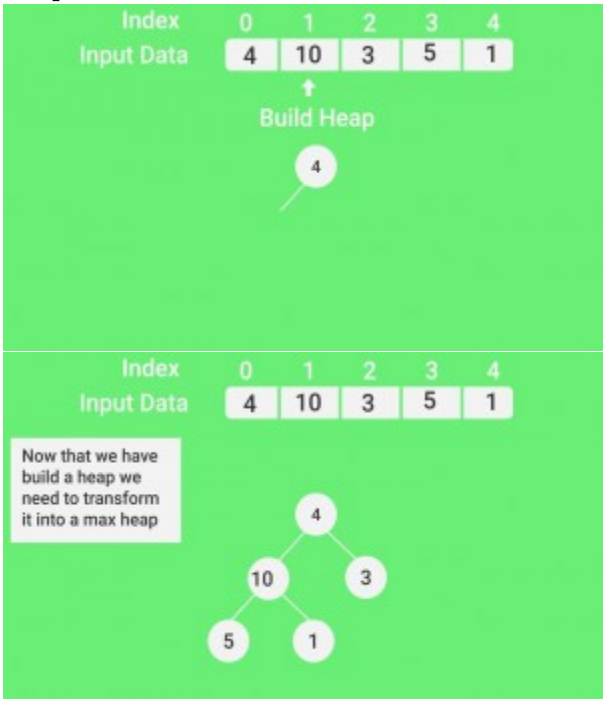
Notes:
Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable (See this)

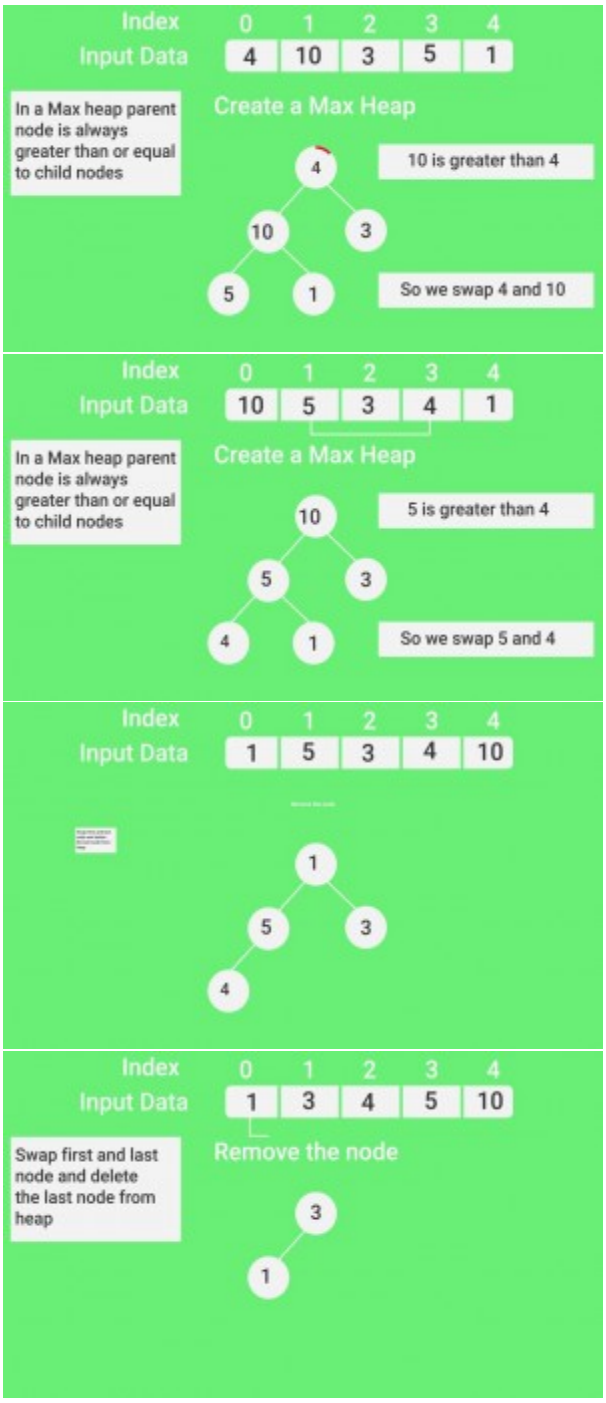
Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort
1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See Applications of Heap Data Structure

Snapshots:





Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:
QuickSort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Shivi_Aggarwal, Abby_akku

Source

https://www.geeksforgeeks.org/heap-sort/

Chapter 19

Height of a complete binary tree (or Heap) with N nodes

Height of a complete binary tree (or Heap) with N nodes - GeeksforGeeks

Consider a Binary Heap of size N. We need to find height of it.

Examples :

```
Input : N = 6
Output : 2
        ()
      /    \
     ()     ()
    /  \    /
  ()    () ()

Input : N = 9
Output :
        ()
      /    \
     ()     ()
    /  \    /  \
  ()    () ()   ()
 / \
() ()
```

Let the size of heap be **N** and height be **h**
If we take few examples, we can notice that the value of h in a complete binary tree is ceil($\log_2$(N+1)) – 1.
Examples :

```
 N    h
---------
 1    0
 2    1
 3    1
 4    2
 5    2
 .....
 .....
```

C++

```
 // CPP program to find height of complete
// binary tree from total nodes.
#include <bits/stdc++.h>
using namespace std;

int height(int N)
{
    return ceil(log2(N + 1)) - 1;
}

// driver node
int main()
{
    int N = 6;
    cout << height(N);
    return 0;
}
```

Java

```
 // Java program to find height
// of complete binary tree
// from total nodes.
import java.lang.*;

class GFG {

    // Function to calculate height
    static int height(int N)
    {
        return (int)Math.ceil(Math.log(N +
                    1) / Math.log(2)) - 1;
    }

    // Driver Code
```

```
    public static void main(String[] args)
    {
        int N = 6;
        System.out.println(height(N));
    }
}

// This code is contributed by
// Smitha Dinesh Semwal
```

Python 3

```
 # Python 3 program to find
# height of complete binary
# tree from total nodes.
import math
def height(N):
    return math.ceil(math.log2(N + 1)) - 1

# driver node
N = 6
print(height(N))

# This code is contributed by
# Smitha Dinesh Semwal
```

C#

```
 // C# program to find height
// of complete binary tree
// from total nodes.
using System;

class GFG {
    static int height(int N)
    {
        return (int)Math.Ceiling(Math.Log(N
                   + 1) / Math.Log(2)) - 1;
    }

    // Driver node
    public static void Main()
    {
        int N = 6;
        Console.Write(height(N));
    }
}
```

```
// This code is contributed by
// Smitha Dinesh Semwal
```

PHP

```
 <?php
// PHP program to find height
// of complete binary tree
// from total nodes.

function height($N)
{
    return ceil(log($N + 1, 2)) - 1;
}

// Driver Code
$N = 6;
echo height($N);

// This code is contributed by aj_36
?>
```

Output :

```
2
```

Improved By : Smitha Dinesh Semwal, jit_t

Source

https://www.geeksforgeeks.org/height-complete-binary-tree-heap-n-nodes/

Chapter 20

How to check if a given array represents a Binary Heap?

How to check if a given array represents a Binary Heap? - GeeksforGeeks

Given an array, how to check if the given array represents a Binary Max-Heap.

Examples:

```
Input:  arr[] = {90, 15, 10, 7, 12, 2}
Output: True
The given array represents below tree
       90
     /    \
   15      10
  /  \     /
 7    12  2
The tree follows max-heap property as every
node is greater than all of its descendants.

Input:  arr[] = {9, 15, 10, 7, 12, 11}
Output: False
The given array represents below tree
       9
     /    \
   15      10
  /  \     /
 7    12  11
The tree doesn't follows max-heap property 9 is
smaller than 15 and 10, and 10 is smaller than 11.
```

A **Simple Solution** is to first check root, if it's greater than all of its descendants. Then check for children of root. Time complexity of this solution is $O(n^2)$

An **Efficient Solution** is to compare root only with its children (not all descendants), if root is greater than its children and same is true for for all nodes, then tree is max-heap (This conclusion is based on transitive property of > operator, i.e., if x > y and y > z, then x > z).

The last internal node is present at index (2n-2)/2 assuming that indexing begins with 0.

Below is C++ implementation of this solution.

```
 // C program to check whether a given array
// represents a max-heap or not
#include <stdio.h>
#include <limits.h>

// Returns true if arr[i..n-1] represents a
// max-heap
bool isHeap(int arr[], int i, int n)
{
   // If a leaf node
   if (i > (n - 2)/2)
       return true;

   // If an internal node and is greater than its children, and
   // same is recursively true for the children
   if (arr[i] >= arr[2*i + 1]  &&  arr[i] >= arr[2*i + 2] &&
       isHeap(arr, 2*i + 1, n) && isHeap(arr, 2*i + 2, n))
       return true;

   return false;
}

// Driver program
int main()
{
    int arr[] = {90, 15, 10, 7, 12, 2, 7, 3};
    int n = sizeof(arr) / sizeof(int);

    isHeap(arr, 0, n)? printf("Yes"): printf("No");

    return 0;
}
```

Output:

```
Yes
```

Time complexity of this solution is O(n). The solution is similar to preorder traversal of Binary Tree.

Thanks to Utkarsh Trivedi for suggesting the above solution.

An **Iterative Solution** is to traverse all internal nodes and check id node is greater than its children or not.

```
 // C program to check whether a given array
// represents a max-heap or not
#include <stdio.h>
#include <limits.h>

// Returns true if arr[i..n-1] represents a
// max-heap
bool isHeap(int arr[],  int n)
{
    // Start from root and go till the last internal
    // node
    for (int i=0; i<=(n-2)/2; i++)
    {
        // If left child is greater, return false
        if (arr[2*i +1] > arr[i])
                return false;

        // If right child is greater, return false
        if (arr[2*i+2] > arr[i])
                return false;
    }
    return true;
}

// Driver program
int main()
{
    int arr[] = {90, 15, 10, 7, 12, 2, 7, 3};
    int n = sizeof(arr) / sizeof(int);

    isHeap(arr, n)? printf("Yes"): printf("No");

    return 0;
}
```

Output:

```
Yes
```

Thanks to Himanshu for suggesting this solution.

Source

https://www.geeksforgeeks.org/how-to-check-if-a-given-array-represents-a-binary-heap/

Chapter 21

How to implement stack using priority queue or heap?

How to implement stack using priority queue or heap? - GeeksforGeeks

How to Implement stack using a priority queue(using min heap)?.

Asked In: Microsoft, Adobe.

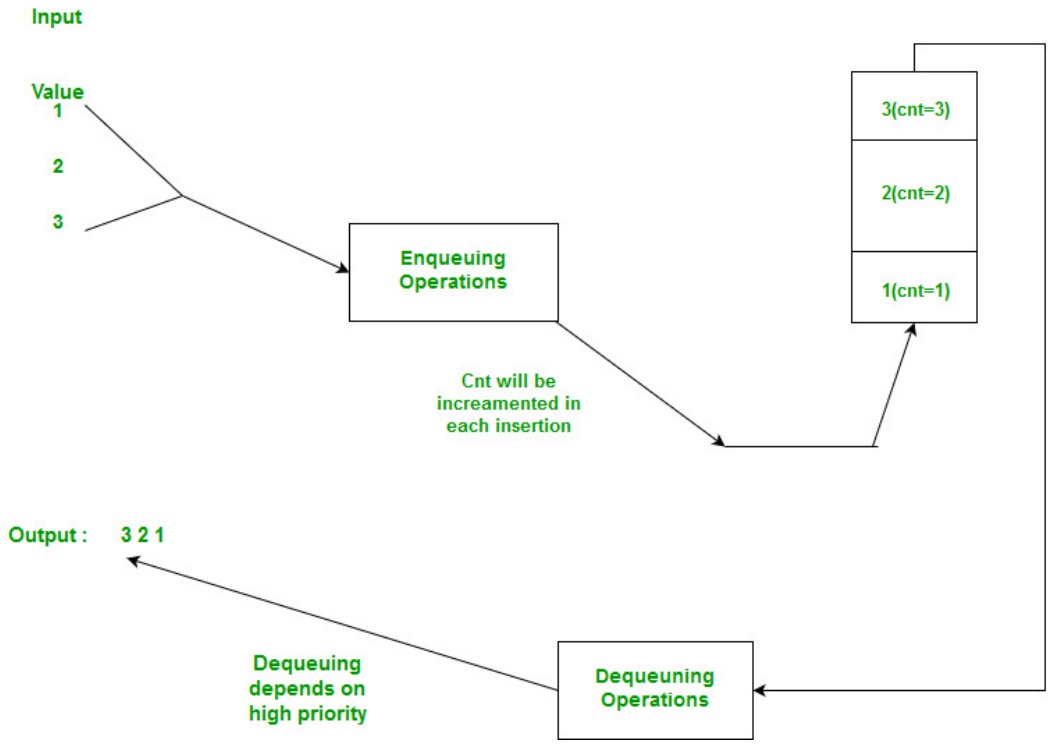
Solution:

In priority queue, we assign priority to the elements that are being pushed. A stack requires elements to be processed in Last in First Out manner. The idea is to associate a count that determines when it was pushed. This count works as a key for the priority queue.

So the implementation of stack uses a priority queue of pairs, with the first element serving as the key.

```
pair <int, int> (key, value)
```

See Below Image to understand Better



Below is C++ implementation of the idea.

```
 // C++ program to implement a stack using
// Priority queue(min heap)
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi;

// User defined stack class
class Stack{

    // cnt is used to keep track of the number of
    //elements in the stack and also serves as key
    //for the priority queue.
    int cnt;
    priority_queue<pair<int, int> > pq;
public:
    Stack():cnt(0){}
    void push(int n);
    void pop();
    int top();
    bool isEmpty();
};
```

```
// push function increases cnt by 1 and
// inserts this cnt with the original value.
void Stack::push(int n){
    cnt++;
    pq.push(pi(cnt, n));
}

// pops element and reduces count.
void Stack::pop(){
    if(pq.empty()){ cout<<"Nothing to pop!!!";}
    cnt--;
    pq.pop();
}

// returns the top element in the stack using
// cnt as key to determine top(highest priority),
// default comparator for pairs works fine in this case
int Stack::top(){
    pi temp=pq.top();
    return temp.second;
}

// return true if stack is empty
bool Stack::isEmpty(){
    return pq.empty();
}

// Driver code
int main()
{
    Stack* s=new Stack();
    s->push(1);
    s->push(2);
    s->push(3);
    while(!s->isEmpty()){
        cout<<s->top()<<endl;
        s->pop();
    }
}
```

Output:

```
 3
 2
 1
```

Now, as we can see this implementation takes O(log n) time for both push and pop operations. This can be slightly optimized by using fibonacci heap implementation of priority queue which would give us O(1) time complexity for push operation, but pop still requires O(log n) time.

Source

https://www.geeksforgeeks.org/implement-stack-using-priority-queue-or-heap/

Chapter 22

Huffman Coding | Greedy Algo-3

Huffman Coding | Greedy Algo-3 - GeeksforGeeks

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

See this for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding
1) Build a Huffman Tree from input characters.
2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree
Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

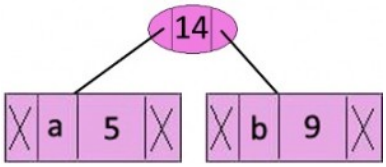
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

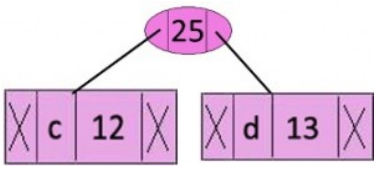
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

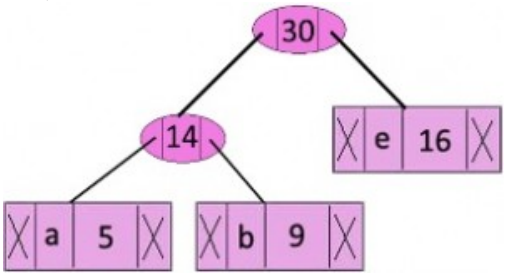
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

```
character           Frequency
Internal Node          14
       e               16
Internal Node          25
       f               45
```

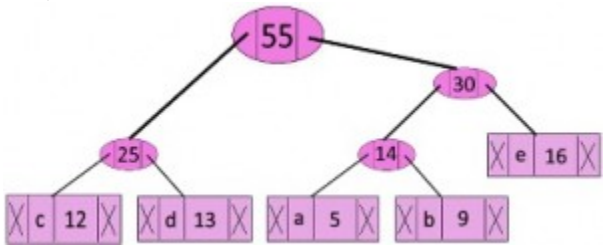
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

```
character          Frequency
Internal Node         25
Internal Node         30
      f               45
```

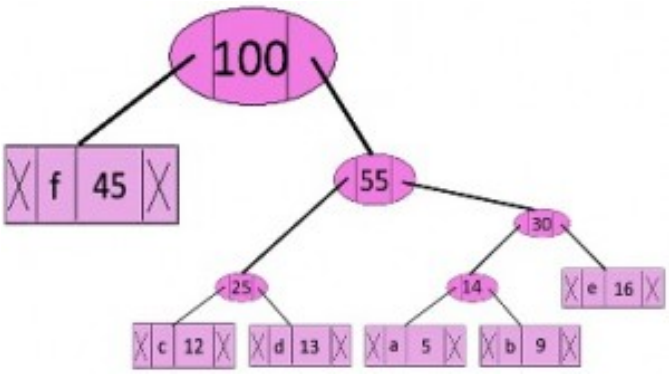
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

```
character     Frequency
       f         45
Internal Node    55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

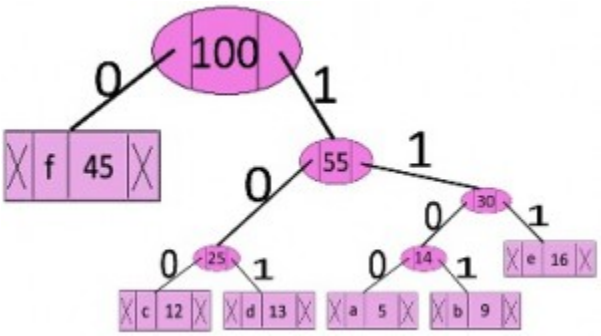


Now min heap contains only one node.

```
character       Frequency
Internal Node     100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character   code-word
    f          0
    c          100
    d          101
    a          1100
    b          1101
    e          111
```

C

```
 // C program for Huffman Coding
#include <stdio.h>
#include <stdlib.h>

// This constant can be avoided by explicitly
// calculating height of Huffman Tree
#define MAX_TREE_HT 100

// A Huffman tree node
struct MinHeapNode {

    // One of the input characters
    char data;

    // Frequency of the character
    unsigned freq;

    // Left and right child of this node
    struct MinHeapNode *left, *right;
};

// A Min Heap:  Collection of
// min heap (or Hufmman tree) nodes
struct MinHeap {

    // Current size of min heap
    unsigned size;

    // capacity of min heap
    unsigned capacity;

    // Attay of minheap node pointers
    struct MinHeapNode** array;
};

// A utility function allocate a new
// min heap node with given character
// and frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp
        = (struct MinHeapNode*)malloc
(sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
```

```
    return temp;
}

// A utility function to create
// a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)

{

    struct MinHeap* minHeap
        = (struct MinHeap*)malloc(sizeof(struct MinHeap));

    // current size is 0
    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array
        = (struct MinHeapNode**)malloc(minHeap->
capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to
// swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a,
                     struct MinHeapNode** b)

{

    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// 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->array[left]->
freq < minHeap->array[smallest]->freq)
        smallest = left;
```

```
    if (right < minHeap->size && minHeap->array[right]->
freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[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
struct MinHeapNode* extractMin(struct MinHeap* minHeap)

{

    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0]
        = minHeap->array[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,
                   struct MinHeapNode* minHeapNode)

{

    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

        minHeap->array[i] = minHeap->array[(i - 1) / 2];
```

```
        i = (i - 1) / 2;
    }

    minHeap->array[i] = minHeapNode;
}

// 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);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);

    printf("\n");
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root)

{

    return !(root->left) && !(root->right);
}

// Creates a min heap of capacity
// equal to size and inserts all character of
// data[] in min heap. Initially size of
// min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)

{

    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
```

```
    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)

{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity
    // equal to size.  Initially, there are
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {

        // Step 2: Extract the two minimum
        // freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3:  Create a new internal
        // node with frequency equal to the
        // sum of the two nodes frequencies.
        // Make the two extracted node as
        // left and right children of this new node.
        // Add this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the
    // root node and the tree is complete.
    return extractMin(minHeap);
}

// Prints huffman codes from the root of Huffman Tree.
// It uses arr[] to store codes
```

```
void printCodes(struct MinHeapNode* root, int arr[], int top)

{

    // Assign 0 to left edge and recur
    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {

        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then
    // it contains one of the input
    // characters, print the character
    // and its code from arr[]
    if (isLeaf(root)) {

        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// The main function that builds a
// Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)

{
    // Construct Huffman Tree
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using
    // the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
}

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

```
{

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

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

    HuffmanCodes(arr, freq, size);

    return 0;
}
```

C++ using STL

```
 // C++ program for Huffman Coding
#include <bits/stdc++.h>
using namespace std;

// A Huffman tree node
struct MinHeapNode {

    // One of the input characters
    char data;

    // Frequency of the character
    unsigned freq;

    // Left and right child
    MinHeapNode *left, *right;

    MinHeapNode(char data, unsigned freq)

    {

        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

// For comparison of
// two heap nodes (needed in min heap)
struct compare {

    bool operator()(MinHeapNode* l, MinHeapNode* r)

    {
        return (l->freq > r->freq);
```

```
    }
};

// Prints huffman codes from
// the root of Huffman Tree.
void printCodes(struct MinHeapNode* root, string str)
{

    if (!root)
        return;

    if (root->data != '$')
        cout << root->data << ": " << str << "\n";

    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

// The main function that builds a Huffman Tree and
// print codes by traversing the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    // Create a min heap & inserts all characters of data[]
    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

    // Iterate while size of heap doesn't become 1
    while (minHeap.size() != 1) {

        // Extract the two minimum
        // freq items from min heap
        left = minHeap.top();
        minHeap.pop();

        right = minHeap.top();
        minHeap.pop();

        // Create a new internal node with
        // frequency equal to the sum of the
        // two nodes frequencies. Make the
        // two extracted node as left and right children
        // of this new node. Add this node
        // to the min heap '$' is a special value
        // for internal nodes, not used
```

```
        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        minHeap.push(top);
    }

    // Print Huffman codes using
    // the Huffman tree built above
    printCodes(minHeap.top(), "");
}

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

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

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

    HuffmanCodes(arr, freq, size);

    return 0;
}

// This code is contributed by Aditya Goel
```

Java

```
 import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;

// node class is the basic structure
// of each node present in the huffman - tree.
class HuffmanNode {

    int data;
    char c;

    HuffmanNode left;
    HuffmanNode right;
}

// comparator class helps to compare the node
// on the basis of one of its attribute.
```

```
// Here we will be compared
// on the basis of data values of the nodes.
class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y)
    {

        return x.data - y.data;
    }
}

public class Huffman {

    // recursive function to print the
    // huffman-code through the tree traversal.
    // Here s is the huffman - code generated.
    public static void printCode(HuffmanNode root, String s)
    {

        // base case; if the left and right are null
        // then its a leaf node and we print
        // the code s generated by traversing the tree.
        if (root.left
                == null
            && root.right
                   == null
            && Character.isLetter(root.c)) {

            // c is the character in the node
            System.out.println(root.c + ":" + s);

            return;
        }

        // if we go to left then add "0" to the code.
        // if we go to the right add"1" to the code.

        // recursive calls for left and
        // right sub-tree of the generated tree.
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }

    // main function
    public static void main(String[] args)
    {

        Scanner s = new Scanner(System.in);
```

```
// number of characters.
int n = 6;
char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
int[] charfreq = { 5, 9, 12, 13, 16, 45 };

// creating a priority queue q.
// makes a min-priority queue(min-heap).
PriorityQueue<HuffmanNode> q
    = new PriorityQueue<HuffmanNode>(n, new MyComparator());

for (int i = 0; i < n; i++) {

    // creating a huffman node object
    // and adding it to the priority-queue.
    HuffmanNode hn = new HuffmanNode();

    hn.c = charArray[i];
    hn.data = charfreq[i];

    hn.left = null;
    hn.right = null;

    // add functions adds
    // the huffman node to the queue.
    q.add(hn);
}

// create a root node
HuffmanNode root = null;

// Here we will extract the two minimum value
// from the heap each time until
// its size reduces to 1, extract until
// all the nodes are extracted.
while (q.size() > 1) {

    // first min extract.
    HuffmanNode x = q.peek();
    q.poll();

    // second min extarct.
    HuffmanNode y = q.peek();
    q.poll();

    // new node f which is equal
    HuffmanNode f = new HuffmanNode();

    // to the sum of the frequency of the two nodes
```

```
            // assigning values to the f node.
            f.data = x.data + y.data;
            f.c = '-';

            // first extracted node as left child.
            f.left = x;

            // second extracted node as the right child.
            f.right = y;

            // marking the f node as the root node.
            root = f;

            // add this node to the priority-queue.
            q.add(f);
        }

        // print the codes by traversing the tree
        printCode(root, "");
    }
}

// This code is contributed by Kunwar Desh Deepak Singh
```

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).

If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

Reference:
http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : kddeepak

Source

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Chapter 23

Huffman Decoding

Huffman Decoding - GeeksforGeeks

We have discussed Huffman Encoding in a previous post. In this post decoding is discussed.

Examples:

```
Input Data : AAAAAABCCCCCCDDEEEEE
Frequencies : A: 6, B: 1, C: 6, D: 2, E: 5
Encoded Data :
00000000000011001010101010111111101010101
Huffman Tree: '#' is the special character used
              for internal nodes as character field
              is not needed for internal nodes.
               #(20)
             /       \
        #(12)         #(8)
     /      \        /    \
    A(6)     C(6) E(5)     #(3)
                          /    \
                        B(1)    D(2)
Code of 'A' is '00', code of 'C' is '01', ..
Decoded Data : AAAAAABCCCCCCDDEEEEE
```

```
Input Data : GeeksforGeeks
Character With there Frequencies
e 10, f 1100, g 011, k 00, o 010, r 1101, s 111
Encoded Huffman data :
011101000111110001011010111010000111
Decoded Huffman Data
geeksforgeeks
```

To decode the encoded data we require the Huffman tree. We iterate through the binary encoded data. To find character corresponding to current bits, we use following simple steps.

1. We start from root and do following until a leaf is found.
2. If current bit is 0, we move to left node of the tree.
3. If the bit is 1, we move to right node of the tree.
4. If during traversal, we encounter a leaf node, we print character of that particular leaf node and then again continue the iteration of the encoded data starting from step 1.

The below code takes a string as input, it encodes it and save in a variable encodedString. Then it decodes it and print the original string.

The below code performs full Huffman Encoding and Decoding of a given input data.

```
 // C++ program to encode and decode a string using
// Huffman Coding.
#include <bits/stdc++.h>
#define MAX_TREE_HT 256
using namespace std;

// to map each character its huffman value
map<char, string> codes;

// to store the frequency of character of the input data
map<char, int> freq;

// A Huffman tree node
struct MinHeapNode
{
    char data;             // One of the input characters
    int freq;             // Frequency of the character
    MinHeapNode *left, *right; // Left and right child

    MinHeapNode(char data, int freq)
    {
        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

// utility function for the priority queue
struct compare
{
    bool operator()(MinHeapNode* l, MinHeapNode* r)
    {
        return (l->freq > r->freq);
    }
```

```
};

// utility function to print characters along with
// there huffman value
void printCodes(struct MinHeapNode* root, string str)
{
    if (!root)
        return;
    if (root->data != '$')
        cout << root->data << ": " << str << "\n";
    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

// utility function to store characters along with
// there huffman value in a hash table, here we
// have C++ STL map
void storeCodes(struct MinHeapNode* root, string str)
{
    if (root==NULL)
        return;
    if (root->data != '$')
        codes[root->data]=str;
    storeCodes(root->left, str + "0");
    storeCodes(root->right, str + "1");
}

// STL priority queue to store heap tree, with respect
// to their heap root node value
priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

// function to build the Huffman tree and store it
// in minHeap
void HuffmanCodes(int size)
{
    struct MinHeapNode *left, *right, *top;
    for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)
        minHeap.push(new MinHeapNode(v->first, v->second));
    while (minHeap.size() != 1)
    {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
```

```
    }
    storeCodes(minHeap.top(), "");
}

// utility function to store map each character with its
// frequency in input string
void calcFreq(string str, int n)
{
    for (int i=0; i<str.size(); i++)
        freq[str[i]]++;
}

// function iterates through the encoded string s
// if s[i]=='1' then move to node->right
// if s[i]=='0' then move to node->left
// if leaf node append the node->data to our output string
string decode_file(struct MinHeapNode* root, string s)
{
    string ans = "";
    struct MinHeapNode* curr = root;
    for (int i=0;i<s.size();i++)
    {
        if (s[i] == '0')
           curr = curr->left;
        else
           curr = curr->right;

        // reached leaf node
        if (curr->left==NULL and curr->right==NULL)
        {
            ans += curr->data;
            curr = root;
        }
    }
    // cout<<ans<<endl;
    return ans+'\0';
}

// Driver program to test above functions
int main()
{
    string str = "geeksforgeeks";
    string encodedString, decodedString;
    calcFreq(str, str.length());
    HuffmanCodes(str.length());
    cout << "Character With there Frequencies:\n";
    for (auto v=codes.begin(); v!=codes.end(); v++)
        cout << v->first <<' ' << v->second << endl;
```

```
    for (auto i: str)
        encodedString+=codes[i];

    cout << "\nEncoded Huffman data:\n" << encodedString << endl;

    decodedString = decode_file(minHeap.top(), encodedString);
    cout << "\nDecoded Huffman Data:\n" << decodedString << endl;
    return 0;
}
```

Output:

```
Character With there Frequencies
e 10
f 1100
g 011
k 00
o 010
r 1101
s 111

Encoded Huffman data
01110100011111000101101011101000111

Decoded Huffman Data
geeksforgeeks
```

Comparing Input file size and Output file size:
Comparing the input file size and the Huffman encoded output file. We can calculate the size of the output data in a simple way. Lets say our input is a string "geeksforgeeks" and is stored in a file input.txt.
Input File Size:

```
Input: "geeksforgeeks"
Total number of character i.e. input length: 13
Size: 13 character occurrences * 8 bits = 104 bits or 13 bytes.
```

Output File Size:

```
Input: "geeksforgeeks"
-----------------------------------------------
Character |  Frequency |  Binary Huffman Value |
```

```
-------------------------------------------------
   e      |      4     |         10             |
   f      |      1     |         1100           |
   g      |      2     |         011            |
   k      |      2     |         00             |
   o      |      1     |         010            |
   r      |      1     |         1101           |
   s      |      2     |         111            |
-------------------------------------------------

So to calculate output size:
e: 4 occurrences * 2 bits = 8 bits
f: 1 occurrence  * 4 bits = 4 bits
g: 2 occurrences * 3 bits = 6 bits
k: 2 occurrences * 2 bits = 4 bits
o: 1 occurrence  * 3 bits = 3 bits
r: 1 occurrence  * 4 bits = 4 bits
s: 2 occurrences * 3 bits = 6 bits

Total Sum: 35 bits approx 5 bytes
```

Hence, we could see that after encoding the data we have saved a large amount of data. The above method can also help us to determine the value of N i.e. the length of the encoded data.

Source

https://www.geeksforgeeks.org/huffman-decoding/

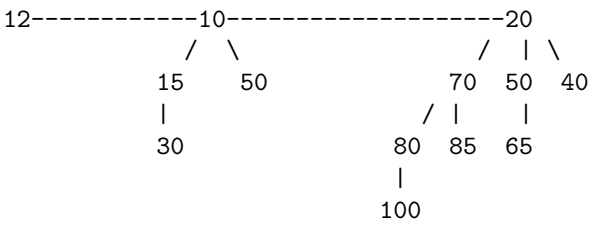
Chapter 24

Implementation of Binomial Heap

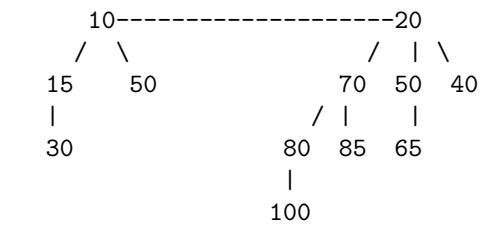
Implementation of Binomial Heap - GeeksforGeeks

In previous article, we have discussed about the concepts related to Binomial heap.

Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3
Binomial Trees of orders 0, 2 and 3 from left to right.



In this article, implementation of Binomial Heap is discussed. Following functions implemented :

1. **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
2. **getMin(H):** A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.
3. **extractMin(H):** This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

```
 // C++ program to implement different operations
// on Binomial Heap
#include<bits/stdc++.h>
using namespace std;

// A Binomial Tree node.
struct Node
{
    int data, degree;
    Node *child, *sibling, *parent;
};

Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->degree = 0;
    temp->child = temp->parent = temp->sibling = NULL;
    return temp;
}

// This function merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
    // Make sure b1 is smaller
    if (b1->data > b2->data)
        swap(b1, b2);

    // We basically make larger valued tree
    // a child of smaller valued tree
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;

    return b1;
}
```

```
// This function perform union operation on two
// binomial heap i.e. l1 & l2
list<Node*> unionBionomialHeap(list<Node*> l1,
                               list<Node*> l2)
{
    // _new to another binomial heap which contain
    // new heap after merging l1 & l2
    list<Node*> _new;
    list<Node*>::iterator it = l1.begin();
    list<Node*>::iterator ot = l2.begin();
    while (it!=l1.end() && ot!=l2.end())
    {
        // if D(l1) <= D(l2)
        if((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }
        // if D(l1) > D(l2)
        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }

    // if there remains some elements in l1
    // binomial heap
    while (it != l1.end())
    {
        _new.push_back(*it);
        it++;
    }

    // if there remains some elements in l2
    // binomial heap
    while (ot!=l2.end())
    {
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}

// adjust function rearranges the heap so that
// heap is in increasing order of degree and
// no two binomial trees have same degree in this heap
list<Node*> adjust(list<Node*> _heap)
```

```
{
    if (_heap.size() <= 1)
        return _heap;
    list<Node*> new_heap;
    list<Node*>::iterator it1,it2,it3;
    it1 = it2 = it3 = _heap.begin();

    if (_heap.size() == 2)
    {
        it2 = it1;
        it2++;
        it3 = _heap.end();
    }
    else
    {
        it2++;
        it3=it2;
        it3++;
    }
    while (it1 != _heap.end())
    {
        // if only one element remains to be processed
        if (it2 == _heap.end())
            it1++;

        // If D(it1) < D(it2) i.e. merging of Binomial
        // Tree pointed by it1 & it2 is not possible
        // then move next in heap
        else if ((*it1)->degree < (*it2)->degree)
        {
            it1++;
            it2++;
            if(it3!=_heap.end())
                it3++;
        }

        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are same
        // in heap
        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
            it1++;
            it2++;
            it3++;
        }
```

```
        // if degree of two Binomial Tree are same in heap
        else if ((*it1)->degree == (*it2)->degree)
        {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1,*it2);
            it2 = _heap.erase(it2);
            if(it3 != _heap.end())
                it3++;
        }
    }
    return _heap;
}

// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
                              Node *tree)
{
    // creating a new heap i.e temp
    list<Node*> temp;

    // inserting Binomial Tree into heap
    temp.push_back(tree);

    // perform union operation to finally insert
    // Binomial Tree in original heap
    temp = unionBionomialHeap(_heap,temp);

    return adjust(temp);
}

// removing minimum key element from binomial heap
// this function take Binomial Tree as input and return
// binomial heap after
// removing head of that tree i.e. minimum element
list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;

    // making a binomial heap from Binomial Tree
    while (temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
```

```
    return heap;
}

// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
    Node *temp = newNode(key);
    return insertATreeInHeap(_head,temp);
}

// return pointer of minimum value Node
// present in the binomial heap
Node* getMin(list<Node*> _heap)
{
    list<Node*>::iterator it = _heap.begin();
    Node *temp = *it;
    while (it != _heap.end())
    {
        if ((*it)->data < temp->data)
            temp = *it;
        it++;
    }
    return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
    list<Node*> new_heap,lo;
    Node *temp;

    // temp contains the pointer of minimum value
    // element in heap
    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        if (*it != temp)
        {
            // inserting all Binomial Tree into new
            // binomial heap except the Binomial Tree
            // contains minimum element
            new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBionomialHeap(new_heap,lo);
```

```
    new_heap = adjust(new_heap);
    return new_heap;
}

// print function for Binomial Tree
void printTree(Node *h)
{
    while (h)
    {
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
    }
}

// print function for binomial heap
void printHeap(list<Node*> _heap)
{
    list<Node*> ::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        printTree(*it);
        it++;
    }
}


// Driver program to test above functions
int main()
{
    int ch,key;
    list<Node*> _heap;

    // Insert data in the heap
    _heap = insert(_heap,10);
    _heap = insert(_heap,20);
    _heap = insert(_heap,30);

    cout << "Heap elements after insertion:\n";
    printHeap(_heap);

    Node *temp = getMin(_heap);
    cout << "\nMinimum element of heap "
         << temp->data << "\n";

    // Delete minimum element of heap
    _heap = extractMin(_heap);
```

```
    cout << "Heap after deletion of minimum element\n";
    printHeap(_heap);

    return 0;
}
```

Output:

```
The heap is:
50 10 30 40 20
After deleing 10, the heap is:
20 30 40 50
```

Source

https://www.geeksforgeeks.org/implementation-binomial-heap/

Chapter 25

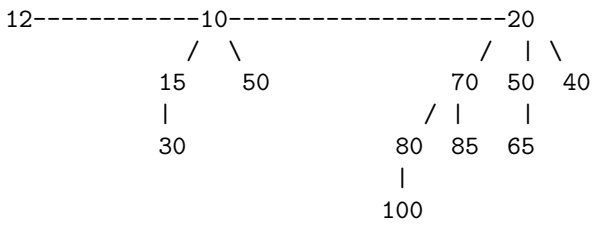
Implementation of Binomial Heap | Set – 2 (delete() and decreseKey())

Implementation of Binomial Heap | Set - 2 (delete() and decreseKey()) - GeeksforGeeks

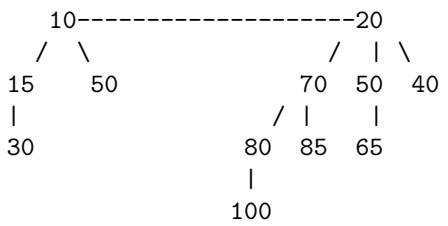
In previous post i.e. Set 1 we have discussed that implements these below functions:

1. **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
2. **getMin(H):** A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires O(Logn) time. It can be optimized to O(1) by maintaining a pointer to minimum key root.
3. **extractMin(H):** This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires O(Logn) time.

Examples:



A Binomial Heap with 13 nodes. It is a collection of 3
Binomial Trees of orders 0, 2 and 3 from left to right.



In this post, below functions are implemented.

1. **delete(H):** Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().
2. **decreaseKey(H):** decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is O(Logn)

```
 // C++ program for implementation of
// Binomial Heap and Operations on it
#include <bits/stdc++.h>
using namespace std;

// Structure of Node
struct Node
{
    int val, degree;
    Node *parent, *child, *sibling;
};

// Making root global to avoid one extra
// parameter in all functions.
Node *root = NULL;

// link two heaps by making h1 a child
// of h2.
int binomialLink(Node *h1, Node *h2)
{
    h1->parent = h2;
    h1->sibling = h2->child;
    h2->child = h1;
    h2->degree = h2->degree + 1;
}

// create a Node
Node *createNode(int n)
{
```

```
    Node *new_node = new Node;
    new_node->val = n;
    new_node->parent = NULL;
    new_node->sibling = NULL;
    new_node->child = NULL;
    new_node->degree = 0;
    return new_node;
}

// This function merge two Binomial Trees
Node *mergeBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;

    // define a Node
    Node *res = NULL;

    // check degree of both Node i.e.
    // which is greater or smaller
    if (h1->degree <= h2->degree)
        res = h1;

    else if (h1->degree > h2->degree)
        res = h2;

    // traverse till if any of heap gets empty
    while (h1 != NULL && h2 != NULL)
    {
        // if degree of h1 is smaller, increment h1
        if (h1->degree < h2->degree)
            h1 = h1->sibling;

        // Link h1 with h2 in case of equal degree
        else if (h1->degree == h2->degree)
        {
            Node *sib = h1->sibling;
            h1->sibling = h2;
            h1 = sib;
        }

        // if h2 is greater
        else
        {
            Node *sib = h2->sibling;
            h2->sibling = h1;
```

```
            h2 = sib;
        }
    }
    return res;
}

// This function perform union operation on two
// binomial heap i.e. h1 & h2
Node *unionBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL && h2 == NULL)
       return NULL;

    Node *res = mergeBHeaps(h1, h2);

    // Traverse the merged list and set
    // values according to the degree of
    // Nodes
    Node *prev = NULL, *curr = res,
         *next = curr->sibling;
    while (next != NULL)
    {
        if ((curr->degree != next->degree) ||
                ((next->sibling != NULL) &&
                 (next->sibling)->degree ==
                 curr->degree))
        {
            prev = curr;
            curr = next;
        }

        else
        {
            if (curr->val <= next->val)
            {
                curr->sibling = next->sibling;
                binomialLink(next, curr);
            }
            else
            {
                if (prev == NULL)
                    res = next;
                else
                    prev->sibling = next;
                binomialLink(curr, next);
                curr = next;
            }
        }
```

```
        next = curr->sibling;
    }
    return res;
}

// Function to insert a Node
void binomialHeapInsert(int x)
{
    // Create a new node and do union of
    // this node with root
    root = unionBHeaps(root, createNode(x));
}

// Function to display the Nodes
void display(Node *h)
{
    while (h)
    {
        cout << h->val << " ";
        display(h->child);
        h = h->sibling;
    }
}

// Function to reverse a list
// using recursion.
int revertList(Node *h)
{
    if (h->sibling != NULL)
    {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
    }
    else
        root = h;
}

// Function to extract minimum value
Node *extractMinBHeap(Node *h)
{
    if (h == NULL)
       return NULL;

    Node *min_node_prev = NULL;
    Node *min_node = h;

    // Find minimum value
    int min = h->val;
```

```
    Node *curr = h;
    while (curr->sibling != NULL)
    {
        if ((curr->sibling)->val < min)
        {
            min = (curr->sibling)->val;
            min_node_prev = curr;
            min_node = curr->sibling;
        }
        curr = curr->sibling;
    }

    // If there is a single Node
    if (min_node_prev == NULL &&
        min_node->sibling == NULL)
        h = NULL;

    else if (min_node_prev == NULL)
        h = min_node->sibling;

    // Remove min node from list
    else
        min_node_prev->sibling = min_node->sibling;

    // Set root (which is global) as children
    // list of min node
    if (min_node->child != NULL)
    {
        revertList(min_node->child);
        (min_node->child)->sibling = NULL;
    }

    // Do union of root h and children
    return unionBHeaps(h, root);
}

// Function to search for an element
Node *findNode(Node *h, int val)
{
    if (h == NULL)
      return NULL;

     // check if key is equal to the root's data
    if (h->val == val)
        return h;

    // Recur for child
    Node *res = findNode(h->child, val);
```

```
    if (res != NULL)
       return res;

    return findNode(h->sibling, val);
}

// Function to decrease the value of old_val
// to new_val
void decreaseKeyBHeap(Node *H, int old_val,
                              int new_val)
{
    // First check element present or not
    Node *node = findNode(H, old_val);

    // return if Node is not present
    if (node == NULL)
        return;

    // Reduce the value to the minimum
    node->val = new_val;
    Node *parent = node->parent;

    // Update the heap according to reduced value
    while (parent != NULL && node->val < parent->val)
    {
        swap(node->val, parent->val);
        node = parent;
        parent = parent->parent;
    }
}

// Function to delete an element
Node *binomialHeapDelete(Node *h, int val)
{
    // Check if heap is empty or not
    if (h == NULL)
        return NULL;

    // Reduce the value of element to minimum
    decreaseKeyBHeap(h, val, INT_MIN);

    // Delete the minimum element from heap
    return extractMinBHeap(h);
}

// Driver code
int main()
{
```

```
    // Note that root is global
    binomialHeapInsert(10);
    binomialHeapInsert(20);
    binomialHeapInsert(30);
    binomialHeapInsert(40);
    binomialHeapInsert(50);

    cout << "The heap is:\n";
    display(root);

    // Delete a particular element from heap
    root = binomialHeapDelete(root, 10);

    cout << "\nAfter deleing 10, the heap is:\n";

    display(root);

    return 0;
}
```

Output:

```
The heap is:
50 10 30 40 20
After deleing 10, the heap is:
20 30 40 50
```

Source

https://www.geeksforgeeks.org/implementation-binomial-heap-set-2/

Chapter 26

Iterative HeapSort

Iterative HeapSort - GeeksforGeeks

HeapSort is a comparison based sorting technique where we first build Max Heap and then swaps the root element with last element (size times) and maintains the heap property each time to finally make it sorted.

Examples:

```
Input :  10 20 15 17 9 21
Output : 9 10 15 17 20 21

Input:  12 11 13 5 6 7 15 5 19
Output: 5 5 6 7 11 12 13 15 19
```

In first Example, first we have to build Max Heap.
So, we will start from 20 as child and check for its parent. Here 10 is smaller, so we will swap these two.
Now, 20 10 15 17 9 21
Now, child 17 is greater than its parent 10. So, both will be swapped and order will be
20 17 15 10 9 21
Now, child 21 is greater than parent 15. So, both will be swapped.
20 17 21 10 9 15
Now, again 21 is bigger than parent 20. So,
21 17 20 10 9 15
This is Max Heap.

Now, we have to apply sorting. Here, we have to swap first element with last one and we have to maintain Max Heap property.
So, after first swapping : 15 17 20 10 9 21
It clearly violates Max Heap property. So, we have to maintain it. So, order will be
20 17 15 10 9 21
17 10 15 9 20 21

15 10 9 17 20 21
10 9 15 17 20 21
9 10 15 17 20 21
Here, underlined part is sorted part.

```
 // C++ program for implementation
// of Iterative Heap Sort
#include <bits/stdc++.h>
using namespace std;

// function build Max Heap where value
// of each child is always smaller
// than value of their parent
void buildMaxHeap(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        // if child is bigger than parent
        if (arr[i] > arr[(i - 1) / 2])
        {
            int j = i;

            // swap child and parent until
            // parent is smaller
            while (arr[j] > arr[(j - 1) / 2])
            {
                swap(arr[j], arr[(j - 1) / 2]);
                j = (j - 1) / 2;
            }
        }
    }
}

void heapSort(int arr[], int n)
{
    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--)
    {
        // swap value of first indexed
        // with last indexed
        swap(arr[0], arr[i]);

        // maintaining heap property
        // after each swapping
        int j = 0, index;

        do
```

```
        {
            index = (2 * j + 1);

            // if left child is smaller than
            // right child point index variable
            // to right child
            if (arr[index] < arr[index + 1] &&
                                index < (i - 1))
                index++;

            // if parent is smaller than child
            // then swapping parent with child
            // having higher value
            if (arr[j] < arr[index] && index < i)
                swap(arr[j], arr[index]);

            j = index;

        } while (index < i);
    }
}

// Driver Code to test above
int main()
{
    int arr[] = {10, 20, 15, 17, 9, 21};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Given array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\n\n");

    heapSort(arr, n);

    // print array after sorting
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output :

```
Given array: 10 20 15 17 9 21
```

```
Sorted array: 9 10 15 17 20 21
```

Here, both function buildMaxHeap and heapSort runs in O(nlogn) time. So, overall time complexity is O(nlogn)

Source

https://www.geeksforgeeks.org/iterative-heap-sort/

Chapter 27

Job Selection Problem – Loss Minimization Strategy | Set 2

Job Selection Problem - Loss Minimization Strategy | Set 2 - GeeksforGeeks

We have discussed one loss minimization strategy before: Job Sequencing Problem – Loss Minimization. In this article, we will look at another strategy that applies to a slightly different problem.

We are given a sequence of N goods of production numbered from 1 to N. Each good has a volume denoted by (Vi). The constraint is that once a good has been completed its volume starts decaying at a fixed percentage (P) per day. All goods decay at the same rate and further each good take one day to complete.

We are required to find the order in which the goods should be produced so that overall volume of goods is maximized.

Example-1:

```
Input: 4, 2, 151, 15, 1, 52, 12 and P = 10%
Output: 222.503
```

Solution: In the optimum sequence of jobs, the total volume of goods left at the end of all jobs is 222.503

Example-2:

```
Input: 3, 1, 41, 52, 15, 4, 1, 63, 12 and P = 20%
Output: 145.742
```

Solution: In the optimum sequence of jobs the total volume of goods left at the end of all jobs is 145.72

Explanation –
Since this is an optimization problem, we can try to solve this problem by using a greedy algorithm. On each day we make a selection from among the goods that are yet to be produced. Thus all we need is a local selection criteria or heuristic, which when applied to select the jobs will give us the optimum result.

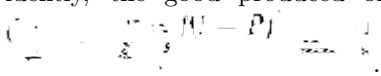
Instead of trying to maximize the volume, we can also try to minimize the losses. Since the total volume that can be obtained from all goods is also constant, if we minimize the losses we are guaranteed to get the optimum answer.

Now consider any good having volume V
Loss after Day 1: PV
Loss after Day 2: PV + P(1-P)V or V(2P-P^2)
Loss after Day 3: V(2P-P^2) + P(1 – 2P + P^2)V or V(3P – 3P^2 + P^3)

As the day increases the losses too increase. So the trick would be to ensure that the goods are not kept idle after production. Further, since we are required to produce at least one job per day, we should perform low volume jobs, and then perform the high volume jobs. This strategy works due to two factors.

1. High Volume goods are not kept idle after production.
2. As the volume decreases the loss per day too decreases, so for low volume goods the losses become negligible after a few days.

So in order to obtain the optimum solution we produce the larger volume goods later on. For the first day select the good with least volume and produce it. Remove the produced good from the list of goods. For the next day repeat the same. Keep repeating while there are goods left to be produced.

When calculating the total volume at the end of production, keep in mind the the good produced on day i, will have  times its volume left. Evidently, the good produced on day N (last day) will have its volume intact since  .

Algorithm –

```
Step 1: Add all the goods to a min-heap
Step 2: Repeat following steps while Queue is not empty
        Extract the good  at the head of the heap
        Print the good
        Remove the good from the heap
        [END OF LOOP]
Step 4: End
```

Complexity –
We perform exactly N push() and pop() operations each of which takes log (N) time. Hence time complexity is O(N * log(N)).

Below is the Cpp implementation of the solution.

```
 #include <bits/stdc++.h>
using namespace std;

void optimum_sequence_jobs(vector<int>& V, double P)
{
    int j = 1, N = V.size() - 1;
    double result = 0;

    // Create a min-heap (priority queue)
    priority_queue<int, vector<int>, greater<int> > Queue;

    // Add all goods to the the Queue
    for (int i = 1; i <= N; i++)
        Queue.push(V[i]);

    // Pop Goods from Queue as long as it is not empty
    while (!Queue.empty()) {

        // Print the good
        cout << Queue.top() << " ";

        // Add the Queue to the vector
        // so that total voulme can be calculated
        V[j++] = Queue.top();
        Queue.pop();
    }

    // Calclulating volume of goods left when all
    // are produced. Move from right to left of
    // sequence multiplying each volume by
    // increasing powers of 1 - P starting from 0
    for (int i = N; i >= 1; i--)
        result += pow((1 - P), N - i) * V[i];

    // Print result
    cout << endl << result << endl;
}

// Driver code
int main()
{
    // For implementation simplicity days are numbered
    // from 1 to N. Hence 1 based indexing is used
    vector<int> V{ -1, 3, 5, 4, 1, 2, 7, 6, 8, 9, 10 };

    // 10% loss per day
```

```
    double P = 0.10;

    optimum_sequence_jobs(V, P);

    return 0;
}
```

Output –

```
1 2 3 4 5 6 7 8 9 10
41.3811
```

Source

https://www.geeksforgeeks.org/job-selection-problem-loss-minimization-strategy-set-2/

Chapter 28

K maximum sum combinations from two arrays

K maximum sum combinations from two arrays - GeeksforGeeks

Given two equally sized arrays (A, B) and N (size of both arrays).
A **sum combination** is made by adding one element from array A and another element of array B. Display the **maximum K valid sum combinations** from all the possible sum combinations.

Examples:

```
Input :  A[] : {3, 2}
         B[] : {1, 4}
         K : 2 [Number of maximum sum
               combinations to be printed]
Output : 7    // (A : 3) + (B : 4)
         6    // (A : 2) + (B : 4)

Input :  A[] : {4, 2, 5, 1}
         B[] : {8, 0, 3, 5}
         K : 3
Output : 13   // (A : 5) + (B : 8)
         12   // (A : 4) + (B :  8)
         10   // (A : 2) + (B : 8)
```

Approach 1 (Naive Algorithm) :
We can use Brute force through all the possible combinations that can be made by taking one element from array A and another from array B and inserting them to a max heap. In a max heap maximum element is at the root node so whenever we pop from max heap we get the maximum element present in the heap. After inserting all the sum combinations we take out N elements from max heap and display it.

Below is the implementation of the above approach.

C++

```
 // A simple C++ program to find N maximum
// combinations from two arrays,
#include <bits/stdc++.h>
using namespace std;

// function to display first N maximum sum
// combinations
void NMaxCombinations(int A[], int B[], int N,
                                        int K)
{
    // max heap.
    priority_queue<int> pq;

    // insert all the possible combinations
    // in max heap.
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            pq.push(A[i] + B[j]);

    // pop first N elements from max heap
    // and display them.
    int count = 0;
    while (count < K) {
        cout << pq.top() << endl;
        pq.pop();
        count++;
    }
}

// Driver Code.
int main()
{
    int A[] = { 4, 2, 5, 1 };
    int B[] = { 8, 0, 5, 3 };
    int N = sizeof(A)/sizeof(A[0]);
    int K = 3;
    NMaxCombinations(A, B, N, K);
    return 0;
}
```

Java

```
 // Java program to find N
```

```
// maximum combinations
// from two arrays,
import java.io.*;
import java.util.*;

class GFG {

    // function to display first N
    // maximum sum combinations
    static void NMaxCombinations(int A[], int B[], int N,
                                                   int K)
    {
        // max heap.
        PriorityQueue<Integer> pq =
                            new PriorityQueue<Integer>(Collections.reverseOrder());

        // insert all the possible
        // combinations in max heap.
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                pq.add(A[i] + B[j]);

        // pop first N elements
        // from max heap and
        // display them.
        int count = 0;

        while (count < K)
        {
            System.out.println(pq.peek());
            pq.remove();
            count++;
        }
    }

    public static void main (String[] args)
    {
        int A[] = { 4, 2, 5, 1 };
        int B[] = { 8, 0, 5, 3 };
        int N = A.length;
        int K = 3;

        NMaxCombinations(A, B, N, K);

    }
}

// This code is contributed by Gitanjali.
```

Python 3

```
 # python program to find
# N maximum combinations
# from two arrays
import math
from Queue import PriorityQueue

# function to display first N
# maximum sum combinations
def NMaxCombinations( A, B, N, K):

    # max heap.
    pq = PriorityQueue()

    # insert all the possible
    # combinations in max heap.
    for i in range(0, N):
        for j in range(0, N):
           a = A[i] + B[j]
           pq.put((-a, a))

    # pop first N elements from
    # max heap and display them.
    count = 0
    while (count < K):
        print(pq.get()[1])
        count = count + 1

# Driver method
A = [ 4, 2, 5, 1 ]
B = [ 8, 0, 5, 3 ]
N = len(A)
K = 3
NMaxCombinations(A, B, N, K)

# This code is contributed
# by Gitanjali.
```

Output:

```
13
12
10
```

Time Complexity : O(N^2)

Approach 2 (Sorting, Max heap, Map) :
Instead of brute forcing through all the possible sum combinations we should find a way to limit our search space to possible candidate sum combinations.
1. Sort both arrays array A and array B.
2. Create a max heap i.e priority_queue in C++ to store the sum combinations along with the indices of elements from both arrays A and B which make up the sum. Heap is ordered by the sum.
3. Initialize the heap with the maximum possible sum combination i.e (A[N – 1] + B[N – 1] where N is the size of array) and with the indices of elements from both arrays (N – 1, N – 1). The tuple inside max heap will be (A[N-1] + B[N – 1], N – 1, N – 1). Heap is ordered by first value i.e sum of both elements.
4. Pop the heap to get the current largest sum and along with the indices of the element that make up the sum. Let the tuple be (sum, i, j).
4.1. Next insert (A[i – 1] + B[j], i – 1, j) and (A[i] + B[j – 1], i, j – 1) into the max heap but make sure that the pair of indices i.e (i – 1, j) and (i, j – 1) are not
already present in the max heap. To check this we can use set in C++.
4.2 Go back to 4 until K times.

```
 // An efficient C++ program to find top K elements
// from two arrays.
#include <bits/stdc++.h>
using namespace std;

// Function returns a vector containing N maximum
// sum combinations.
vector<int> NMaxCombinations(vector<int>& A,
                      vector<int>& B, int K)
{
    // sort both arrays A and B
    sort(A.begin(), A.end());
    sort(B.begin(), B.end());

    int N = A.size();

    // Max heap which contains tuple of the format
    // (sum, (i, j)) i and j are the indices
    // of the elements from array A
    // and array B which make up the sum.
    priority_queue<pair<int, pair<int, int> > > pq;

    // my_set is used to store the indices of
    // the  pair(i, j) we use my_set to make sure
    // the indices doe not repeat inside max heap.
    set<pair<int, int> > my_set;

    // initialize the heap with the maximum sum
    // combination ie (A[N - 1] + B[N - 1])
    // and also push indices (N - 1, N - 1) along
```

```
    // with sum.
    pq.push(make_pair(A[N - 1] + B[N - 1],
                      make_pair(N-1, N-1)));

    my_set.insert(make_pair(N - 1, N - 1));

    // iterate upto K
    for (int count=0; count<K; count++) {

        // tuple format (sum, i, j).
        pair<int, pair<int, int> > temp = pq.top();
        pq.pop();

        cout << temp.first << endl;

        int i = temp.second.first;
        int j = temp.second.second;

        int sum = A[i - 1] + B[j];

        // insert (A[i - 1] + B[j], (i - 1, j))
        // into max heap.
        pair<int, int> temp1 = make_pair(i - 1, j);

        // insert only if the pair (i - 1, j) is
        // not already present inside the map i.e.
        // no repeating pair should be present inside
        // the heap.
        if (my_set.find(temp1) == my_set.end()) {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }

        // insert (A[i] + B[j - 1], (i, j - 1))
        // into max heap.
        sum = A[i] + B[j - 1];
        temp1 = make_pair(i, j - 1);

        // insert only if the pair (i, j - 1)
        // is not present inside the heap.
        if (my_set.find(temp1) == my_set.end()) {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }
    }
}

// Driver Code.
```

```
int main()
{
    vector<int> A = { 1, 4, 2, 3 };
    vector<int> B = { 2, 5, 1, 6 };
    int K = 4;
    NMaxCombinations(A, B, K);
    return 0;
}
```

Output :

```
10
9
9
8
```

Time Complexity :
O(N log N) assuming K <= N

Improved By : nikhil741

Source

https://www.geeksforgeeks.org/k-maximum-sum-combinations-two-arrays/

Chapter 29

K-ary Heap

K-ary Heap - GeeksforGeeks

Prerequisite – Binary Heap

K-ary heaps are a generalization of binary heap(K=2) in which each node have K children instead of 2. Just like binary heap, it follows two properties:

1) Nearly complete binary tree, with all levels having maximum number of nodes except the last, which is filled in left to right manner.

2) Like Binary Heap, it can be divided into two categories: (a) Max k-ary heap (key at root is greater than all descendants and same is recursively true for all nodes). (b) Min k-ary heap (key at root is lesser than all descendants and same is recursively true for all nodes)

Examples:

```
3-ary max heap - root node is maximum
                 of all nodes
         10
     /    |   \
   7      9     8
 / | \   /
4  6  5 7
```

```
3-ary min heap -root node is minimum
                of all nodes
         10
      /   |  \
    12    11  13
  / | \
14 15 18
```

The height of a complete k-ary tree with n-nodes is given by $\log_k n$.

Applications of K-ary Heap:

- K-ary heap when used in the implementation of priority queue allows faster decrease key operation as compared to binary heap ($O(\log_2 n)$) for binary heap vs $O(\log_k n)$ for K-ary heap). Nevertheless, it causes the complexity of extractMin() operation to increase to O(k log $_k$n) as compared to the complexity of $O(\log_2 n)$ when using binary heaps for priority queue. This allows K-ary heap to be more efficient in algorithms where decrease priority operations are more common than extractMin() operation.Example: Dijkstra's algorithm for single source shortest path and Prim's algorithm for minimum spanning tree
- K-ary heap has better memory cache behaviour than a binary heap which allows them to run more quickly in practice, although it has a larger worst case running time of both extractMin() and delete() operation (both being O(k log $_k$n)).

Implementation
Assuming 0 based indexing of array, an array represents a K-ary heap such that for any node we consider:

- Parent of the node at index i (except root node) is located at index (i-1)/k
- Children of the node at index i are at indices (k*i)+1 , (k*i)+2 …. (k*i)+k
- The last non-leaf node of a heap of size n is located at index (n-2)/k

buildHeap() : Builds a heap from an input array.
This function runs a loop starting from the last non-leaf node all the way upto the root node, calling a function restoreDown(also known as maHeapify) for each index that restores the passed index at the correct position of the heap by shifting the node down in the K-ary heap building it in a bottom up manner.
Why do we start the loop from the last non-leaf node ?
Because all the nodes after that are leaf nodes which will trivially satisfy the heap property as they don't have any children and hence, are already roots of a K-ary max heap.

restoreDown() (or maxHeapify) : Used to maintain heap property.
It runs a loop where it finds the maximum of all the node's children, compares it with its own value and swaps if the max(value of all children) > (value at node). It repeats this step until the node is restored into its original position in the heap.

extractMax() : Extracting the root node.
A k-ary max heap stores the largest element in its root. It returns the root node, copies last node to the first, calls restore down on the first node thus maintaining the heap property.

insert() : Inserting a node into the heap
This can be achieved by inserting the node at the last position and calling restoreUp() on the given index to restore the node at its proper position in the heap. restoreUp() iteratively compares a given node with its parent, since in a max heap the parent is always greater than or equal to its children nodes, the node is swapped with its parent only when its key is greater than the parent.

Combining the above, following is the C++ implementation of K-ary heap.

```
 // C++ program to demonstrate all operations of
// k-ary Heap
#include<bits/stdc++.h>

using namespace std;

// function to heapify (or restore the max- heap
// property). This is used to build a k-ary heap
// and in extractMin()
// att[] -- Array that stores heap
// len   -- Size of array
// index -- index of element to be restored
//          (or heapified)
void restoreDown(int arr[], int len, int index,
                                        int k)
{
    // child array to store indexes of all
    // the children of given node
    int child[k+1];

    while (1)
    {
        // child[i]=-1 if the node is a leaf
        // children (no children)
        for (int i=1; i<=k; i++)
            child[i] = ((k*index + i) < len) ?
                    (k*index + i) : -1;

        // max_child stores the maximum child and
        // max_child_index holds its index
        int max_child = -1, max_child_index ;

        // loop to find the maximum of all
        // the children of a given node
        for (int i=1; i<=k; i++)
        {
            if (child[i] != -1 &&
                arr[child[i]] > max_child)
            {
                max_child_index = child[i];
                max_child = arr[child[i]];
            }
        }

        // leaf node
        if (max_child == -1)
            break;
```

```
        // swap only if the key of max_child_index
        // is greater than the key of node
        if (arr[index] < arr[max_child_index])
            swap(arr[index], arr[max_child_index]);

        index = max_child_index;
    }
}

// Restores a given node up in the heap. This is used
// in decreaseKey() and insert()
void restoreUp(int arr[], int index, int k)
{
    // parent stores the index of the parent variable
    // of the node
    int parent = (index-1)/k;

    // Loop should only run till root node in case the
    // element inserted is the maximum restore up will
    // send it to the root node
    while (parent>=0)
    {
        if (arr[index] > arr[parent])
        {
            swap(arr[index], arr[parent]);
            index = parent;
            parent = (index -1)/k;
        }

        // node has been restored at the correct position
        else
            break;
    }
}

// Function to build a heap of arr[0..n-1] and alue of k.
void buildHeap(int arr[], int n, int k)
{
    // Heapify all internal nodes starting from last
    // non-leaf node all the way upto the root node
    // and calling restore down on each
    for (int i= (n-1)/k; i>=0; i--)
        restoreDown(arr, n, i, k);
}

// Function to insert a value in a heap. Parameters are
// the array, size of heap, value k and the element to
// be inserted
```

```
void insert(int arr[], int* n, int k, int elem)
{
    // Put the new element in the last position
    arr[*n] = elem;

    // Increase heap size by 1
    *n = *n+1;

    // Call restoreUp on the last index
    restoreUp(arr, *n-1, k);
}

// Function that returns the key of root node of
// the heap and then restores the heap property
// of the remaining nodes
int extractMax(int arr[], int* n, int k)
{
    // Stores the key of root node to be returned
    int max = arr[0];

    // Copy the last node's key to the root node
    arr[0] = arr[*n-1];

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

    // Call restoreDown on the root node to restore
    // it to the correct position in the heap
    restoreDown(arr, *n, 0, k);

    return max;
}

// Driver program
int main()
{
    const int capacity = 100;
    int arr[capacity] = {4, 5, 6, 7, 8, 9, 10};
    int n = 7;
    int k = 3;

    buildHeap(arr, n, k);

    printf("Built Heap : \n");
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    int element = 3;
```

```
    insert(arr, &n, k, element);

    printf("\n\nHeap after insertion of %d: \n",
            element);
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    printf("\n\nExtracted max is %d",
                   extractMax(arr, &n, k));

    printf("\n\nHeap after extract max: \n");
    for (int i=0; i<n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output

```
Built Heap :
10 9 6 7 8 4 5

Heap after insertion of 3:
10 9 6 7 8 4 5 3

Extracted max is 10

Heap after extract max:
9 8 6 7 3 4 5
```

Time Complexity Analysis

- For a k-ary heap, with n nodes the maximum height of the given heap will be $\log_k n$. So restoreUp() run for maximum of $\log_k n$ times (as at every iteration the node is shifted one level up is case of restoreUp() or one level down in case of restoreDown).
- restoreDown() calls itself recursively for k children. So time complexity of this functions is O(k $\log_k n$).
- Insert and decreaseKey() operations call restoreUp() once. So complexity is O($\log_k n$).
- Since extractMax() calls restoreDown() once, its complexity O(k $\log_k n$)
- Time complexity of build heap is O(n) (Analysis is similar to binary heap)

Improved By : Amey Pawar, MohitRohatgi

Source

https://www.geeksforgeeks.org/k-ary-heap/

Chapter 30

K-th Largest Sum Contiguous Subarray

K-th Largest Sum Contiguous Subarray - GeeksforGeeks

Given an array of integers. Write a program to find the K-th largest sum of contiguous subarray within the array of numbers which has negative and positive numbers.

Examples:

```
Input: a[] = {20, -5, -1}
         k = 3
Output: -1
Explanation: All sum of contiguous
subarrays are (20, 15, 14, -5, -6, -1)
so the 4th largest sum is -1.

Input: a[] = {10, -10, 20, -40}
         k = 6
Output: -10
Explanation: The 6th largest sum among
sum of all contiguous subarrays is -10.
```

A **brute force approach** approach is to store all the contiguous sums in another array and sort it, and print the k-th largest. But in case of number of elements being large, the array in which we store the contiguous sums will run out of memory as the number of contiguous subarrays will be large (quadratic order)

An **efficient approach** is store the pre-sum of the array in a sum[] array. We can find sum of contiguous subarray from index i to j as sum[j]-sum[i-1]

Now for storing the Kth largest sum, use a min heap (priority queue) in which we push the contiguous sums till we get K elements, once we have our K elements, check if the element

if greater then the Kth element it is inserted to the min heap with popping out the top element in the min-heap, else not inserted . At the end the top element in the min-heap will be your answer.

Below is the implementation of above approach.

C++

```
 // CPP program to find the k-th largest sum
// of subarray
#include <bits/stdc++.h>
using namespace std;

// function to calculate kth largest elemnt
// in contiguous subarray sum
int kthLargestSum(int arr[], int n, int k)
{
    // array to store predix sums
    int sum[n + 1];
    sum[0] = 0;
    sum[1] = arr[0];
    for (int i = 2; i <= n; i++)
        sum[i] = sum[i - 1] + arr[i - 1];

    // priority_queue of min heap
    priority_queue<int, vector<int>, greater<int> > Q;

    // loop to calculate the contigous subarray
    // sum position-wise
    for (int i = 1; i <= n; i++)
    {

        // loop to traverse all positions that
        // form contiguous subarray
        for (int j = i; j <= n; j++)
        {
            // calculates the contiguous subarray
            // sum from j to i index
            int x = sum[j] - sum[i - 1];

            // if queue has less then k elements,
            // then simply push it
            if (Q.size() < k)
                Q.push(x);

            else
            {
                // it the min heap has equal to
```

```
                // k elements then just check
                // if the largest kth element is
                // smaller than x then insert
                // else its of no use
                if (Q.top() < x)
                {
                    Q.pop();
                    Q.push(x);
                }
            }
        }
    }

    // the top element will be then kth
    // largest element
    return Q.top();
}

// Driver program to test above function
int main()
{
    int a[] = { 10, -10, 20, -40 };
    int n = sizeof(a) / sizeof(a[0]);
    int k = 6;

    // calls the function to find out the
    // k-th largest sum
    cout << kthLargestSum(a, n, k);
    return 0;
}
```

Java

```
 // Java program to find the k-th
// argest sum of subarray
import java.util.*;

class KthLargestSumSubArray
{
    // function to calculate kth largest
    // element in contiguous subarray sum
    static int kthLargestSum(int arr[], int n, int k)
    {
        // array to store predix sums
        int sum[] = new int[n + 1];
        sum[0] = 0;
        sum[1] = arr[0];
        for (int i = 2; i <= n; i++)
```

```
        sum[i] = sum[i - 1] + arr[i - 1];

    // priority_queue of min heap
    PriorityQueue<Integer> Q = new PriorityQueue<Integer> ();

    // loop to calculate the contigous subarray
    // sum position-wise
    for (int i = 1; i <= n; i++)
    {

        // loop to traverse all positions that
        // form contiguous subarray
        for (int j = i; j <= n; j++)
        {
            // calculates the contiguous subarray
            // sum from j to i index
            int x = sum[j] - sum[i - 1];

            // if queue has less then k elements,
            // then simply push it
            if (Q.size() < k)
                Q.add(x);

            else
            {
                // it the min heap has equal to
                // k elements then just check
                // if the largest kth element is
                // smaller than x then insert
                // else its of no use
                if (Q.peek() < x)
                {
                    Q.poll();
                    Q.add(x);
                }
            }
        }
    }

    // the top element will be then kth
    // largest element
    return Q.poll();
}

// Driver Code
public static void main(String[] args)
{
    int a[] = new int[]{ 10, -10, 20, -40 };
```

```
        int n = a.length;
        int k = 6;

        // calls the function to find out the
        // k-th largest sum
        System.out.println(kthLargestSum(a, n, k));
    }
}

/* This code is contributed by Danish Kaleem */
```

Output:

```
-10
```

Time complexity: O(n^2 log (k))
Auxiliary Space : O(k) for min-heap and we can store the sum array in the array itself as it is of no use.

Source

https://www.geeksforgeeks.org/k-th-largest-sum-contiguous-subarray/

Chapter 31

Kth smallest element after every insertion

Kth smallest element after every insertion - GeeksforGeeks

Given an infinite stream of integers, find the k'th largest element at any point of time. It may be assumed that 1 <= k <= n.

```
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).

The idea is to use min heap.
1) Store first k elements in min heap.
2) For every element from (k+1)-th to n-th, do following.
.....a) Print root of heap.
.....b) If current element is more than root of heap, pop root and insert

```
 // CPP program to find k-th largest element in a
// stream after every insertion.
#include <bits/stdc++.h>
using namespace std;

int kthLargest(int stream[], int n, int k)
{
   // Create a min heap and store first k-1 elements
   // of stream into
   priority_queue<int, vector<int>, greater<int> > pq;
```

```
   // Push first k elements and print "_" (k-1) times
   for (int i=0; i<k-1; i++)
   {
      pq.push(stream[i]);
      cout << "_ ";
   }
   pq.push(stream[k-1]);

   for (int i=k; i<n; i++)
   {
       // We must insert last element before we
       // decide last k-th largest output.
       if (i < n-1)
          cout << pq.top() << " ";

       if (stream[i] > pq.top())
       {
           pq.pop();
           pq.push(stream[i]);
       }
   }

   // Print last k-th largest element (after
   // (inserting last element)
   cout << pq.top();
}

// Driver code
int main()
{
   int arr[] = {10, 20, 11, 70, 50, 40, 100, 55};
   int k = 3;
   int n = sizeof(arr)/sizeof(arr[0]);
   kthLargest(arr, n, k);
   return 0;
}
```

Output:

```
_ _ 10 11 20 40 55
```

If stream contains elements of non-primitive types, we may define our own compactor function and create a priority_queue accordingly.

Source

https://www.geeksforgeeks.org/kth-smallest-element-after-every-insertion/

Chapter 32

Kth smallest element in a row-wise and column-wise sorted 2D array | Set 1

Kth smallest element in a row-wise and column-wise sorted 2D array | Set 1 - GeeksforGeeks

Given an n x n matrix, where every row and column is sorted in non-decreasing order. Find the kth smallest element in the given 2D array.

For example, consider the following 2D array.

```
        10, 20, 30, 40
        15, 25, 35, 45
        24, 29, 37, 48
        32, 33, 39, 50
The 3rd smallest element is 20 and 7th smallest element is 30
```

The idea is to use min heap. Following are detailed step.
1) Build a min heap of elements from first row. A heap entry also stores row number and column number.
2) Do following k times.
…a) Get minimum element (or root) from min heap.
…b) Find row number and column number of the minimum element.
…c) Replace root with the next element from same column and min-heapify the root.
3) Return the last extracted root.

Following is C++ implementation of above algorithm.

```
 // kth largest element in a 2d array sorted row-wise and column-wise
#include<iostream>
```

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

// A structure to store an entry of heap.  The entry contains
// a value from 2D array, row and column numbers of the value
struct HeapNode {
    int val;  // value to be stored
    int r;    // Row number of value in 2D array
    int c;    // Column number of value in 2D array
};

// A utility function to swap two HeapNode items.
void swap(HeapNode *x, HeapNode *y) {
    HeapNode z = *x;
    *x = *y;
    *y = z;
}

// A utility function to minheapify the node harr[i] of a heap
// stored in harr[]
void minHeapify(HeapNode harr[], int i, int heap_size)
{
    int l = i*2 + 1;
    int r = i*2 + 2;
    int smallest = i;
    if (l < heap_size && harr[l].val < harr[i].val)
        smallest = l;
    if (r < heap_size && harr[r].val < harr[smallest].val)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        minHeapify(harr, smallest, heap_size);
    }
}

// A utility function to convert harr[] to a max heap
void buildHeap(HeapNode harr[], int n)
{
    int i = (n - 1)/2;
    while (i >= 0)
    {
        minHeapify(harr, i, n);
        i--;
    }
}

// This function returns kth smallest element in a 2D array mat[][]
```

```
int kthSmallest(int mat[4][4], int n, int k)
{
    // k must be greater than 0 and smaller than n*n
    if (k <= 0 || k > n*n)
       return INT_MAX;

    // Create a min heap of elements from first row of 2D array
    HeapNode harr[n];
    for (int i = 0; i < n; i++)
        harr[i] =  {mat[0][i], 0, i};
    buildHeap(harr, n);

    HeapNode hr;
    for (int i = 0; i < k; i++)
    {
       // Get current heap root
       hr = harr[0];

       // Get next value from column of root's value. If the
       // value stored at root was last value in its column,
       // then assign INFINITE as next value
       int nextval = (hr.r < (n-1))? mat[hr.r + 1][hr.c]: INT_MAX;

       // Update heap root with next value
       harr[0] =  {nextval, (hr.r) + 1, hr.c};

       // Heapify root
       minHeapify(harr, 0, n);
    }

    // Return the value at last extracted root
    return hr.val;
}

// driver program to test above function
int main()
{
  int mat[4][4] = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {25, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
  cout << "7th smallest element is " << kthSmallest(mat, 4, 7);
  return 0;
}
```

Output:

```
7th smallest element is 30
```

Time Complexity: The above solution involves following steps.
1) Build a min heap which takes O(n) time
2) Heapify k times which takes O(kLogn) time.
Therefore, overall time complexity is O(n + kLogn) time.

The above code can be optimized to build a heap of size k when k is smaller than n. In that case, the kth smallest element must be in first k rows and k columns.

We will soon be publishing more efficient algorithms for finding the kth smallest element.

This article is compiled by Ravi Gupta. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/kth-smallest-element-in-a-row-wise-and-column-wise-sorted-2d-array-set-1/

Chapter 33

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

K'th Smallest/Largest Element in Unsorted Array | Set 1 - GeeksforGeeks

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(nlogn) 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(nLogn).

C++

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

Java

```
 // Java code for kth smallest element
// in an array
import java.util.Arrays;
import java.util.Collections;

class GFG
{
    // Function to return k'th smallest
    // element in a given array
    public static int kthSmallest(Integer [] arr,
                                          int k)
    {
        // Sort the given array
        Arrays.sort(arr);

        // Return k'th element in
        // the sorted array
        return arr[k-1];
    }

    // driver program
    public static void main(String[] args)
    {
        Integer arr[] = new Integer[]{12, 3, 5, 7, 19};
        int k = 2;
        System.out.print( "K'th smallest element is "+
                           kthSmallest(arr, k) );
```

```
    }
}

// This code is contributed by Chhavi
```

C#

```
 // C# code for kth smallest element
// in an array
using System;

class GFG {

    // Function to return k'th smallest
    // element in a given array
    public static int kthSmallest(int []arr,
                                  int k)
    {

        // Sort the given array
        Array.Sort(arr);

        // Return k'th element in
        // the sorted array
        return arr[k-1];
    }

    // driver program
    public static void Main()
    {
        int []arr = new int[]{12, 3, 5,
                              7, 19};
        int k = 2;
        Console.Write( "K'th smallest element"
              + " is "+ kthSmallest(arr, k) );
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
 <?php
// Simple PHP program to find
// k'th smallest element

// Function to return k'th smallest
```

```
// element in a given array
function kthSmallest($arr, $n, $k)
{

    // Sort the given array
    sort($arr);

    // Return k'th element
    // in the sorted array
    return $arr[$k - 1];
}

    // Driver Code
    $arr = array(12, 3, 5, 7, 19);
    $n =count($arr);
    $k = 2;
    echo "K'th smallest element is "
       , kthSmallest($arr, $n, $k);

// This code is contributed by anuj_67.
?>
```

```
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(nLogn). A simple optomization is to create a Min Heapof 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<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 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

```
 // A C++ program to find k'th smallest element using max heap
#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 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 5
```

Method 4 (QuickSelect)
This is an optimization over method 1 if QuickSortis 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 $O(n^2)$, but it works in O(n) on average.

C++

```
 #include<iostream>
#include<climits>
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;
}
```

Java

```
 // Java code for kth smallest element in an array
import java.util.Arrays;
import java.util.Collections;

class GFG
{
    // 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
    public static int partition(Integer [] arr, int l,
                                            int r)
    {
        int x = arr[r], i = l;
        for (int j = l; j <= r - 1; j++)
        {
            if (arr[j] <= x)
            {
                //Swapping arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;

                i++;
            }
        }

        //Swapping arr[i] and arr[r]
        int temp = arr[i];
        arr[i] = arr[r];
        arr[r] = temp;
```

```
        return i;
    }

    // This function returns k'th smallest element
    // in arr[l..r] using QuickSort based method.
    // ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
    public static int kthSmallest(Integer[] 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 position is more, recur for
            // left subarray
            if (pos-l > k-1)
                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 Integer.MAX_VALUE;
    }

    // Driver program to test above methods
    public static void main(String[] args)
    {
        Integer arr[] = new Integer[]{12, 3, 5, 7, 4, 19, 26};
        int k = 3;
        System.out.print( "K'th smallest element is " +
                     kthSmallest(arr, 0, arr.length - 1, k) );
    }
}

// This code is contributed by Chhavi
```

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 (see the following posts).

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)

References:
http://www.ics.uci.edu/~eppstein/161/960125.html
http://www.cs.rit.edu/~ib/Classes/CS515_Spring12-13/Slides/022-SelectMasterThm.pdf

Improved By : nitin mittal, vt_m

Source

https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/

Chapter 34

K'th largest element in a stream

K'th largest element in a stream - GeeksforGeeks

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
```

This article is contributed by **Shivam Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/kth-largest-element-in-a-stream/

Chapter 35

LFU (Least Frequently Used) Cache Implementation

LFU (Least Frequently Used) Cache Implementation - GeeksforGeeks

Least Frequently Used (LFU) is a caching algorithm in which the least frequently used cache block is removed whenever the cache is overflowed. In LFU we check the old page as well as the frequency of that page and if the frequency of the page is larger than the old page we cannot remove it and if all the old pages are having same frequency then take last i.e FIFO method for that and remove that page.

Min-heap data structure is a good option to implement this algorithm, as it handles insertion, deletion, and update in logarithmic time complexity. A tie can be resolved by removing the least recently used cache block. The following two containers have been used to solve the problem:

- A vector of integer pairs has been used to represent the cache, where each pair consists of the block number and the number of times it has been used. The vector is ordered in the form of a min-heap, which allows us to access the least frequently used block in constant time.
- A hashmap has been used to store the indices of the cache blocks which allows searching in constant time.

Below is the implementation of the above approach:

```
 // C++ program for LFU cache implementation
#include <bits/stdc++.h>
using namespace std;

// Generic function to swap two pairs
void swap(pair<int, int>& a, pair<int, int>& b)
{
```

```
    pair<int, int> temp = a;
    a = b;
    b = temp;
}

// Returns the index of the parent node
inline int parent(int i)
{
    return (i - 1) / 2;
}

// Returns the index of the left child node
inline int left(int i)
{
    return 2 * i + 1;
}

// Returns the index of the right child node
inline int right(int i)
{
    return 2 * i + 2;
}

// Self made heap tp Rearranges
//  the nodes in order to maintain the heap property
void heapify(vector<pair<int, int> >& v,
             unordered_map<int, int>& m, int i, int n)
{
    int l = left(i), r = right(i), minim;
    if (l < n)
        minim = ((v[i].second < v[l].second) ? i : l);
    else
        minim = i;
    if (r < n)
        minim = ((v[minim].second < v[r].second) ? minim : r);
    if (minim != i) {
        m[v[minim].first] = i;
        m[v[i].first] = minim;
        swap(v[minim], v[i]);
        heapify(v, m, minim, n);
    }
}

// Function to Increment the frequency
// of a node and rearranges the heap
void increment(vector<pair<int, int> >& v,
               unordered_map<int, int>& m, int i, int n)
{
```

```
    ++v[i].second;
    heapify(v, m, i, n);
}

// Function to Insert a new node in the heap
void insert(vector<pair<int, int> >& v,
            unordered_map<int, int>& m, int value, int& n)
{

    if (n == v.size()) {
        m.erase(v[0].first);
        cout << "Cache block " << v[0].first
                              << " removed.\n";
        v[0] = v[--n];
        heapify(v, m, 0, n);
    }
    v[n++] = make_pair(value, 1);
    m.insert(make_pair(value, n - 1));
    int i = n - 1;

    // Insert a node in the heap by swapping elements
    while (i && v[parent(i)].second > v[i].second) {
        m[v[i].first] = parent(i);
        m[v[parent(i)].first] = i;
        swap(v[i], v[parent(i)]);
        i = parent(i);
    }
    cout << "Cache block " << value << " inserted.\n";
}

// Function to refer to the block value in the cache
void refer(vector<pair<int, int> >& cache, unordered_map<int,
                    int>& indices, int value, int& cache_size)
{
    if (indices.find(value) == indices.end())
        insert(cache, indices, value, cache_size);
    else
        increment(cache, indices, indices[value], cache_size);
}

// Driver Code
int main()
{
    int cache_max_size = 4, cache_size = 0;
    vector<pair<int, int> > cache(cache_max_size);
    unordered_map<int, int> indices;
    refer(cache, indices, 1, cache_size);
    refer(cache, indices, 2, cache_size);
```

```
    refer(cache, indices, 1, cache_size);
    refer(cache, indices, 3, cache_size);
    refer(cache, indices, 2, cache_size);
    refer(cache, indices, 4, cache_size);
    refer(cache, indices, 5, cache_size);
    return 0;
}
```

Output:

```
Cache block 1 inserted.
Cache block 2 inserted.
Cache block 3 inserted.
Cache block 4 inserted.
Cache block 3 removed.
Cache block 5 inserted.
```

Source

https://www.geeksforgeeks.org/lfu-least-frequently-used-cache-implementation/

Chapter 36

Largest Derangement of a Sequence

Largest Derangement of a Sequence - GeeksforGeeks

Given any sequence $S = [1, 2, 3, \ldots, N-2, N-1, N]$, find the **largest derangement** of S.

A derangement D is any permutation of S, such that no two elements at the same position in S and D are equal.

The Largest Derangement is such that $D_1 > D_2 > D_3 \ldots > D_{N-1} > D_N$.

Examples:

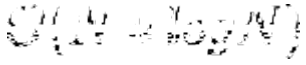
```
Input : seq[] = {5, 4, 3, 2, 1}
Output : 4 5 2 1 3

Input : seq[] = {56, 21, 42, 67, 23, 74}
Output : 74, 67, 56, 42, 23, 21
```

Since we are interested in generating largest derangement, we start putting larger elements in more significant positions.

Start from left, at any position i place the next largest element among the values of the sequence which have not yet been placed in positions before i.

To scan all positions takes N iteration. In each iteration we are required to find a maximum numbers, so a trivial implementation would be $O(N^2)$ complexity,

However if we use a data structure like max-heap to find the maximum element, then the complexity reduces to 

Below is C++ implementation.

```
 // CPP program to find the largest derangement
#include <bits/stdc++.h>
using namespace std;

void printLargest(int seq[], int N)
{
    int res[N]; // Stores result

    // Insert all elements into a priority queue
    std::priority_queue<int> pq;
    for (int i = 0; i < N; i++)
        pq.push(seq[i]);

    // Fill Up res[] from left to right
    for (int i = 0; i < N; i++) {
        int d = pq.top();
        pq.pop();
        if (d != seq[i] || i == N - 1) {
            res[i] = d;
        } else {

            // New Element poped equals the element
            // in original sequence. Get the next
            // largest element
            res[i] = pq.top();
            pq.pop();
            pq.push(d);
        }
    }

    // If given sequence is in descending order then
    // we need to swap last two elements again
    if (res[N - 1] == seq[N - 1]) {
        res[N - 1] = res[N - 2];
        res[N - 2] = seq[N - 1];
    }

    printf("\nLargest Derangement \n");
    for (int i = 0; i < N; i++)
        printf("%d ", res[i]);
}

// Driver code
```

```
int main()
{
    int seq[] = { 92, 3, 52, 13, 2, 31, 1 };
    int n = sizeof(seq)/sizeof(seq[0]);
    printLargest(seq, n);
    return 0;
}
```

Output:

```
Sequence:
92 3 52 13 2 31 1
Largest Derangement
52 92 31 3 13 1 2
```

Note:
The method can be easily modified to obtain the smallest derangement as well.
Instead of a **Max Heap**, we should use a **Min Heap** to consecutively get minimum elements

Source

https://www.geeksforgeeks.org/largest-derangement-sequence/

Chapter 37

Largest triplet product in a stream

Largest triplet product in a stream - GeeksforGeeks

Given a stream of integers represented as arr[]. For each index i from 0 to n-1, print the multiplication of largest, second largest, third largest element of the subarray arr[0…i]. If i $<$ 2 print -1.

Examples:

```
Input : arr[] = {1, 2, 3, 4, 5}
Output :-1
        -1
         6
         24
         60
Explanation : for i = 2 only three elements
are there {1, 2, 3} so answer is 6. For i = 3
largest three elements are {2, 3, 4} their
product is 2*3*4 = 24 ....so on
```

We will use priority queue here.

1. Insert arr[i] in the priority queue
2. As the top element in priority queue is largest so pop it and store it as x. Now the top element in the priority queue will be the second largest element in subarray arr[0…i] pop it and store as y. Now the top element is third largest element in subarray arr[0…i] so pop it and store it as z.
3. Print x*y*z
4. Reinsert x, y, z.

```
 // C++ implementation of largest triplet
// multiplication
#include<bits/stdc++.h>
using namespace std;

// Prints the product of three largest numbers
// in subarray arr[0..i]
void LargestTripletMultiplication(int arr[], int n)
{
    // call a priority queue
    priority_queue <int> q;

    // traversing the array
    for (int i = 0; i<n; i++)
    {
        // pushing arr[i] in array
        q.push(arr[i]);

        // if less than three elements are present
        // in array print -1
        if (q.size() < 3)
            cout << "-1" << endl;
        else
        {
            // pop three largest elements
            int x = q.top();
            q.pop();
            int y = q.top();
            q.pop();
            int z = q.top();
            q.pop();

            // Reinsert x, y, z in priority_queue
            int ans = x*y*z;
            cout << ans << endl;
            q.push(x);
            q.push(y);
            q.push(z);
        }
    }
    return ;
}

// Driver Function
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    LargestTripletMultiplication(arr, n);
    return 0;
}
```

Output:

```
-1
-1
6
24
60
```

Source

https://www.geeksforgeeks.org/largest-triplet-product-stream/

Chapter 38

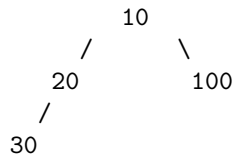
Leaf starting point in a Binary Heap data structure

Leaf starting point in a Binary Heap data structure - GeeksforGeeks

Binary Heap is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). In other words, we can say that it's an almost complete binary tree.

A Binary heap is typically represented as array. If we take a closer look, we can noticed that in a Heap with number of nodes n, the leaves start from a particular index and following it, all the nodes are leaves till index n.

Let's see an example to observe this:



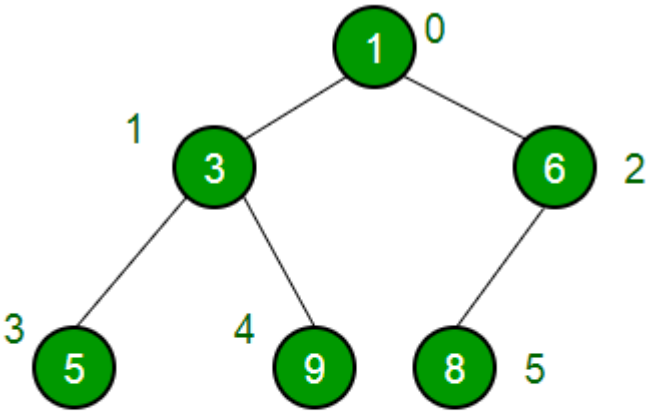
Let us represent this in the form of an array Arr whose index starts from 1 :
we have:
Arr[1] = 10
Arr[2] = 20
Arr[3] = 100
Arr[4] = 30
If we observe, the first leaf (i.e. 100) starts from the index 3. Following it Arr[4] is also a leaf.

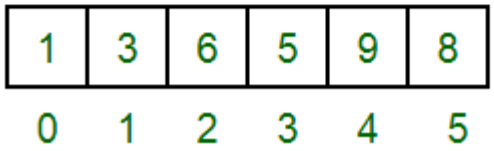
By carefully analyzing, the following conclusion is observed:

The first leaf in a Heap starts from [**floor(n/2)**]**+1** and all the nodes following it till
n are leaves.

Conclusion: In a Heap having n elements, Elements from indexes [**(floor(n/2)+1) to n**] are leaves.

What is starting index of leaves if indexes start from 0 instead of 1?
The above explanation assumes indexes starting from 1, but in most of the programming languages, index starts with 0.





If we consider 0 as starting index, then leaves starts from **floor(n/2)** and exist till end, i.e., (n-1).

Source

https://www.geeksforgeeks.org/leaf-starting-point-binary-heap-data-structure/

Chapter 39

Leftist Tree / Leftist Heap

Leftist Tree / Leftist Heap - GeeksforGeeks

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a complete binary tree), a leftist tree may be very unbalanced.

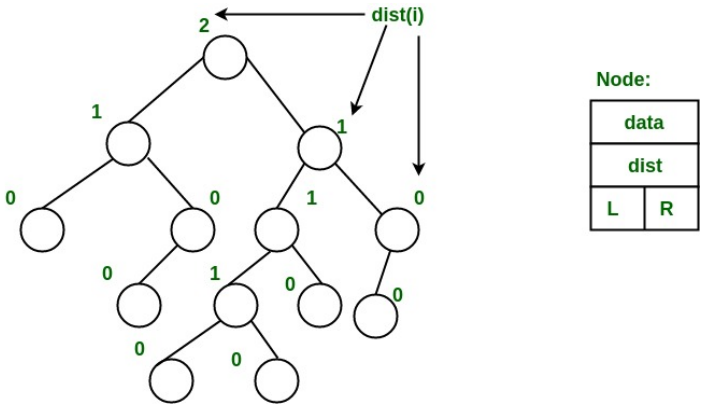
Below are time complexities of **Leftist Tree / Heap**.

```
  Function       Complexity              Comparison
1) Get Min:       O(1)       [same as both Binary and Binomial]
2) Delete Min:    O(Log n)   [same as both Binary and Binomial]
3) Insert:        O(Log n)   [O(Log n) in Binary and O(1) in
                             Binomial and O(Log n) for worst case]
4) Merge:         O(Log n)   [O(Log n) in Binomial]
```

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :** key(i) >= key(parent(i))
2. **Heavier on left side :** dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist(i) = 1 + dist(right(i)).

Example: The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by dist(i) = 1 + dist(right(i)). The same is followed for each node and their s-value(or rank) is calculated.



From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast $2^x - 1$ nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin() can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

Idea behind Merging :
Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
   – Update dist() of merged root.
   – Swap left and right subtrees just below root, if needed, to keep leftist property of merged
   result

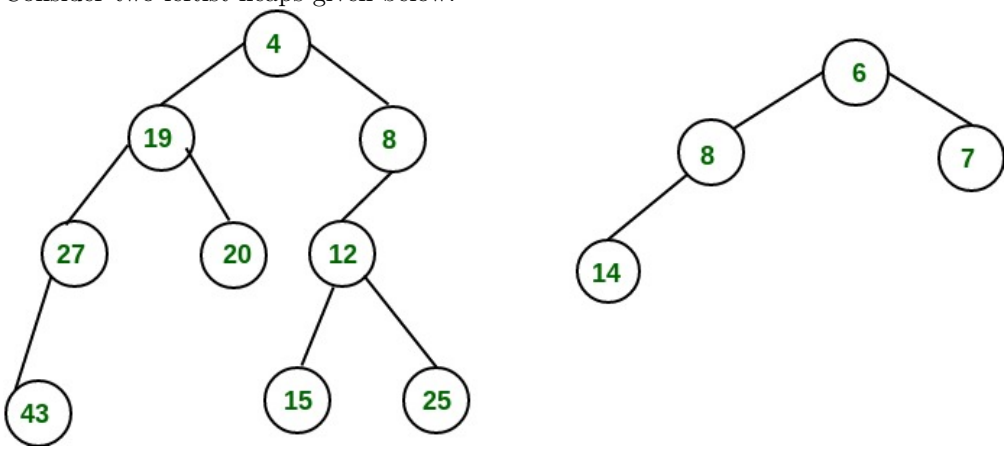
Source : http://courses.cs.washington.edu/courses/cse326/08sp/lectures/05-leftist-heaps.pdf

Detailed Steps for Merge:

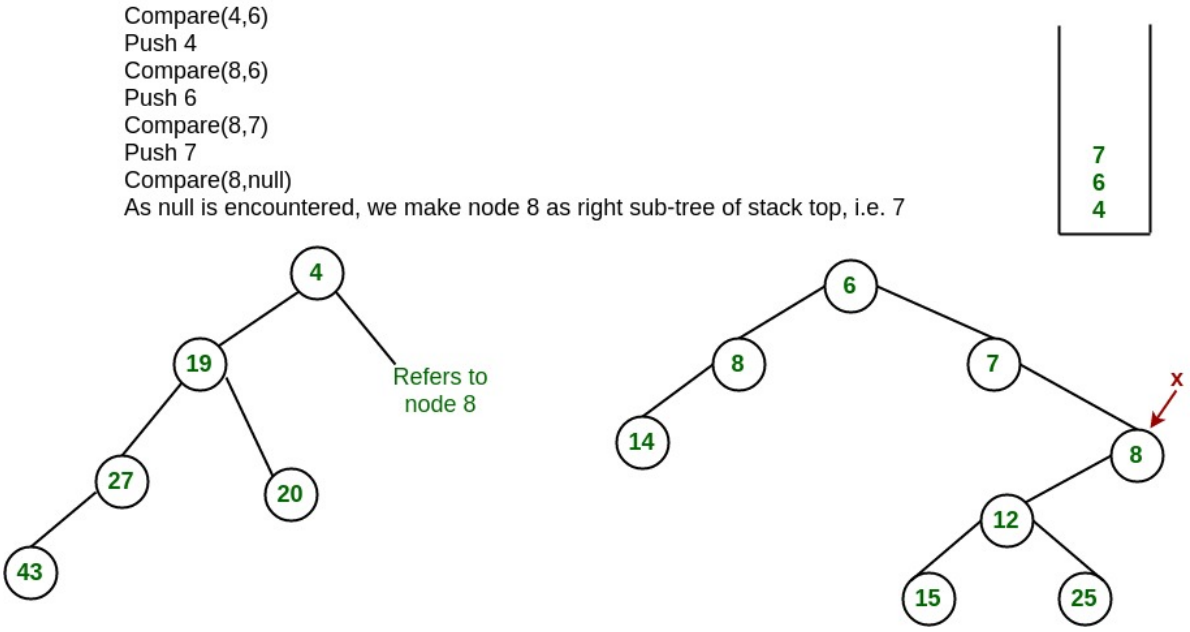
1. Compare the roots of two heaps.

2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Example:
Consider two leftist heaps given below:

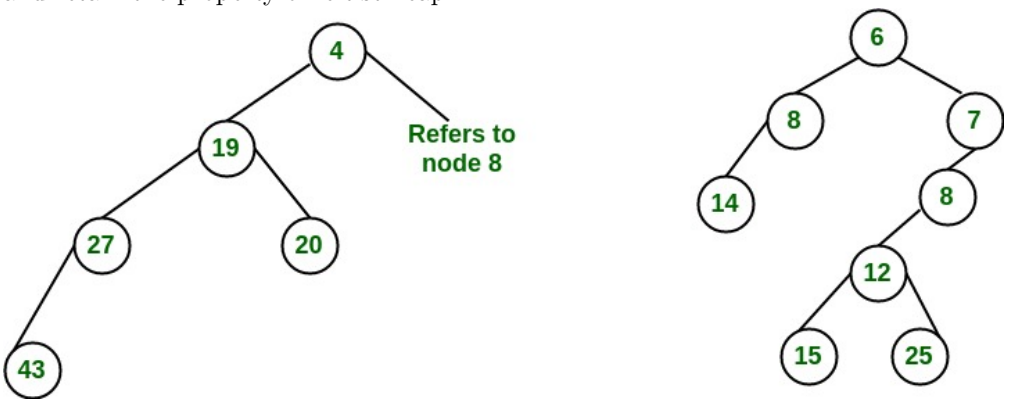


Merge them into a single leftist heap

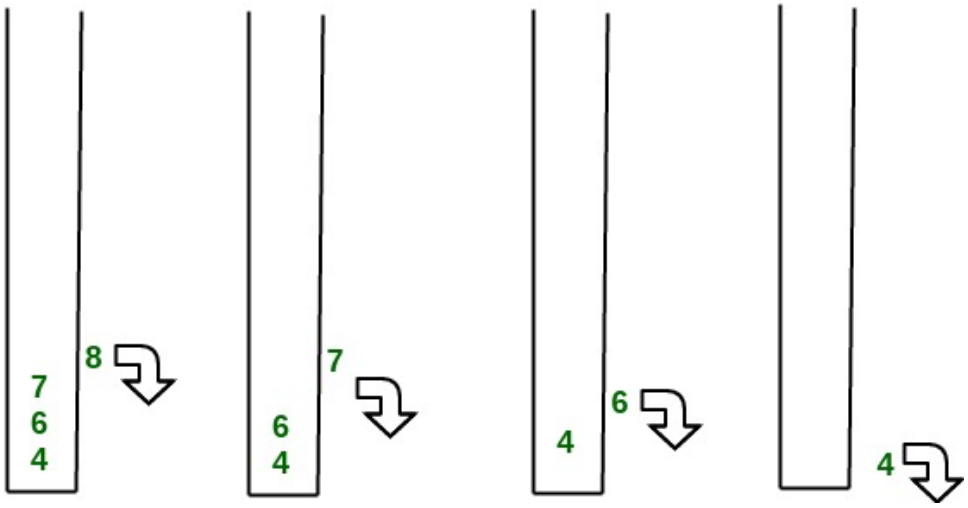


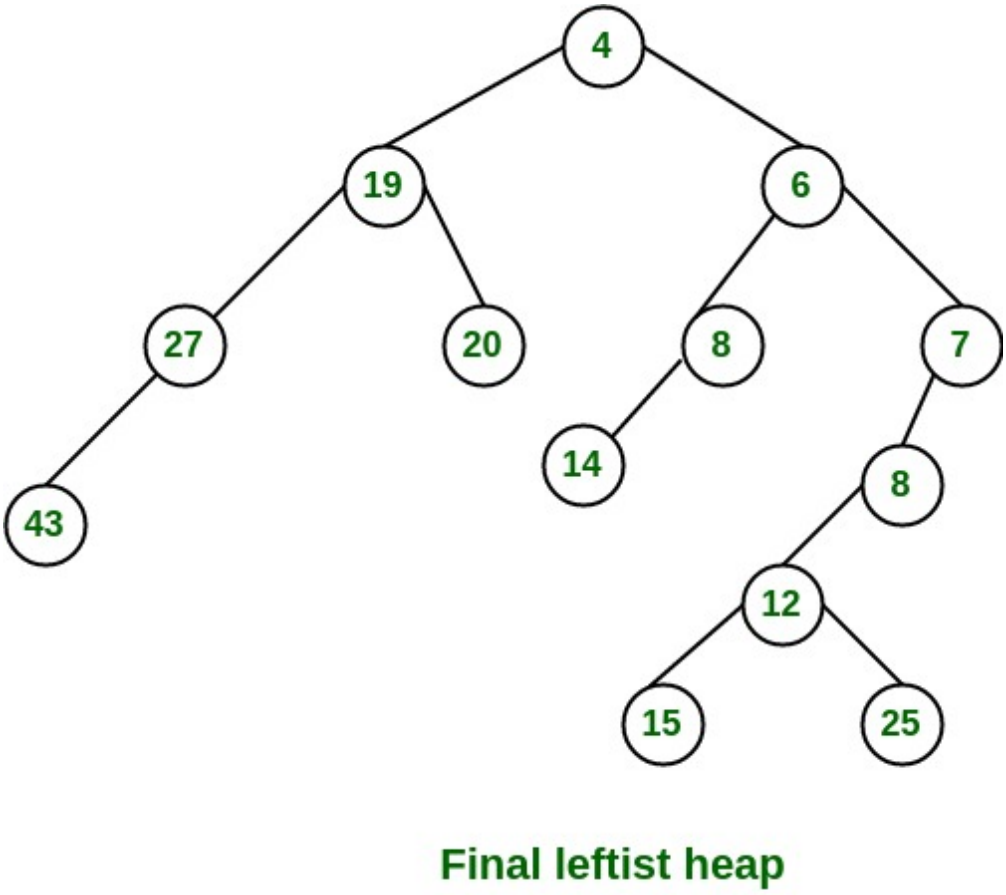
The subtree at node 7 violates the property of leftist heap so we swap it with the left child

and retain the property of leftist heap.



Convert to leftist heap. Repeat the process

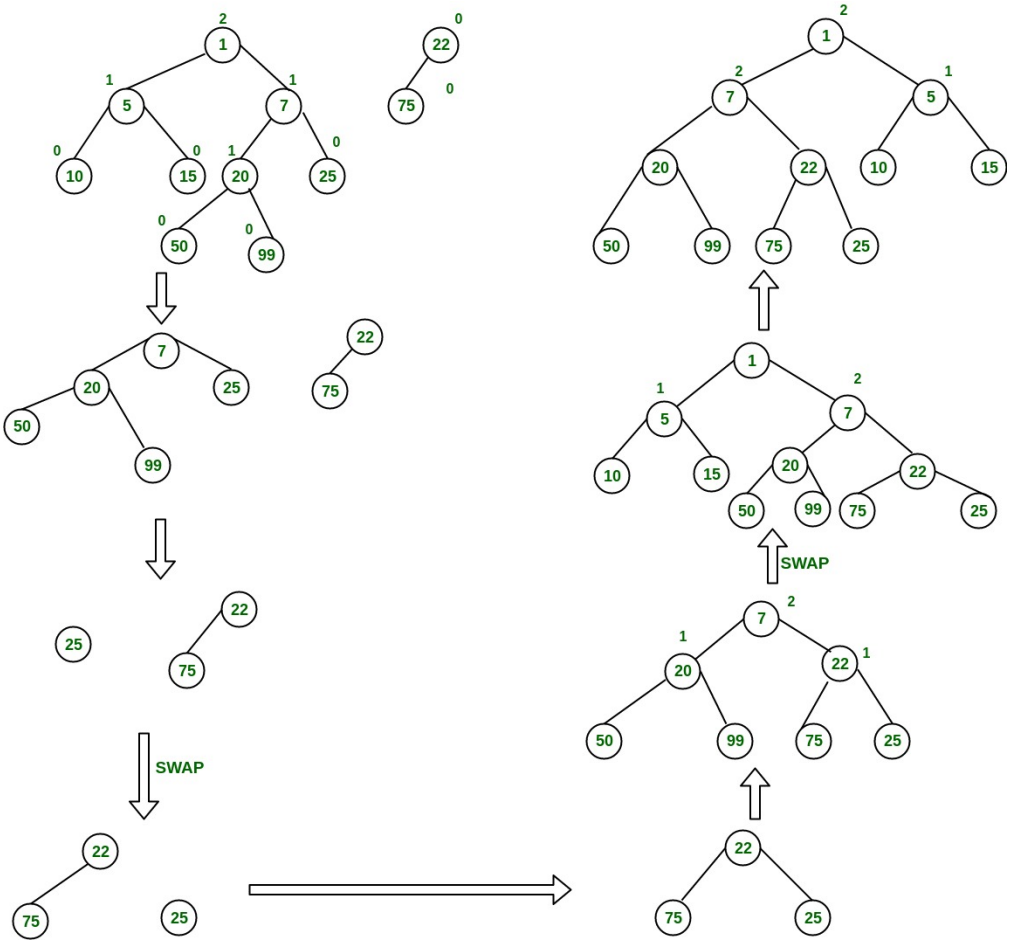




Final leftist heap

The worst case time complexity of this algorithm is O(log n) in the worst case, where n is the number of nodes in the leftist heap.

Another example of merging two leftist heap:



Implementation of leftist Tree / leftist Heap:

```
 //C++ program for leftist heap / leftist tree
#include <iostream>
#include <cstdlib>
using namespace std;

// Node Class Declaration
class LeftistNode
{
public:
    int element;
    LeftistNode *left;
    LeftistNode *right;
    int dist;
    LeftistNode(int & element, LeftistNode *lt = NULL,
                LeftistNode *rt = NULL, int np = 0)
    {
```

```
        this->element = element;
        right = rt;
        left = lt,
        dist = np;
    }
};

//Class Declaration
class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                       LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                        LeftistNode *h2);
    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
    root = NULL;
}

// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}

// Destruct the leftist heap
```

```
LeftistHeap::~LeftistHeap()
{
    makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
 Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                                LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
 Assumes trees are not empty, and h1's root contains
  smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                 LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
    }
    return h1;
}
```

```
// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
    LeftistNode *tmp = t->left;
    t->left = t->right;
    t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
  heap order.*/
void LeftistHeap::Insert(int &x)
{
    root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
    return root->element;
}

/* Remove the smallest item from the priority queue.
Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
    LeftistNode *oldRoot = root;
    root = Merge(root->left, root->right);
    delete oldRoot;
}

/* Remove the smallest item from the priority queue.
Pass back the smallest item, or throw Underflow if empty.*/
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}

/* Test if the priority queue is logically empty.
 Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()
```

```
{
    return root == NULL;
}

/* Test if the priority queue is logically full.
 Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
    return false;
}

// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
        delete t;
    }
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
    if (t == NULL)
        return NULL;
    else
        return new LeftistNode(t->element, clone(t->left),
                               clone(t->right), t->dist);
```

```
}

//Driver program
int main()
{
    LeftistHeap h;
    LeftistHeap h1;
    LeftistHeap h2;
    int x;
    int arr[]= {1, 5, 7, 10, 15};
    int arr1[]= {22, 75};

    h.Insert(arr[0]);
    h.Insert(arr[1]);
    h.Insert(arr[2]);
    h.Insert(arr[3]);
    h.Insert(arr[4]);
    h1.Insert(arr1[0]);
    h1.Insert(arr1[1]);

    h.deleteMin(x);
    cout<< x <<endl;

    h1.deleteMin(x);
    cout<< x <<endl;

    h.Merge(h1);
    h2 = h;

    h2.deleteMin(x);
    cout<< x << endl;

    return 0;
}
```

Output:

```
1
22
5
```

References:
Wikipedia- Leftist Tree
CSC378: Leftist Trees

Source

https://www.geeksforgeeks.org/leftist-tree-leftist-heap/

Chapter 40

Maximum difference between two subsets of m elements

Maximum difference between two subsets of m elements - GeeksforGeeks

Given an array of n integers and a number m, find the maximum possible difference between two sets of m elements chosen from given array.

Examples:

```
Input : arr[] = 1 2 3 4 5
            m = 4
Output : 4
The maximum four elements are 2, 3,
4 and 5. The minimum four elements are
1, 2, 3 and 4. The difference between
two sums is (2 + 3 + 4 + 5) - (1 + 2
+ 3 + 4) = 4

Input : arr[] = 5 8 11 40 15
           m = 2
Output : 42
The difference is (40 + 15) - (5  + 8)
```

The idea is to first sort the array, then find sum of first m elements and sum of last m elements. Finally return difference between two sums.

CPP

```
 // C++ program to find difference
// between max and min sum of array
#include <algorithm>
```

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

// utility function
int find_difference(int arr[], int n, int m)
{
    int max = 0, min = 0;

    // sort array
    sort(arr, arr + n);

    for (int i = 0, j = n - 1;
         i < m; i++, j--) {
        min += arr[i];
        max += arr[j];
    }

    return (max - min);
}

// Driver code
int main()
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int m = 4;
    cout << find_difference(arr, n, m);
    return 0;
}
```

Java

```
 // Java program to find difference
// between max and min sum of array
import java.util.Arrays;

class GFG {
    // utility function
    static int find_difference(int arr[], int n,
                               int m)
    {
        int max = 0, min = 0;

        // sort array
        Arrays.sort(arr);

        for (int i = 0, j = n - 1;
             i < m; i++, j--) {
```

```
            min += arr[i];
            max += arr[j];
        }

        return (max - min);
    }

    // Driver program
    public static void main(String arg[])
    {
        int arr[] = { 1, 2, 3, 4, 5 };
        int n = arr.length;
        int m = 4;
        System.out.print(find_difference(arr, n, m));
    }
}

// This code is contributed by Anant Agarwal.
```

Python3

```
 # Python program to
# find difference
# between max and
# min sum of array

def find_difference(arr, n, m):
    max = 0; min = 0

    # sort array
    arr.sort();
    j = n-1
    for i in range(m):
        min += arr[i]
        max += arr[j]
        j = j - 1

    return (max - min)

# Driver code
if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5]
    n = len(arr)
    m = 4

    print(find_difference(arr, n, m))

# This code is contributed by
```

```
# Harshit Saini
```

C#

```
 // C# program to find difference
// between max and min sum of array
using System;

class GFG {

    // utility function
    static int find_difference(int[] arr, int n,
                                           int m)
    {
        int max = 0, min = 0;

        // sort array
        Array.Sort(arr);

        for (int i = 0, j = n - 1;
            i < m; i++, j--) {
            min += arr[i];
            max += arr[j];
        }

        return (max - min);
    }

    // Driver program
    public static void Main()
    {
        int[] arr = { 1, 2, 3, 4, 5 };
        int n = arr.Length;
        int m = 4;
        Console.Write(find_difference(arr, n, m));
    }
}

// This code is contributed by nitin mittal
```

PHP

```
 <?php
// PHP program to find difference
// between max and min sum of array

// utility function
```

```
function find_difference($arr, $n, $m)
{
    $max = 0; $min = 0;

    // sort array
    sort($arr);
    sort( $arr,$n);

    for($i = 0, $j = $n - 1; $i <$m; $i++, $j--)
    {
        $min += $arr[$i];
        $max += $arr[$j];
    }

    return ($max - $min);
}

// Driver code
{
    $arr = array(1, 2, 3, 4, 5);
    $n = sizeof($arr) / sizeof($arr[0]);
    $m = 4;
    echo find_difference($arr, $n, $m);
    return 0;
}

// This code is contributed by nitin mittal.
?>
```

Output:

```
4
```

We can optimize the above solution using more efficient approaches discussed in below post.
k largest(or smallest) elements in an array | added Min Heap method

Improved By : nitin mittal, SanyamAggarwal

Source

https://www.geeksforgeeks.org/difference-maximum-sum-minimum-sum-n-m-elementsin-review/

Chapter 41

Maximum distinct elements after removing k elements

Maximum distinct elements after removing k elements - GeeksforGeeks

Given an array **arr**[] containing **n** elements. The problem is to find maximum number of distinct elements after removing **k** elements from the array.
Note: 1 <= k <= n.

Examples:

```
Input : arr[] = {5, 7, 5, 5, 1, 2, 2}, k = 3
Output : 4
Remove 2 occurrences of element 5 and
1 occurrence of element 2.

Input : arr[] = {1, 2, 3, 4, 5, 6, 7}, k = 5
Output : 2
```

Approach: Following are the steps:

1. Create a hash table to store the frequency of each element.
2. Insert frequency of each element in a max heap.
3. Now, perform the following operation **k** times. Remove an element from the max heap. Decrement its value by 1. After this if element is not equal to 0, then again push the element in the max heap.
4. After the completion of step 3, the number of elements in the max heap is the required answer.

```
 // C++ implementation to find maximum distinct
// elements after removing k elements
```

```
#include <bits/stdc++.h>
using namespace std;

// function to find maximum distinct elements
// after removing k elements
int maxDistinctNum(int arr[], int n, int k)
{
    // 'um' implemented as hash table to store
    // frequency of each element
    unordered_map<int, int> um;

    // priority_queue 'pq' implemented as
    // max heap
    priority_queue<int> pq;

    // storing frequency of each element in 'um'
    for (int i = 0; i < n; i++)
        um[arr[i]]++;

    // inserting frequency of each element in 'pq'
    for (auto it = um.begin(); it != um.end(); it++)
        pq.push(it->second);

    while (k--) {

        // get the top element of 'pq'
        int temp = pq.top();

        // remove top element from 'pq'
        pq.pop();

        // decrement the popped element by 1
        temp--;

        // if true, then push the element in 'pq'
        if (temp != 0)
            pq.push(temp);
    }

    // required maximum distinct elements
    return ((int)pq.size());
}

// Driver program to test above
int main()
{
    int arr[] = { 5, 7, 5, 5, 1, 2, 2 };
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int k = 3;
    cout << "Maximum distinct elements = "
         << maxDistinctNum(arr, n, k);
    return 0;
}
```

Output:

```
Maximum distinct elements = 4
```

Time Complexity: O(k*logd), where **d** is the number of distinct elements in the given array.

Source

https://www.geeksforgeeks.org/maximum-distinct-elements-removing-k-elements/

Chapter 42

Median in a stream of integers (running integers)

Median in a stream of integers (running integers) - GeeksforGeeks

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
#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.

Median of Stream of Running Integers using STL

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/

Chapter 43

Median of Stream of Running Integers using STL

Median of Stream of Running Integers using STL - GeeksforGeeks

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 calulate 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 C++ 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)

Source

https://www.geeksforgeeks.org/median-of-stream-of-running-integers-using-stl/

Chapter 44

Merge K sorted linked lists | Set 1

Merge K sorted linked lists | Set 1 - GeeksforGeeks

Given K sorted linked lists of size N each, merge them and print the sorted output.

Example:

```
Input: k = 3, n =  4
list1 = 1->3->5->7->NULL
list2 = 2->4->6->8->NULL
list3 = 0->9->10->11

Output:
0->1->2->3->4->5->6->7->8->9->10->11
```

Method 1 (Simple)

A Simple Solution is to initialize result as first list. Now traverse all lists starting from second list. Insert every node of currently traversed list into result in a sorted way. Time complexity of this solution is $O(N^2)$ where N is total number of nodes, i.e., N = kn.

Method 2 (Using Min Heap)

A **Better solution** is to use Min Heap based solution which is discussed here for arrays. Time complexity of this solution would be O(nk Log k)

Method 3 (Using Divide and Conquer))

In this post, **Divide and Conquer** approach is discussed. This approach doesn't require extra space for heap and works in O(nk Log k)

We already know that merging of two linked lists can be done in O(n) time and O(1) space (For arrays O(n) space is required). The idea is to pair up K lists and merge each pair in linear time using O(1) space. After first cycle, K/2 lists are left each of size 2*N. After second cycle, K/4 lists are left each of size 4*N and so on. We repeat the procedure until we have only one list left.

Below is C++ implementation of the above idea.

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

// A Linked List node
struct Node
{
    int data;
    Node* next;
};

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Takes two lists sorted in increasing order, and merge
   their nodes together to make one big sorted list. Below
   function takes O(Log n) extra space for recursive calls,
   but it can be easily modified to work with same time and
   O(1) extra space  */
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if(b == NULL)
        return (a);
```

```
    /* Pick either a or b, and recur */
    if(a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}

// The main function that takes an array of lists
// arr[0..last] and generates the sorted output
Node* mergeKLists(Node* arr[], int last)
{
    // repeat until only one list is left
    while (last != 0)
    {
        int i = 0, j = last;

        // (i, j) forms a pair
        while (i < j)
        {
            // merge List i with List j and store
            // merged list in List i
            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair
            i++, j--;

            // If all pairs are merged, update last
            if (i >= j)
                last = j;
        }
    }

    return arr[0];
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
```

```
    temp->next = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // Merge all lists
    Node* head = mergeKLists(arr, k - 1);

    printList(head);

    return 0;
}
```

Output :

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Time Complexity of above algorithm is O(nk logk) as outer while loop in function mergeK-Lists() runs log k times and every time we are processing nk elements.

Merge k sorted linked lists | Set 2 (Using Min Heap)

Source

https://www.geeksforgeeks.org/merge-k-sorted-linked-lists/

Chapter 45

Merge k sorted arrays | Set 1

Merge k sorted arrays | Set 1 - GeeksforGeeks

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
```

A **simple solution** is to create an output array of size n*k and one by one copy all arrays to it. Finally, sort the output array using any O(n Log n) sorting algorithm. This approach takes O(nk Log nk) time.

One efficient solution is to first merge arrays into groups of 2. After first merging, we have k/2 arrays. We again merge arrays in groups, now we have k/4 arrays. We keep doing it unit we have one array left. The time complexity of this solution would O(nk Log k). How? Every merging in first iteration would take 2n time (merging two arrays of size n). Since there are total k/2 merging, total time in first iteration would be O(nk). Next iteration would also take O(nk). There will be total O(Log k) iterations, hence time complexity is O(nk Log k)

Another efficient solution is to use Min Heap. This Min Heap based solution has same time complexity which is O(nk Log k). But for different sized arrays, this solution works much better.

Following is detailed algorithm.
1. Create an output array of size n*k.
2. Create a min heap of size k and insert 1st element in all the arrays into the heap

3. Repeat following steps n*k times.

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

b) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Following is C++ implementation of the above algorithm.

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

#define n 4

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

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

    // 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++)
    {
        harr[i].element = arr[i][0]; // Store the first element
        harr[i].i = i;  // index of array
        harr[i].j = 1;  // Index of next element to be stored from array
    }
    MinHeap hp(harr, k); // Create the 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*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
        else root.element =  INT_MAX; //INT_MAX is for infinite

        // 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
```

Time Complexity: The main step is 3rd step, the loop runs n*k times. In every iteration of loop, we call heapify which takes O(Logk) time. Therefore, the time complexity is O(nk Logk).

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

Thanks to vigneshfor suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : maxflex

Source

https://www.geeksforgeeks.org/merge-k-sorted-arrays/

Chapter 46

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

Merge k sorted arrays | Set 2 (Different Sized Arrays) - GeeksforGeeks

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.

```
 // 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
```

Source

https://www.geeksforgeeks.org/merge-k-sorted-arrays-set-2-different-sized-arrays/

Chapter 47

Merge k sorted linked lists | Set 2 (Using Min Heap)

Merge k sorted linked lists | Set 2 (Using Min Heap) - GeeksforGeeks

Given **k** sorted linked lists each of size **n**, merge them and print the sorted output.

Examples:

```
Input: k = 3, n =  4
list1 = 1->3->5->7->NULL
list2 = 2->4->6->8->NULL
list3 = 0->9->10->11

Output:
0->1->2->3->4->5->6->7->8->9->10->11
```

Source: Merge K sorted Linked Lists | Method 2

Approach: An efficient solution for the problem has been dicussed in **Method 3** of this post. Here another solution has been provided which the uses the **MIN HEAP** data structure. This solution is based on the min heap approach used to solve the problem 'merge k sorted arrays' which is discussed here.

```
 // C++ implementation to merge k sorted linked lists
// | Using MIN HEAP method
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    struct Node* next;
```

```
};

// 'compare' function used to build up the
// priority queue
struct compare {
    bool operator()(struct Node* a, struct Node* b)
    {
        return a->data > b->data;
    }
};

// function to merge k sorted linked lists
struct Node* mergeKSortedLists(struct Node* arr[], int k)
{
    struct Node* head = NULL, *last;

    // priority_queue 'pq' implemeted as min heap with the
    // help of 'compare' function
    priority_queue<Node*, vector<Node*>, compare> pq;

    // push the head nodes of all the k lists in 'pq'
    for (int i = 0; i < k; i++)
        pq.push(arr[i]);

    // loop till 'pq' is not empty
    while (!pq.empty()) {

        // get the top element of 'pq'
        struct Node* top = pq.top();
        pq.pop();

        // check if there is a node next to the 'top' node
        // in the list of which 'top' node is a member
        if (top->next != NULL)
            // push the next node in 'pq'
            pq.push(top->next);

        // if final merged list is empty
        if (head == NULL) {
            head = top;

            // points to the last node so far of
            // the final merged list
            last = top;
        }

        else {
            // insert 'top' at the end of the merged list so far
```

```
            last->next = top;

            // update the 'last' pointer
            last = top;
        }
    }

    // head node of the required merged list
    return head;
}

// function to print the singly linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Utility function to create a new node
struct Node* newNode(int data)
{
    // allocate node
    struct Node* new_node = new Node();

    // put in the data
    new_node->data = data;
    new_node->next = NULL;

    return new_node;
}

// Driver program to test above
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    // creating k = 3 sorted lists
    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);
```

```
    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // merge the k sorted lists
    struct Node* head = mergeKSortedLists(arr, k);

    // print the merged list
    printList(head);

    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Time Complexity: O(nk Logk)
Auxiliary Space: O(k)

Source

https://www.geeksforgeeks.org/merge-k-sorted-linked-lists-set-2-using-min-heap/

Chapter 48

Merge two binary Max Heaps

Merge two binary Max Heaps - GeeksforGeeks

Given two binary max heaps as arrays, merge the given heaps.

Examples :

```
Input  : a = {10, 5, 6, 2},
         b = {12, 7, 9}
Output : {12, 10, 9, 2, 5, 7, 6}
```

The idea is simple. We create an array to store result. We copy both given arrays one by one to result. Once we have copied all elements, we call standard build heap to construct full merged max heap.

C++

```
 // C++ program to merge two max heaps.
#include <iostream>
using namespace std;

// Standard heapify function to heapify a
// subtree rooted under idx. It assumes
// that subtrees of node are already heapified.
void maxHeapify(int arr[], int n, int idx)
{
    // Find largest of node and its children
    if (idx >= n)
```

```
        return;
    int l = 2 * idx + 1;
    int r = 2 * idx + 2;
    int max;
    if (l < n && arr[l] > arr[idx])
        max = l;
    else
        max = idx;
    if (r < n && arr[r] > arr[max])
        max = r;

    // Put maximum value at root and
    // recur for the child with the
    // maximum value
    if (max != idx) {
        swap(arr[max], arr[idx]);
        maxHeapify(arr, n, max);
    }
}

// Builds a max heap of given arr[0..n-1]
void buildMaxHeap(int arr[], int n)
{
    // building the heap from first non-leaf
    // node by calling max heapify function
    for (int i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);
}

// Merges max heaps a[] and b[] into merged[]
void mergeHeaps(int merged[], int a[], int b[],
                int n, int m)
{
    // Copy elements of a[] and b[] one by one
    // to merged[]
    for (int i = 0; i < n; i++)
        merged[i] = a[i];
    for (int i = 0; i < m; i++)
        merged[n + i] = b[i];

    // build heap for the modified array of
    // size n+m
    buildMaxHeap(merged, n + m);
}

// Driver code
int main()
{
```

```
    int a[] = { 10, 5, 6, 2 };
    int b[] = { 12, 7, 9 };

    int n = sizeof(a) / sizeof(a[0]);
    int m = sizeof(b) / sizeof(b[0]);

    int merged[m + n];
    mergeHeaps(merged, a, b, n, m);

    for (int i = 0; i < n + m; i++)
        cout << merged[i] << " ";

    return 0;
}
```

Java

```
 // Java program to merge two max heaps.

class GfG {

    // Standard heapify function to heapify a
    // subtree rooted under idx. It assumes
    // that subtrees of node are already heapified.
    public static void maxHeapify(int[] arr, int n,
                                              int i)
    {
        // Find largest of node and its children
        if (i >= n) {
            return;
        }
        int l = i * 2 + 1;
        int r = i * 2 + 2;
        int max;
        if (l < n && arr[l] > arr[i]) {
            max = l;
        }
        else
            max = i;
        if (r < n && arr[r] > arr[max]) {
            max = r;
        }

        // Put maximum value at root and
        // recur for the child with the
        // maximum value
        if (max != i) {
            int temp = arr[max];
```

```
            arr[max] = arr[i];
            arr[i] = temp;
            maxHeapify(arr, n, max);
        }
    }

    // Merges max heaps a[] and b[] into merged[]
    public static void mergeHeaps(int[] arr, int[] a,
                                  int[] b, int n, int m)
    {
        for (int i = 0; i < n; i++) {
            arr[i] = a[i];
        }
        for (int i = 0; i < m; i++) {
            arr[n + i] = b[i];
        }
        n = n + m;

        // Builds a max heap of given arr[0..n-1]
        for (int i = n / 2 - 1; i >= 0; i--) {
            maxHeapify(arr, n, i);
        }
    }

    // Driver Code
    public static void main(String[] args)
    {
        int[] a = {10, 5, 6, 2};
        int[] b = {12, 7, 9};
        int n = a.length;
        int m = b.length;

        int[] merged = new int[m + n];

        mergeHeaps(merged, a, b, n, m);

        for (int i = 0; i < m + n; i++)
            System.out.print(merged[i] + " ");
        System.out.println();
    }
}
```

C#

```
 // C# program to merge two max heaps.
using System;

class GfG {
```

```
// Standard heapify function to heapify a
// subtree rooted under idx. It assumes
// that subtrees of node are already heapified.
public static void maxHeapify(int[] arr,
                              int n, int i)
{
    // Find largest of node
    // and its children
    if (i >= n) {
        return;
    }
    int l = i * 2 + 1;
    int r = i * 2 + 2;
    int max;
    if (l < n && arr[l] > arr[i]) {
        max = l;
    }
    else
        max = i;
    if (r < n && arr[r] > arr[max]) {
        max = r;
    }

    // Put maximum value at root and
    // recur for the child with the
    // maximum value
    if (max != i) {
        int temp = arr[max];
        arr[max] = arr[i];
        arr[i] = temp;
        maxHeapify(arr, n, max);
    }
}

// Merges max heaps a[] and b[] into merged[]
public static void mergeHeaps(int[] arr, int[] a,
                              int[] b, int n, int m)
{
    for (int i = 0; i < n; i++) {
        arr[i] = a[i];
    }
    for (int i = 0; i < m; i++) {
        arr[n + i] = b[i];
    }
    n = n + m;

    // Builds a max heap of given arr[0..n-1]
```

```
        for (int i = n / 2 - 1; i >= 0; i--) {
            maxHeapify(arr, n, i);
        }
    }

    // Driver Code
    public static void Main()
    {
        int[] a = {10, 5, 6, 2};
        int[] b = {12, 7, 9};
        int n = a.Length;
        int m = b.Length;

        int[] merged = new int[m + n];

        mergeHeaps(merged, a, b, n, m);

        for (int i = 0; i < m + n; i++)
            Console.Write(merged[i] + " ");
        Console.WriteLine();
    }
}

// This code is contributed by nitin mittal
```

Output:

```
12 10 9 2 5 7 6
```

Since time complexity for building the heap from array of n elements is O(n). The complexity of merging the heaps is equal to O(n + m).

Improved By : nitin mittal

Source

https://www.geeksforgeeks.org/merge-two-binary-max-heaps/

Chapter 49

Merge two sorted arrays in Python using heapq

Merge two sorted arrays in Python using heapq - GeeksforGeeks

Given two sorted arrays, the task is to merge them in a sorted manner.

Examples:

```
Input :  arr1 = [1, 3, 4, 5]
         arr2 = [2, 4, 6, 8]
Output : arr3 = [1, 2, 3, 4, 4, 5, 6, 8]

Input  : arr1 = [5, 8, 9]
         arr2 = [4, 7, 8]
Output : arr3 = [4, 5, 7, 8, 8, 9]
```

This problem has existing solution please refer Merge two sorted arrays link. We will solve this problem in python using **heapq.merge()** in a single line of code.

```
 # Function to merge two sorted arrays
from heapq import merge

def mergeArray(arr1,arr2):
    return list(merge(arr1, arr2))

# Driver function
if __name__ == "__main__":
    arr1 = [1,3,4,5]
    arr2 = [2,4,6,8]
    print mergeArray(arr1, arr2)
```

Output:

```
[1, 2, 3, 4, 4, 5, 6, 8]
```

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.
To create a heap, use a list initialized to [], or you can transform a populated list into a heap via function heapify().The following functions are provided:

- **heapq.heappush(heap,item) :** Push the value item onto the heap, maintaining the heap invariant.
- **heapq.heappop(heap) :** Pop and return the smallest item from the heap, maintaining the heap invariant. If the heap is empty, **IndexError** is raised. To access the smallest item without popping it, use heap[0].
- **heapq.heappushpop(heap, item) :** Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than heappush() followed by a separate call to heappop().
- **heapq.heapify(x) :** Transform list x into a heap, in-place, in linear time.
- **heapq.merge(*iterables) :** Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an iterator over the sorted values.

Source

https://www.geeksforgeeks.org/merge-two-sorted-arrays-python-using-heapq/

Chapter 50

Minimum increment/decrement to make array non-Increasing

Minimum increment/decrement to make array non-Increasing - GeeksforGeeks

Given an array a, your task is to convert it into a non-increasing form such that we can either increment or decrement the array value by 1 in minimum changes possible.

Examples :

```
Input : a[] = {3, 1, 2, 1}
Output : 1
Explanation :
We can convert the array into 3 1 1 1 by
changing 3rd element of array i.e. 2
into its previous integer 1 in one step
hence only one step is required.

Input : a[] = {3, 1, 5, 1}
Output : 4
We need to decrease 5 to 1 to make array sorted
in non-increasing order.

Input : a[] = {1, 5, 5, 5}
Output : 4
We need to increase 1 to 5.
```

Brute-Force approach : We consider both possibilities for every element and find the minimum of two possibilities.

Efficient Approach : Calculate the sum of absolute differences between the final array elements and the current array elements. Thus, the answer will be the sum of the difference

between the ith element and the smallest element occurred till then. For this, we can maintain a min-heap to find the smallest element encountered till now. In the min-priority queue, we will put the elements and new elements are compared with the previous minimum. If new minimum is found we will update it, this is done because each of the next element which is coming should be smaller than the current minimum element found till. Here, we calculate the difference so that we can get how much we have to change the current number so that it will be equal or less than previous numbers encountered till. Lastly, the sum of all these difference will be our answer as this will give the final value upto which we have to change the elements.

Below is **C++** implementation of the above approach

```
 // CPP code to count the change required to
// convert the array into non-increasing array
#include <bits/stdc++.h>
using namespace std;

int DecreasingArray(int a[], int n)
{
    int sum = 0, dif = 0;

    // min heap
    priority_queue<int, vector<int>, greater<int> > pq;

    // Here in the loop we will
    // check that whether the upcoming
    // element of array is less than top
    // of priority queue. If yes then we
    // calculate the difference. After
    // that we will remove that element
    // and push the current element in
    // queue. And the sum is incremented
    // by the value of difference
    for (int i = 0; i < n; i++) {
        if (!pq.empty() && pq.top() < a[i]) {
            dif = a[i] - pq.top();
            sum += dif;
            pq.pop();
            pq.push(a[i]);
        }
        pq.push(a[i]);
    }

    return sum;
}

// Driver Code
int main()
{
```

```
    int a[] = { 3, 1, 2, 1 };
    int n = sizeof(a) / sizeof(a[0]);

    cout << DecreasingArray(a, n);

    return 0;
}
```

Output:

```
1
```

Time Complexity: **O(n log(n))**
Space Complexity: **O(n)**

Also see : Convert to strictly increasing array with minimum changes.

Source

https://www.geeksforgeeks.org/minimum-incrementdecrement-to-make-array-non-increasing/

Chapter 51

Minimum product of k integers in an array of positive Integers

Minimum product of k integers in an array of positive Integers - GeeksforGeeks

Given an array of n positive integers. We are required to write a program to print the minimum product of k integers of the given array.

Examples:

```
Input : 198 76 544 123 154 675
         k = 2
Output : 9348
We get minimum product after multiplying
76 and 123.

Input : 11 8 5 7 5 100
        k = 4
Output : 1400
```

The idea is simple, we find the smallest k elements and print multiplication of them. In below implementation, we have used simple Heap based approach where we insert array elements into a min heap and then find product of top k elements.

C++

```
 // CPP program to find minimum product of
// k elements in an array
#include <bits/stdc++.h>
using namespace std;

int minProduct(int arr[], int n, int k)
```

```
{
    priority_queue<int, vector<int>, greater<int> > pq;
    for (int i = 0; i < n; i++)
        pq.push(arr[i]);

    int count = 0, ans = 1;

    // One by one extract items from max heap
    while (pq.empty() == false && count < k) {
        ans = ans * pq.top();
        pq.pop();
        count++;
    }

    return ans;
}

// Driver code
int main()
{
    int arr[] = {198, 76, 544, 123, 154, 675};
    int k = 2;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum product is "
         << minProduct(arr, n, k);
    return 0;
}
```

Python3

```
 # Python3 program to find minimum
# product of k elements in an array
import math
import heapq

def minProduct(arr, n, k):

    heapq.heapify(arr)
    count = 0
    ans = 1

    # One by one extract
    # items from min heap
    while ( arr ) and count < k:
        x = heapq.heappop(arr)
        ans = ans * x
        count = count + 1
```

```
    return ans;

# Driver method
arr = [198, 76, 544, 123, 154, 675]
k = 2
n = len(arr)
print ("Minimum product is",
       minProduct(arr, n, k))
```

Output:

```
Minimum product is 9348
```

Time Complexity : O(n * log n)

Note that the above problem can be solved in O(n) time using methods discussed here and here.

Source

https://www.geeksforgeeks.org/minimum-product-k-integers-array-positive-integers/

Chapter 52

Minimum sum of two numbers formed from digits of an array

Minimum sum of two numbers formed from digits of an array - GeeksforGeeks

Given an array of digits (values are from 0 to 9), find the minimum possible sum of two numbers formed from digits of the array. All digits of given array must be used to form the two numbers.

Examples:

```
Input: [6, 8, 4, 5, 2, 3]
Output: 604
The minimum sum is formed by numbers
358 and 246

Input: [5, 3, 0, 7, 4]
Output: 82
The minimum sum is formed by numbers
35 and 047
```

Since we want to minimize the sum of two numbers to be formed, we must divide all digits in two halves and assign half-half digits to them. We also need to make sure that the leading digits are smaller.
We build a Min Heap with the elements of the given array, which takes O(n) worst time. Now we retrieve min values (2 at a time) of array, by polling from the Priority Queue and append these two min values to our numbers, till the heap becomes empty, i.e., all the elements of array get exhausted. We return the sum of two formed numbers, which is our required answer.

C/C++

```
 // C++ program to find minimum sum of two numbers
// formed from all digits in a given array.
#include<bits/stdc++.h>
using namespace std;

// Returns sum of two numbers formed
// from all digits in a[]
int minSum(int arr[], int n)
{
    // min Heap
    priority_queue <int, vector<int>, greater<int> > pq;

    // to store the 2 numbers formed by array elements to
    // minimize the required sum
    string num1, num2;

    // Adding elements in Priority Queue
    for(int i=0; i<n; i++)
        pq.push(arr[i]);

    // checking if the priority queue is non empty
    while(!pq.empty())
    {
        // appending top of the queue to the string
        num1+=(48 + pq.top());
        pq.pop();
        if(!pq.empty())
        {
            num2+=(48 + pq.top());
            pq.pop();
        }
    }

    // converting string to integer
    int a = atoi(num1.c_str());
    int b = atoi(num2.c_str());

    // returning the sum
    return a+b;
}

int main()
{
    int arr[] = {6, 8, 4, 5, 2, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout<<minSum(arr, n)<<endl;
    return 0;
}
```

```
// Contributed By: Harshit Sidhwa
```

Java

```
 // Java program to find minimum sum of two numbers
// formed from all digits in a given array.
import java.util.PriorityQueue;

class MinSum
{
    // Returns sum of two numbers formed
    // from all digits in a[]
    public static long solve(int[] a)
    {
        // min Heap
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

        // to store the 2 numbers formed by array elements to
        // minimize the required sum
        StringBuilder num1 = new StringBuilder();
        StringBuilder num2 = new StringBuilder();

        // Adding elements in Priority Queue
        for (int x : a)
            pq.add(x);

        // checking if the priority queue is non empty
        while (!pq.isEmpty())
        {
            num1.append(pq.poll()+ "");
            if (!pq.isEmpty())
                num2.append(pq.poll()+ "");
        }

        // the required sum calculated
        long sum = Long.parseLong(num1.toString()) +
                   Long.parseLong(num2.toString());

        return sum;
    }

    // Driver code
    public static void main (String[] args)
    {
        int arr[] = {6, 8, 4, 5, 2, 3};
        System.out.println("The required sum is "+ solve(arr));
    }
}
```

Output:

```
The required sum is 604
```

Source

https://www.geeksforgeeks.org/minimum-sum-two-numbers-formed-digits-array-2/

Chapter 53

Number of ways to form a heap with n distinct integers

Number of ways to form a heap with n distinct integers - GeeksforGeeks

Given n, how many distinctMax Heapcan be made from n distinct integers?

Examples:

```
Input : n = 3
Output : Assume the integers are 1, 2, 3.
Then the 2 possible max heaps are:
           3
          / \
         1   2

          3
         / \
        2   1

Input : n = 4
Output : Assume the integers are 1, 2, 3, 4.
Then the 3 possible max heaps are:
        4
       / \
      3   2
     /
    1

        4
       / \
      2   3
```

```
 /
1

    4
   / \
  3   1
 /
2
```

Since there is only one element as the **root**, it must be the largest number. Now we have n-1 remaining elements. The main observation here is that because of the max heap properties, the **structure** of the heap nodes will remain the same in all instances, but only the values in the nodes will change.

Assume there are **l** elements in the **left sub-tree** and **r** elements in the **right sub-tree**. Now for the root, l + r = n-1. From this we can see that we can **choose** any l of the remaining n-1 elements for the left sub-tree as they are all smaller than the root.

We know the there are $\binom{n-1}{l}$ ways to do this. Next for each instance of these, we can have many heaps with l elements and for each of those we can have many heaps with r elements. Thus we can consider them as subproblems and **recur** for the final answer as:

T(n) = $\binom{n-1}{l}$ * T(L) * T(R).

Now we have to find the **values** for l and r for a given n. We know that the height of the heap h = $\log_2 n$. Also the maximum number of elements that can be present in the h th **level** of any heap, m = 2^h, where the root is at the 0th level. Moreover the number of elements actually present in the last level of the heap p = n – (2^h – 1). (since $2^{h-1} - 1$ number of nodes present till the penultimate level). Thus, there can be two **cases**: when the last level is more than or equal to half-filled:

l = 2^h – 1, if p >= m / 2
(or) the last level is less than half-filled:

l = 2^h – 1 – ((m / 2) – p), if p < m / 2

(we get 2^h – 1 here because left subtree has [illegible] nodes.
From this we can also say that r = n – l – 1.

We can use the **dynamic programming** approach discussed in this post here to find the values of $\binom{n-1}{l}$. Similarly if we look at the recursion tree for the **optimal substructure** recurrence formed above, we can see that it also has **overlapping subproblems** property, hence can be solved using dynamic programming:

```
T(7)
```

```
          /    \
        T(3)   T(3)
       /  \     /  \
    T(1)  T(1) T(1) T(1)
```

Following is the c++ implementation of the above approach:

```
 // CPP program to count max heaps with n distinct keys
#include <iostream>
using namespace std;

#define MAXN 105 // maximum value of n here

// dp[i] = number of max heaps for i distinct integers
int dp[MAXN];

// nck[i][j] = number of ways to choose j elements
//             form i elements, no order */
int nck[MAXN][MAXN];

// log2[i] = floor of logarithm of base 2 of i
int log2[MAXN];

// to calculate nCk
int choose(int n, int k)
{
    if (k > n)
        return 0;
    if (n <= 1)
        return 1;
    if (k == 0)
        return 1;

    if (nck[n][k] != -1)
        return nck[n][k];

    int answer = choose(n - 1, k - 1) + choose(n - 1, k);
    nck[n][k] = answer;
    return answer;
}

// calculate l for give value of n
int getLeft(int n)
{
    if (n == 1)
        return 0;

    int h = log2[n];
```

```
    // max number of elements that can be present in the
    // hth level of any heap
    int numh = (1 << h); //(2 ^ h)

    // number of elements that are actually present in
    // last level(hth level)
    // (2^h - 1)
    int last = n - ((1 << h) - 1);

    // if more than half-filled
    if (last >= (numh / 2))
        return (1 << h) - 1; // (2^h) - 1
    else
        return (1 << h) - 1 - ((numh / 2) - last);
}

// find maximum number of heaps for n
int numberOfHeaps(int n)
{
    if (n <= 1)
        return 1;

    if (dp[n] != -1)
        return dp[n];

    int left = getLeft(n);
    int ans = (choose(n - 1, left) * numberOfHeaps(left)) *
                            (numberOfHeaps(n - 1 - left));
    dp[n] = ans;
    return ans;
}

// function to initialize arrays
int solve(int n)
{
    for (int i = 0; i <= n; i++)
        dp[i] = -1;

    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
            nck[i][j] = -1;

    int currLog2 = -1;
    int currPower2 = 1;

    // for each power of two find logarithm
    for (int i = 1; i <= n; i++) {
```

```
        if (currPower2 == i) {
            currLog2++;
            currPower2 *= 2;
        }
        log2[i] = currLog2;
    }

    return numberOfHeaps(n);
}

// driver function
int main()
{
    int n = 10;
    cout << solve(n) << endl;
    return 0;
}
```

Output:

```
3360
```

Asked in: Directi.

Improved By : rsatish1110

Source

https://www.geeksforgeeks.org/number-ways-form-heap-n-distinct-integers/

Chapter 54

Overview of Data Structures | Set 2 (Binary Tree, BST, Heap and Hash)

Overview of Data Structures | Set 2 (Binary Tree, BST, Heap and Hash) - GeeksforGeeks

We have discussed Overview of Array, Linked List, Queue and Stack. In this article following Data Structures are discussed.

5. Binary Tree
6. Binary Search Tree
7. Binary Heap
9. Hashing

Binary Tree
Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.
A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

Binary Tree Representation: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL. A Binary Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

A Binary Tree can be traversed in two ways:
Depth First Traversal: Inorder (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)
Breadth First Traversal: Level Order Traversal

Binary Tree Properties:

```
The maximum number of nodes at level 'l' = 2l-1.

Maximum number of nodes = 2h - 1.
Here h is height of a tree. Height is considered
as is maximum number of nodes on root to leaf path

Minimum possible height =  ceil(Log2(n+1))

In Binary tree, number of leaf nodes is always one
more than nodes with two children.

Time Complexity of Tree Traversal: O(n)
```

Examples : One reason to use binary tree or tree in general is for the things that form a hierarchy. They are useful in File structures where each file is located in a particular directory and there is a specific hierarchy associated with files and directories. Another example where Trees are useful is storing heirarchical objects like JavaScript Document Object Model considers HTML page as a tree with nesting of tags as parent child relations.

Binary Search Tree
In Binary Search Tree is a Binary Tree with following additional properties:
1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Time Complexities:

```
Search :  O(h)
Insertion : O(h)
Deletion : O(h)
Extra Space : O(n) for pointers

h: Height of BST
n: Number of nodes in BST

If Binary Search Tree is Height Balanced,
then h = O(Log n)

Self-Balancing BSTs such as AVL Tree, Red-Black
Tree and Splay Tree make sure that height of BST
remains O(Log n)
```

BST provide moderate access/search (quicker than Linked List and slower than arrays).
BST provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists).

Examples : Its main use is in search application where data is constantly entering/leaving and data needs to printed in sorted order. For example in implementation in E- commerce

websites where a new product is added or product goes out of stock and all products are lised in sorted order.

Binary Heap
A Binary Heap is a Binary Tree with following properties.
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap. It is mainly implemented using array.

```
Get Minimum in Min Heap: O(1) [Or Get Max in Max Heap]
Extract Minimum Min Heap: O(Log n) [Or Extract Max in Max Heap]
Decrease Key in Min Heap: O(Log n)  [Or Extract Max in Max Heap]
Insert: O(Log n)
Delete: O(Log n)
```

Example : Used in implementing efficient priority-queues, which in turn are used for scheduling processes in operating systems. Priority Queues are also used in Dijstra's and Prim's graph algorithms.
The Heap data structure can be used to efficiently find the k smallest (or largest) elements in an array, merging k sorted arrays, median of a stream, etc.
Heap is a special data structure and it cannot be used for searching of a particular element.

HashingHash Function: A function that converts a given big input key to a small practical integer value. The mapped integer value is used as an index in hash table. A good hash function should have following properties
1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

Chaining:The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

```
Space : O(n)
Search    : O(1) [Average]    O(n) [Worst case]
Insertion : O(1) [Average]    O(n) [Worst Case]
Deletion  : O(1) [Average]    O(n) [Worst Case]
```

Hashing seems better than BST for all the operations. But in hashing, elements are unordered and in BST elements are stored in an ordered manner. Also BST is easy to implement but hash functions can sometimes be very complex to generate. In BST, we can also efficiently find floor and ceil of values.

Example : Hashing can be used to remove duplicates from a set of elements. Can also be used find frequency of all items. For example, in web browsers, we can check visited urls using hashing. In firewalls, we can use hashing to detect spam. We need to hash IP addresses. Hashing can be used in any situation where want search() insert() and delete() in O(1) time.

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : Rahul1421

Source

https://www.geeksforgeeks.org/overview-of-data-structures-set-2-binary-tree-bst-heap-and-hash/

Chapter 55

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

Print all elements in sorted order from row and column wise sorted matrix - GeeksforGeeks

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.

This article is contributed by **Varun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/print-elements-sorted-order-row-column-wise-sorted-matrix/

Chapter 56

Print all nodes less than a value x in a Min Heap.

Print all nodes less than a value x in a Min Heap. - GeeksforGeeks

Given a binary min heap and a value x, print all the binary heap nodes having value less than the given value x.

```
Examples : Consider the below min heap as
        common input two both below examples.
                   2
                 /     \
                3        15
              /    \     / \
             5       4  45  80
            / \     / \
           6   150 77 120

Input  : x = 15
Output : 2 3 5 6 4

Input  : x = 80
Output : 2 3 5 6 4 77 15 45
```

The idea is to do a preorder traversal of the give Binary heap. While doing preorder traversal, if the value of a node is greater than the given value x, we return to the previous recursive call. Because all children nodes in a min heap are greater than the parent node. Otherwise we print current node and recur for its children.

```
 // A C++ program to print all values
// smaller than a given value in Binary
```

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

// A class for Min Heap
class MinHeap
{
    // pointer to array of elements in heap
    int *harr;

    // maximum possible size of min heap
    int capacity;
    int heap_size; // Current no. of elements.
public:
    // Constructor
    MinHeap(int capacity);

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

    int parent(int i) { return (i-1)/2; }
    int left(int i) { return (2*i + 1); }
    int right(int i) { return (2*i + 2); }

    // Inserts a new key 'k'
    void insertKey(int k);

    //Function to print all nodes smaller than k
    void printSmallerThan(int k, int pos);
};

// Function to print all elements smaller than k
void MinHeap::printSmallerThan(int x, int pos=0)
{
    /* Make sure item exists */
    if (pos >= heap_size)
        return;

    if (harr[pos] >= x)
    {
        /* Skip this node and its descendants,
          as they are all >= x . */
        return;
    }

    cout << harr[pos] << " ";
```

```
    printSmallerThan(x, left(pos));
    printSmallerThan(x, right(pos));
}

// Constructor: Builds a heap from a given
// array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(harr[i], harr[parent(i)]);
       i = parent(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] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
```

```
    {
        swap(harr[i], harr[smallest]);
        MinHeapify(smallest);
    }
}

// Driver program to test above functions
int main()
{
    MinHeap h(50);
    h.insertKey(3);
    h.insertKey(2);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    h.insertKey(80);
    h.insertKey(6);
    h.insertKey(150);
    h.insertKey(77);
    h.insertKey(120);

    // Print all nodes smaller than 100.
    int x = 100;
    h.printSmallerThan(x);

    return 0;
}
```

Output:

```
2 3 5 6 4 77 15 45 80
```

Source

https://www.geeksforgeeks.org/print-all-nodes-less-than-a-value-x-in-a-min-heap/

Chapter 57

Priority Queue in Python

Priority Queue in Python - GeeksforGeeks

Priority Queue is an extension of the queue with following properties.
1) An element with high priority is dequeued before an element with low priority.
2) If two elements have the same priority, they are served according to their order in the queue.

Below is **simple implementation** of priority queue.

```
 # A simple implementation of Priority Queue
# using Queue.
class PriorityQueue(object):
    def __init__(self):
        self.queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    # for checking if the queue is empty
    def isEmpty(self):
        return len(self.queue) == []

    # for inserting an element in the queue
    def insert(self, data):
        self.queue.append(data)

    # for popping an element based on Priority
    def delete(self):
        try:
            max = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max]:
                    max = i
```

```
            item = self.queue[max]
            del self.queue[max]
            return item
        except IndexError:
            print()
            exit()

if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue)
    while not myQueue.isEmpty():
        print(myQueue.delete())
```

Output:

```
12 1 14 7
14
12
7
1
()
```

Note that the time complexity of delete is O(n) in above code.

A **better implementation** is to use Binary Heap which is typically used to implement priority queue. Note that Python provides **heapq** in library also.

Source

https://www.geeksforgeeks.org/priority-queue-in-python/

Chapter 58

Priority queue of pairs in C++ (Ordered by first)

Priority queue of pairs in C++ (Ordered by first) - GeeksforGeeks

In C++, priority_queue implements heap. Below are some examples of creating priority queue of pair type.

Max Priority queue (Or Max heap) ordered by first element

```
 // C++ program to create a priority queue of pairs.
// By default a max heap is created ordered
// by first element of pair.
#include <bits/stdc++.h>
using namespace std;

// Driver program to test methods of graph class
int main()
{
    // By default a max heap is created ordered
    // by first element of pair.
    priority_queue<pair<int, int> > pq;

    pq.push(make_pair(10, 200));
    pq.push(make_pair(20, 100));
    pq.push(make_pair(15, 400));

    pair<int, int> top = pq.top();
    cout << top.first << " " << top.second;
    return 0;
}
```

Output :

```
20 100
```

Min Priority queue (Or Min heap) ordered by first element

```
 // C++ program to create a priority queue of pairs.
// We can create a min heap by passing adding two
// parameters, vector and greater().
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi;

// Driver program to test methods of graph class
int main()
{
    // By default a min heap is created ordered
    // by first element of pair.
    priority_queue<pi, vector<pi>, greater<pi> > pq;

    pq.push(make_pair(10, 200));
    pq.push(make_pair(20, 100));
    pq.push(make_pair(15, 400));

    pair<int, int> top = pq.top();
    cout << top.first << " " << top.second;
    return 0;
}
```

Output :

```
10 100
```

Source

https://www.geeksforgeeks.org/priority-queue-of-pairs-in-c-ordered-by-first/

Chapter 59

Program for Preemptive Priority CPU Scheduling

Program for Preemptive Priority CPU Scheduling - GeeksforGeeks

Implementing priority CPU scheduling. In this problem, we are using Min Heap as the data structure for implementing priority scheduling.
In this problem smaller numbers denote higher priority.
The following functions are used in the given code below:

```
struct process {
   processID,
   burst time,
   response time,
   priority,
   arrival time.
}
```

void quicksort(process array[], low, high)– This function is used to arrange the processes in ascending order according to their arrival time.
int partition(process array[], int low, int high)– This function is used to partition the array for sorting.
void insert(process Heap[], process value, int *heapsize, int *currentTime)– It is used to include all the valid and eligible processes in the heap for execution. heapsize defines the number of processes in execution depending on the current time currentTime keeps record of the current CPU time.
void order(process Heap[], int *heapsize, int start)– It is used to reorder the heap according to priority if the processes after insertion of new process.
void extractminimum(process Heap[], int *heapsize, int *currentTime)– This function is used to find the process with highest priority from the heap. It also reorders the heap after extracting the highest priority process.
void scheduling(process Heap[], process array[], int n, int *heapsize, int *currentTime)– This function is responsible for executing the highest priority extracted from

Heap[].
void process(process array[], int n)– This function is responsible for managing the entire execution of the processes as they arrive in the CPU according to their arrival time.

```
 // CPP program to implement preemptive priority scheduling
#include <bits/stdc++.h>
using namespace std;

struct Process {
    int processID;
    int burstTime;
    int tempburstTime;
    int responsetime;
    int arrivalTime;
    int priority;
    int outtime;
    int intime;
};

// It is used to include all the valid and eligible
// processes in the heap for execution. heapsize defines
// the number of processes in execution depending on
// the current time currentTime keeps a record of
// the current CPU time.
void insert(Process Heap[], Process value, int* heapsize,
            int* currentTime)
{
    int start = *heapsize, i;
    Heap[*heapsize] = value;
    if (Heap[*heapsize].intime == -1)
        Heap[*heapsize].intime = *currentTime;
    ++(*heapsize);

    // Ordering the Heap
    while (start != 0 && Heap[(start - 1) / 2].priority >
                                 Heap[start].priority) {
        Process temp = Heap[(start - 1) / 2];
        Heap[(start - 1) / 2] = Heap[start];
        Heap[start] = temp;
        start = (start - 1) / 2;
    }
}

// It is used to reorder the heap according to
// priority if the processes after insertion
// of new process.
void order(Process Heap[], int* heapsize, int start)
{
```

```
    int smallest = start;
    int left = 2 * start + 1;
    int right = 2 * start + 2;
    if (left < *heapsize && Heap[left].priority <
                            Heap[smallest].priority)
        smallest = left;
     if (right < *heapsize && Heap[right].priority <
                           Heap[smallest].priority)
        smallest = right;

    // Ordering the Heap
    if (smallest != start) {
        Process temp = Heap[smallest];
        Heap[smallest] = Heap[start];
        Heap[start] = temp;
        order(Heap, heapsize, smallest);
    }
}

// This function is used to find the process with
// highest priority from the heap. It also reorders
// the heap after extracting the highest priority process.
Process extractminimum(Process Heap[], int* heapsize,
                       int* currentTime)
{
    Process min = Heap[0];
    if (min.responsetime == -1)
        min.responsetime = *currentTime - min.arrivalTime;
    --(*heapsize);
    if (*heapsize >= 1) {
        Heap[0] = Heap[*heapsize];
        order(Heap, heapsize, 0);
    }
    return min;
}

// Compares two intervals according to staring times.
bool compare(Process p1, Process p2)
{
    return (p1.arrivalTime < p2.arrivalTime);
}

// This function is responsible for executing
// the highest priority extracted from Heap[].
void scheduling(Process Heap[], Process array[], int n,
                int* heapsize, int* currentTime)
{
    if (heapsize == 0)
```

```
        return;

    Process min = extractminimum(Heap, heapsize, currentTime);
    min.outtime = *currentTime + 1;
    --min.burstTime;
    printf("process id = %d current time = %d\n",
           min.processID, *currentTime);

    // If the process is not yet finished
    // insert it back into the Heap*/
    if (min.burstTime > 0) {
        insert(Heap, min, heapsize, currentTime);
        return;
    }

    for (int i = 0; i < n; i++)
        if (array[i].processID == min.processID) {
            array[i] = min;
            break;
        }
}

// This function is responsible for
// managing the entire execution of the
// processes as they arrive in the CPU
// according to their arrival time.
void priority(Process array[], int n)
{
    sort(array, array + n, compare);

    int totalwaitingtime = 0, totalbursttime = 0,
        totalturnaroundtime = 0, i, insertedprocess = 0,
        heapsize = 0, currentTime = array[0].arrivalTime,
        totalresponsetime = 0;

    Process Heap[4 * n];

    // Calculating the total burst time
    // of the processes
    for (int i = 0; i < n; i++) {
        totalbursttime += array[i].burstTime;
        array[i].tempburstTime = array[i].burstTime;
    }

    // Inserting the processes in Heap
    // according to arrival time
    do {
        if (insertedprocess != n) {
```

```
            for (i = 0; i < n; i++) {
                if (array[i].arrivalTime == currentTime) {
                    ++insertedprocess;
                    array[i].intime = -1;
                    array[i].responsetime = -1;
                    insert(Heap, array[i], &heapsize, &currentTime);
                }
            }
        }
        scheduling(Heap, array, n, &heapsize, &currentTime);
        ++currentTime;
        if (heapsize == 0 && insertedprocess == n)
            break;
    } while (1);

    for (int i = 0; i < n; i++) {
        totalresponsetime += array[i].responsetime;
        totalwaitingtime += (array[i].outtime - array[i].intime -
                                        array[i].tempburstTime);
        totalbursttime += array[i].burstTime;
    }
    printf("Average waiting time = %f\n",
           ((float)totalwaitingtime / (float)n));
    printf("Average response time =%f\n",
           ((float)totalresponsetime / (float)n));
    printf("Average turn around time = %f\n",
           ((float)(totalwaitingtime + totalbursttime) / (float)n));
}

// Driver code
int main()
{
    int n, i;
    Process a[5];
    a[0].processID = 1;
    a[0].arrivalTime = 4;
    a[0].priority = 2;
    a[0].burstTime = 6;
    a[1].processID = 4;
    a[1].arrivalTime = 5;
    a[1].priority = 1;
    a[1].burstTime = 3;
    a[2].processID = 2;
    a[2].arrivalTime = 5;
    a[2].priority = 3;
    a[3].burstTime = 7;
    a[3].processID = 3;
    a[3].arrivalTime = 1;
```

```
    a[3].priority = 4;
    a[3].burstTime = 2;
    a[4].processID = 5;
    a[4].arrivalTime = 3;
    a[4].priority = 5;
    a[4].burstTime = 4;
    priority(a, 5);
    return 0;
}
```

Output:

```
process id = 3 current time = 1
process id = 3 current time = 2
process id = 5 current time = 3
process id = 1 current time = 4
process id = 4 current time = 5
process id = 4 current time = 6
process id = 4 current time = 7
process id = 1 current time = 8
process id = 1 current time = 9
process id = 1 current time = 10
process id = 1 current time = 11
process id = 1 current time = 12
process id = 2 current time = 13
process id = 5 current time = 14
process id = 5 current time = 15
process id = 5 current time = 16
Average waiting time = 4.400000
Average response time =1.600000
Average turn around time = 7.200000
```

The output displays the order in which the processes are executed in the memory and also shows the average waiting time, average response time and average turn around time for each process.

Source

https://www.geeksforgeeks.org/program-for-preemptive-priority-cpu-scheduling/

Chapter 60

Python Code for time Complexity plot of Heap Sort

Python Code for time Complexity plot of Heap Sort - GeeksforGeeks

Prerequisite : HeapSort
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

We implement Heap Sort here, call it for different sized random lists, measure time taken for different sizes and generate a plot of input size vs time taken.

```
 # Python Code for Implementation and running time Algorithm
# Complexity plot of Heap Sort
# by Ashok Kajal
# This python code intends to implement Heap Sort Algorithm
# Plots its time Complexity on list of different sizes

# ---------------------Important Note -------------------
# numpy, time and matplotlib.pyplot are required to run this code
import time
from numpy.random import seed
from numpy.random import randint
import matplotlib.pyplot as plt


# find left child of node i
def left(i):
    return 2 * i + 1

# find right child of node i
def right(i):
```

```
    return 2 * i + 2

# calculate and return array size
def heapSize(A):
    return len(A)-1


# This fuction takes an array and Heapyfies
# the at node i
def MaxHeapify(A, i):
    # print("in heapy", i)
    l = left(i)
    r = right(i)

    # heapSize = len(A)
    # print("left", l, "Rightt", r, "Size", heapSize)
    if l<= heapSize(A) and A[l] > A[i] :
        largest = l
    else:
        largest = i
    if r<= heapSize(A) and A[r] > A[largest]:
        largest = r
    if largest != i:
       # print("Largest", largest)
        A[i], A[largest]= A[largest], A[i]
       # print("List", A)
        MaxHeapify(A, largest)

# this function makes a heapified array
def BuildMaxHeap(A):
    for i in range(int(heapSize(A)/2)-1, -1, -1):
        MaxHeapify(A, i)

# Sorting is done using heap of array
def HeapSort(A):
    BuildMaxHeap(A)
    B = list()
    heapSize1 = heapSize(A)
    for i in range(heapSize(A), 0, -1):
        A[0], A[i]= A[i], A[0]
        B.append(A[heapSize1])
        A = A[:-1]
        heapSize1 = heapSize1-1
        MaxHeapify(A, 0)


# randomly generates list of different
# sizes and call HeapSort funtion
```

```
elements = list()
times = list()
for i in range(1, 10):

    # generate some integers
    a = randint(0, 1000 * i, 1000 * i)
    # print(i)
    start = time.clock()
    HeapSort(a)
    end = time.clock()

    # print("Sorted list is ", a)
    print(len(a), "Elements Sorted by HeapSort in ", end-start)
    elements.append(len(a))
    times.append(end-start)

plt.xlabel('List Length')
plt.ylabel('Time Complexity')
plt.plot(elements, times, label ='Heap Sort')
plt.grid()
plt.legend()
plt.show()
# This code is contributed by Ashok Kajal
```

Output :

```
Input : Unsorted Lists of Different sizes are Generated Randomly
Output :
1000 Elements Sorted by HeapSort in  0.023797415087301488
2000 Elements Sorted by HeapSort in  0.053856713614550245
3000 Elements Sorted by HeapSort in  0.08474737185133563
4000 Elements Sorted by HeapSort in  0.13578669978414837
5000 Elements Sorted by HeapSort in  0.1658182863213824
6000 Elements Sorted by HeapSort in  0.1875901601906662
7000 Elements Sorted by HeapSort in  0.21982946862249264
8000 Elements Sorted by HeapSort in  0.2724293921580738
9000 Elements Sorted by HeapSort in  0.30996323029421546

Complexity PLot for Heap Sort is Given  Below
```

Source

https://www.geeksforgeeks.org/python-code-for-time-complexity-plot-of-heap-sort/

Chapter 61

Python heapq to find K'th smallest element in a 2D array

Python heapq to find K'th smallest element in a 2D array - GeeksforGeeks

Given an n x n matrix and integer k. Find the k'th smallest element in the given 2D array.

Examples:

```
Input : mat = [[10, 25, 20, 40],
               [15, 45, 35, 30],
               [24, 29, 37, 48],
               [32, 33, 39, 50]]
        k =  7
Output : 7th smallest element is 30
```

We will use similar approach like K'th Smallest/Largest Element in Unsorted Array to solve this problem.

1. Create an empty min heap using heapq in python.
2. Now assign first row (list) in result variable and convert result list into min heap using **heapify** method.
3. Now traverse remaining row elements and push them into created min heap.
4. Now get k'th smallest element using **nsmallest(k, iterable)** method of heapq module.

```
 # Function to find K'th smallest element in
# a 2D array in Python
import heapq

def kthSmallest(input):
```

```
    # assign first row to result variable
    # and convert it into min heap
    result = input[0]
    heapq.heapify(result)

    # now traverse remaining rows and push
    # elements in min heap
    for row in input[1:]:
         for ele in row:
              heapq.heappush(result,ele)

    # get list of first k smallest element because
    # nsmallest(k,list) method returns first k
    # smallest element now print last element of
    # that list
    kSmallest = heapq.nsmallest(k,result)
    print (k,"th smallest element is ",kSmallest[-1])

# Driver program
if __name__ == "__main__":
    input = [[10, 25, 20, 40],
            [15, 45, 35, 30],
            [24, 29, 37, 48],
            [32, 33, 39, 50]]
    k = 7
    kthSmallest(input)
```

Output:

```
7th smallest element is 30
```

Source

https://www.geeksforgeeks.org/python-heapq-find-kth-smallest-element-2d-array/

Chapter 62

Rearrange characters in a string such that no two adjacent are same

Rearrange characters in a string such that no two adjacent are same - GeeksforGeeks

Given a string with repeated characters, task is rearrange characters in a string so that no two adjacent characters are same.

Note : It may be assumed that the string has only lowercase English alphabets.

Examples:

```
Input: aaabc
Output: abaca

Input: aaabb
Output: ababa

Input: aa
Output: Not Possible

Input: aaaabc
Output: Not Possible
```

Asked In : Amazon Interview

Prerequisite : priority_queue.

The idea is to put highest frequency character first (a greedy approach). We use a priority queue (Or Binary Max Heap) and put all characters and ordered by their frequencies (highest

frequency character at root). We one by one take highest frequency character from the heap and add it to result. After we add, we decrease frequency of the character and we temporarily move this character out of priority queue so that it is not picked next time.

We have to follow the step to solve this problem, they are:
1. Build a Priority_queue or max_heap, **pq** that stores characters and their frequencies.
..... Priority_queue or max_heap is built on the bases of frequency of character.
2. Create a temporary Key that will used as the previous visited element (previous element in resultant string. Initialize it { char = '#' , freq = '-1' }
3. While **pq** is not empty.
..... Pop an element and add it to result.
..... Decrease frequency of the popped element by '1'
..... Push the previous element back into the priority_queue if it's frequency > '0'
..... Make the current element as previous element for the next iteration.
4. If length of resultant string and original, print "not possible". Else print result.

Below c++ implementation of above idea

```
 // C++ program to rearrange characters in a string
// so that no two adjacent characters are same.
#include<bits/stdc++.h>
using namespace std;

const int MAX_CHAR = 26;

struct Key
{
    int freq; // store frequency of character
    char ch;

    // function for priority_queue to store Key
    // according to freq
    bool operator<(const Key &k) const
    {
        return freq < k.freq;
    }
};

// Function to rearrange character of a string
// so that no char repeat twice
void rearrangeString(string str)
{
    int n = str.length();

    // Store frequencies of all characters in string
    int count[MAX_CHAR] = {0};
    for (int i = 0 ; i < n ; i++)
        count[str[i]-'a']++;
```

```
    // Insert all characters with their frequencies
    // into a priority_queue
    priority_queue< Key > pq;
    for (char c = 'a' ; c <= 'z' ; c++)
        if (count[c-'a'])
            pq.push( Key { count[c-'a'], c} );

    // 'str' that will store resultant value
    str = "" ;

    // work as the previous visited element
    // initial previous element be. ( '#' and
    // it's frequency '-1' )
    Key prev {-1, '#'} ;

    // traverse queue
    while (!pq.empty())
    {
        // pop top element from queue and add it
        // to string.
        Key k = pq.top();
        pq.pop();
        str = str + k.ch;

        // IF frequency of previous character is less
        // than zero that means it is useless, we
        // need not to push it
        if (prev.freq > 0)
            pq.push(prev);

        // make current character as the previous 'char'
        // decrease frequency by 'one'
        (k.freq)--;
        prev = k;
    }

    // If length of the resultant string and original
    // string is not same then string is not valid
    if (n != str.length())
        cout << " Not valid String " << endl;

    else // valid string
        cout << str << endl;
}

// Driver program to test above function
int main()
{
```

```
    string str = "bbbaa" ;
    rearrangeString(str);
    return 0;
}
```

Output:

```
babab
```

Time complexity : O(n log n)

Source

https://www.geeksforgeeks.org/rearrange-characters-string-no-two-adjacent/

Chapter 63

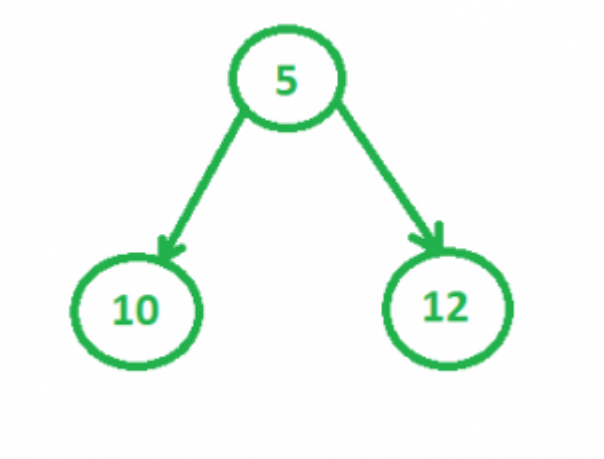
Skew Heap

Skew Heap - GeeksforGeeks

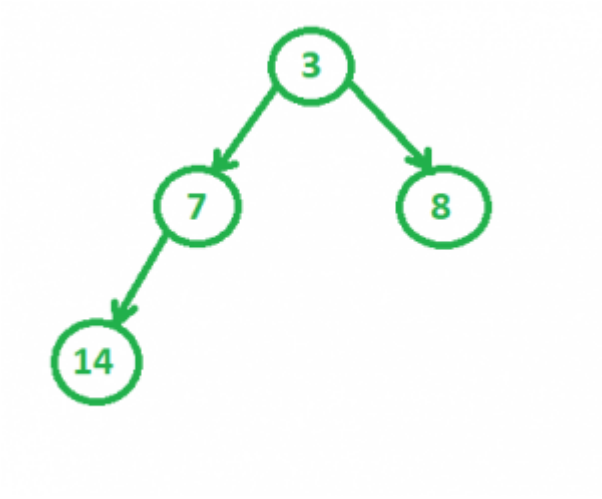
A **skew heap** (or self – adjusting heap) is a heap data structure implemented as a **binary tree**. Skew heaps are advantageous because of their ability to **merge more quickly** than binary heaps. In contrast with binary heaps, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied :

1. The general heap order must be there (root is minimum and same is recursively true for subtrees), but balanced property (all levels must be full except the last) is not required.
2. Main operation in Skew Heaps is Merge. We can implement other operations like insert, extractMin(), etc using Merge only.

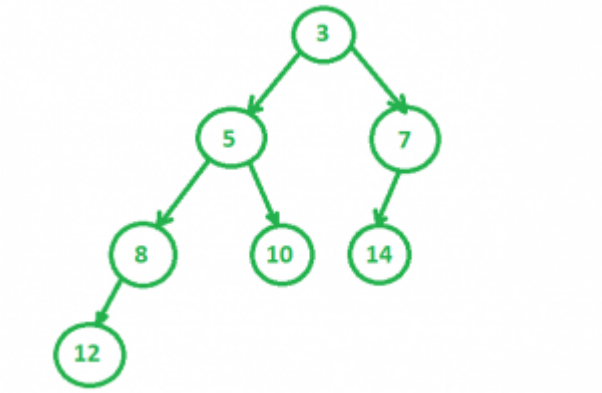
Example :
1. Consider the skew heap 1 to be



2. The second heap to be considered



4. And we obtain the final merged tree as



Recursive Merge Process :
merge(h1, h2)

1. Let h1 and h2 be the two min skew heaps to be merged. Let h1's root be smaller than h2's root (If not smaller, we can swap to get the same).
2. We swap h1->left and h1->right.
3. h1->left = merge(h2, h1->left)

Examples :

```
Let h1 be
        10
     /    \
   20      30
```

```
  /        /
40        50

Let h2 be
       15
     /    \
   25      35
  /  \
45    55

After swapping h1->left and h1->right, we get
        10
     /    \
   30      20
  /        /
50        40

Now we recursively Merge
   30
   /     AND
  40

       15
     /    \
   25      35
  /  \
45    55
After recursive merge, we get (Please do it
using pen and paper).
       15
     /    \
   30      25
  /  \       \
35    40      45

We make this merged tree as left of original
h1 and we get following result.
           10
         /      \
       15        20
    /    \      /
   30    25    40
 /   \     \
35   40    45
```

For visualization : https://www.cs.usfca.edu/~galles/JavascriptVisual/LeftistHeap.html

```
// CPP program to implement Skew Heap
```

```
// operations.
#include <bits/stdc++.h>
using namespace std;

struct SkewHeap
{
    int key;
    SkewHeap* right;
    SkewHeap* left;

    // constructor to make a new
    // node of heap
    SkewHeap()
    {
        key = 0;
        right = NULL;
        left = NULL;
    }

    // the special merge function that's
    // used in most of the other operations
    // also
    SkewHeap* merge(SkewHeap* h1, SkewHeap* h2)
    {
        // If one of the heaps is empty
        if (h1 == NULL)
            return h2;
        if (h2 == NULL)
            return h1;

        // Make sure that h1 has smaller
        // key.
        if (h1->key > h2->key)
           swap(h1, h2);

        // Swap h1->left and h1->right
        swap(h1->left, h1->right);

        // Merge h2 and h1->left and make
        // merged tree as left of h1.
        h1->left = merge(h2, h1->left);

        return h1;
    }

    // function to construct heap using
    // values in the array
    SkewHeap* construct(SkewHeap* root,
```

```
                    int heap[], int n)
    {
        SkewHeap* temp;
        for (int i = 0; i < n; i++) {
            temp = new SkewHeap;
            temp->key = heap[i];
            root = merge(root, temp);
        }
        return root;
    }

    // fucntion to print the Skew Heap,
    // as it is in form of a tree so we use
    // tree traversal algorithms
    void inorder(SkewHeap* root)
    {
        if (root == NULL)
            return;
        else {
            inorder(root->left);
            cout << root->key << "  ";
            inorder(root->right);
        }
        return;
    }
};

// Driver Code
int main()
{
    // Construct two heaps
    SkewHeap heap, *temp1 = NULL,
                   *temp2 = NULL;
    /*
            5
           / \
          /   \
         10   12    */
    int heap1[] = { 12, 5, 10 };
    /*
            3
           / \
          /   \
         7     8
        /
       /
      14    */
    int heap2[] = { 3, 7, 8, 14 };
```

```
    int n1 = sizeof(heap1) / sizeof(heap1[0]);
    int n2 = sizeof(heap2) / sizeof(heap2[0]);
    temp1 = heap.construct(temp1, heap1, n1);
    temp2 = heap.construct(temp2, heap2, n2);

    // Merge two heaps
    temp1 = heap.merge(temp1, temp2);
    /*
            3
           / \
          /   \
         5     7
        / \   /
       8  10 14
      /
     12    */
    cout << "Merged Heap is: " << endl;
    heap.inorder(temp1);
}
```

Output:

```
The heap obtained after merging is:
12  8  5  10  3  14  7
```

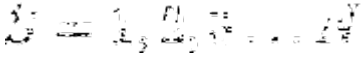
Source

https://www.geeksforgeeks.org/skew-heap/

Chapter 64

Smallest Derangement of Sequence

Smallest Derangement of Sequence - GeeksforGeeks

Given the sequence  find the lexicographically smallest (earliest in dictionary order) derangement of  .

A derangement of S is as any permutation of S such that no two elements in S and its permutation occur at same position.

Examples:

```
Input: 3
Output : 2 3 1
Explanation: The Sequence is 1 2 3.
Possible permutations are (1, 2, 3), (1, 3, 2),
          (2, 1, 3), (2, 3, 1), (3, 1, 2) (3, 2, 1).
Derangements are (2, 3, 1), (3, 1, 2).
Smallest Derangement: (2, 3, 1)

Input : 5
Output : 2 1 4 5 3.
Explanation: Out of all the permutations of
(1, 2, 3, 4, 5), (2, 1, 4, 5, 3) is the first derangement.
```

Method 1:
We can modify the method shown in this article: Largest Derangement
Using a min heap we can successively get the least element and place them in more significant positions, taking care that the property of derangement is maintained.
Complexity: O(N * log N)

Below is the C++ implementation.

```
 // CPP program to generate smallest derangement
// using priority queue.
#include <bits/stdc++.h>
using namespace std;

void generate_derangement(int N)
{
    // Generate Sequence and insert into a
    // priority queue.
    int S[N + 1];
    priority_queue<int, vector<int>, greater<int> > PQ;
    for (int i = 1; i <= N; i++) {
        S[i] = i;
        PQ.push(S[i]);
    }

    // Generate Least Derangement
    int D[N + 1];
    for (int i = 1; i <= N; i++) {
        int d = PQ.top();
        PQ.pop();
        if (d != S[i] || i == N) {
            D[i] = d;
        }
        else {
            D[i] = PQ.top();
            PQ.pop();
            PQ.push(d);
        }
    }

    if (D[N] == S[N])
       swap(S[N], D[N]);

    // Print Derangement
    for (int i = 1; i <= N; i++)
        printf("%d ", D[i]);
    printf("\n");
}

// Driver code
int main()
{
    generate_derangement(10);
    return 0;
}
```

Output:

```
2 1 4 3 6 5 8 7 10 9
```

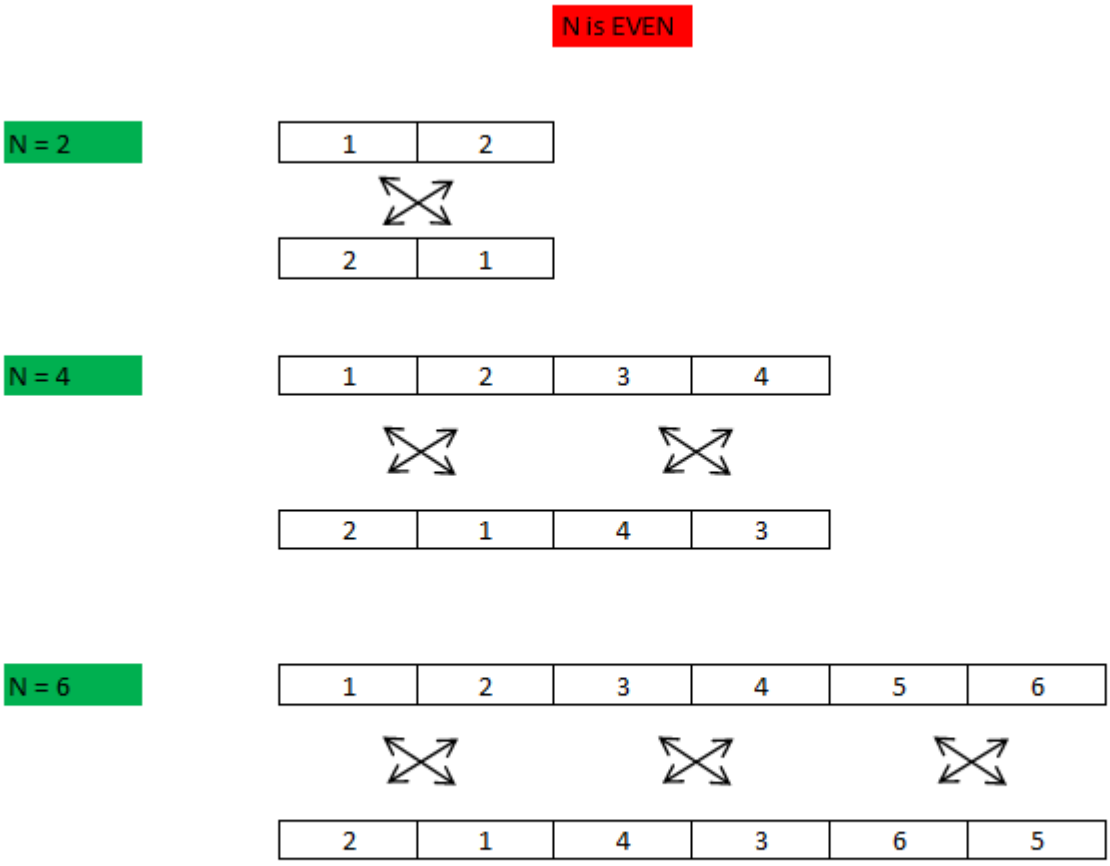
Method 2

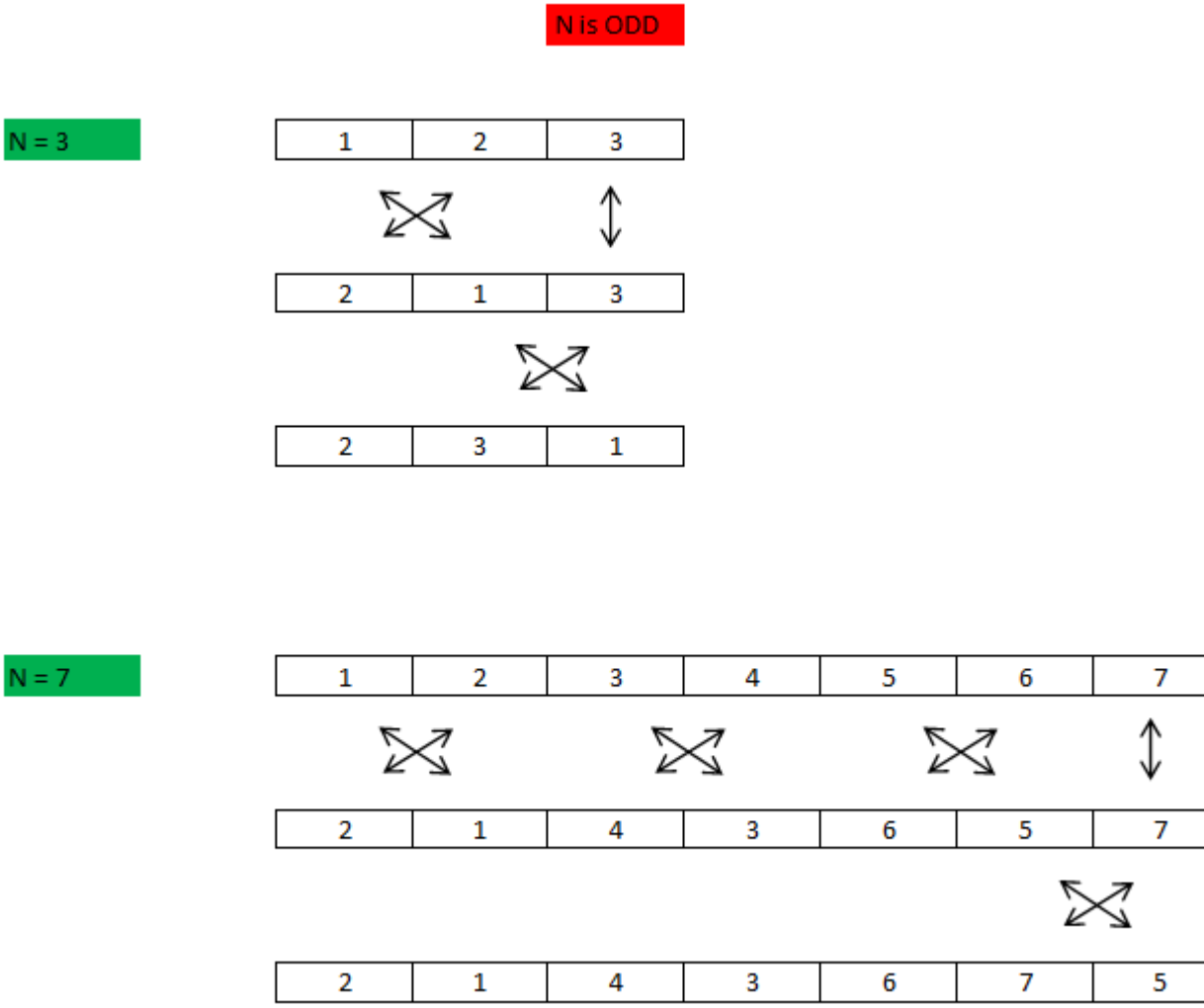
Since we are given a very specific sequence i.e [illegible] we can calculate the answer even more efficiently.

Divide the original sequence into pairs of two elements, and then swap the elements of each pair.

If N is odd then the last pair needs to be swapped again.

Pictorial Representation





Complexity: We perform at most N/2 + 1 swaps, so the complexity is O(N).

Why does this method work

This method is a very specific application of method 1 and is based on observation. Given the nature of sequence, at position i we already know the least element that can be put, which is either i+1 or i-1. Since we are already given the least permutation of S it is clear that the derangement must start from 2 and not 1 ie of the form i+1 (i = 1). The next element will be of the form i – 1 . The element after this will be i + 1 and then next i – 1. This pattern will continue until the end.

This operation is most easily understood as the swapping of adjacent elements of pairs of elements of S.

If we can determine the least element in constant time, then the complexity overhead from heap is eliminated. Hence from O(N * log N) the complexity reduces to O(N).

Below is the C++ implementation

```
 // Efficient C++ program to find smallest
// derangement.
#include <stdio.h>

void generate_derangement(int N)
{
    // Generate Sequence S
    int S[N + 1];
    for (int i = 1; i <= N; i++)
        S[i] = i;

    // Generate Derangement
    int D[N + 1];
    for (int i = 1; i <= N; i += 2) {
        if (i == N) {

            // Only if i is odd
            // Swap S[N-1] and S[N]
            D[N] = S[N - 1];
            D[N - 1] = S[N];
        }
        else {
            D[i] = i + 1;
            D[i + 1] = i;
        }
    }

    // Print Derangement
    for (int i = 1; i <= N; i++)
        printf("%d ", D[i]);
    printf("\n");
}

// Driver Program
int main()
{
    generate_derangement(10);
    return 0;
}
```

Output:

```
2 1 4 3 6 5 8 7 10 9
```

Note : The auxiliary space can be reduced to O(1) if we perform the swapping operations on S itself.

Source

https://www.geeksforgeeks.org/smallest-derangement-sequence/

Chapter 65

Sort a nearly sorted (or K sorted) array

Sort a nearly sorted (or K sorted) array - GeeksforGeeks

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.

```
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)

C++

```
 // 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 arrayn";
    printArray(arr, n);

    return 0;
}
```

C

```
 #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 heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int 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 remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// 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)
{
    // Create a Min Heap of first (k+1) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i<=k && i<n; i++) // i < n condition is needed when k > n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);

    // i is index for remaining elements in arr[] and ti
    // is target index of for cuurent minimum element in
    // Min Heapm 'hp'.
    for(int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[i]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }
}

// 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(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()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    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;
}

// 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 arrayn";
    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 insertand deleteoperations 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.

Source

https://www.geeksforgeeks.org/nearly-sorted-algorithm/

Chapter 66

Sort a nearly sorted array using STL

Sort a nearly sorted array using STL - GeeksforGeeks

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. It may be assumed that k < n.

```
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}
```

A **simple solution** is to sort the array using any standard sorting algorithm. The time complexity of this solution is O(n Log n)

A better solution is to use priority queue (or heap data structure).

```
1) Build a priority queue pq of first (k+1) elements.
2) Initialize index = 0 (For result array).
3) Do following for elements from k+1 to n-1.
    a) Pop an item from pq and put it at index, increment index.
    b) Push arr[i] to pq.
4) While pq is not empty
      Pop an item from pq and put it at index, increment index.
```

We have discussed a simple implementation in Sort a nearly sorted (or K sorted) array. In this post, an STL based implementation is done.

```
 // 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)
    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 arrayn";
    printArray(arr, n);
```

```
    return 0;
}
```

Output:

```
Following is sorted arrayn2 3 6 8 12 56
```

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

Source

https://www.geeksforgeeks.org/sort-a-nearly-sorted-array-using-stl/

Chapter 67

Sort numbers stored on different machines

Sort numbers stored on different machines - GeeksforGeeks

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

```
Example:
       Machine M1 contains 3 numbers: {30, 40, 50}
       Machine M2 contains 2 numbers: {35, 45}
       Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

       Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}
```

Representation of stream of numbers on each machine is considered as linked list. A Min Heap can be used to print all numbers in sorted order.

Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is number of machines.

2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.

3. Repeat the above step 2 until heap is not empty.

Below is C++ implementation of the above approach.

```
// A program to take numbers from different machines and print them in sorted order
```

```
#include <stdio.h>

// A Linked List node
struct ListNode
{
    int data;
    struct ListNode* next;
};

// A Min Heap Node
struct MinHeapNode
{
    ListNode* head;
};

// A Min Heao (Collection of Min Heap nodes)
struct MinHeap
{
    int count;
    int capacity;
    MinHeapNode* array;
};

// A function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
    minHeap->capacity = capacity;
    minHeap->count = 0;
    minHeap->array = new MinHeapNode [minHeap->capacity];
    return minHeap;
}

/* A utility function to insert a new node at the begining
   of linked list */
void push (ListNode** head_ref, int new_data)
{
    /* allocate node */
    ListNode* new_node = new ListNode;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
```

```
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swap( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
         minHeap->array[left].head->data <
         minHeap->array[smallest].head->data
       )
        smallest = left;

    if ( right < minHeap->count &&
         minHeap->array[right].head->data <
         minHeap->array[smallest].head->data
       )
        smallest = right;

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

// A utility function to check whether a Min Heap is empty or not
int isEmpty( MinHeap* minHeap )
{
    return (minHeap->count == 0);
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int i, n;
```

```
    n = minHeap->count  - 1;
    for( i = (n - 1) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// This function inserts array elements to heap and then calls
// buildHeap for heap property among nodes
void populateMinHeap( MinHeap* minHeap, ListNode* *array, int n )
{
    for( int i = 0; i < n; ++i )
        minHeap->array[ minHeap->count++ ].head = array[i];

    buildMinHeap( minHeap );
}

// Return minimum element from all linked lists
ListNode* extractMin( MinHeap* minHeap )
{
    if( isEmpty( minHeap ) )
         return NULL;

    // The root of heap will have minimum value
    MinHeapNode temp = minHeap->array[0];

    // Replace root either with next node of the same list.
    if( temp.head->next )
        minHeap->array[0].head = temp.head->next;
    else // If list empty, then reduce heap size
    {
        minHeap->array[0] = minHeap->array[ minHeap->count - 1 ];
        --minHeap->count;
    }

    minHeapify( minHeap, 0 );
    return temp.head;
}

// The main function that takes an array of lists from N machines
// and generates the sorted output
void externalSort( ListNode *array[], int N )
{
    // Create a min heap of size equal to number of machines
    MinHeap* minHeap = createMinHeap( N );

    // populate first item from all machines
    populateMinHeap( minHeap, array, N );

    while ( !isEmpty( minHeap ) )
```

```
    {
        ListNode* temp = extractMin( minHeap );
        printf( "%d ",temp->data );
    }
}

// Driver program to test above functions
int main()
{
    int N = 3; // Number of machines

    // an array of pointers storing the head nodes of the linked lists
    ListNode *array[N];

    // Create a Linked List 30->40->50 for first machine
    array[0] = NULL;
    push (&array[0], 50);
    push (&array[0], 40);
    push (&array[0], 30);

    // Create a Linked List 35->45 for second machine
    array[1] = NULL;
    push (&array[1], 45);
    push (&array[1], 35);

    // Create Linked List 10->60->70->80 for third machine
    array[2] = NULL;
    push (&array[2], 100);
    push (&array[2], 80);
    push (&array[2], 70);
    push (&array[2], 60);
    push (&array[2], 10);

    // Sort all elements
    externalSort( array, N );

    return 0;
}
```

Output:

```
10 30 35 40 45 50 60 70 80 100
```

Source

https://www.geeksforgeeks.org/sort-numbers-stored-on-different-machines/

Chapter 68

Sum of all elements between k1'th and k2'th smallest elements

Sum of all elements between k1'th and k2'th smallest elements - GeeksforGeeks

Given an array of integers and two numbers k1 and k2. Find sum of all elements between given two k1'th and k2'th smallest elements of array. It may be assumed that (1 <= k1 < k2 <= n) and all elements of array are distinct.

Examples :

```
Input : arr[] = {20, 8, 22, 4, 12, 10, 14},  k1 = 3,  k2 = 6
Output : 26
         3rd smallest element is 10. 6th smallest element
         is 20. Sum of all element between k1 & k2 is
         12 + 14 = 26

Input : arr[] = {10, 2, 50, 12, 48, 13}, k1 = 2, k2 = 6
Output : 73
```

Method 1 (Sorting)
First sort the given array using a O(n log n) sorting algorithm like Merge Sort, Heap Sort, etc and return the sum of all element between index k1 and k2 in the sorted array.

Below is the implementation of the above idea :

C++

```
 // C++ program to find sum of all element between
// to K1'th and k2'th smallest elements in array
```

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

using namespace std;

// Returns sum between two kth smallest element of array
int sumBetweenTwoKth(int arr[], int n, int k1, int k2)
{
    // Sort the given array
    sort(arr, arr + n);

    /* Below code is equivalent to
     int result = 0;
     for (int i=k1; i<k2-1; i++)
      result += arr[i]; */
    return accumulate(arr + k1, arr + k2 - 1, 0);
}

// Driver program
int main()
{
    int arr[] = { 20, 8, 22, 4, 12, 10, 14 };
    int k1 = 3, k2 = 6;
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << sumBetweenTwoKth(arr, n, k1, k2);
    return 0;
}
```

Java

```
 // Java program to find sum of all element
// between to K1'th and k2'th smallest
// elements in array
import java.util.Arrays;

class GFG {

    // Returns sum between two kth smallest
    // element of array
    static int sumBetweenTwoKth(int arr[],
                                int k1, int k2)
    {
        // Sort the given array
        Arrays.sort(arr);

        // Below code is equivalent to
        int result = 0;

        for (int i = k1; i < k2 - 1; i++)
```

```
            result += arr[i];

        return result;
    }

    // Driver code
    public static void main(String[] args)
    {

        int arr[] = { 20, 8, 22, 4, 12, 10, 14 };
        int k1 = 3, k2 = 6;
        int n = arr.length;

        System.out.print(sumBetweenTwoKth(arr,
                                          k1, k2));
    }
}

// This code is contributed by Anant Agarwal.
```

Python3

```
 # Python program to find sum of
# all element between to K1'th and
# k2'th smallest elements in array

# Returns sum between two kth
# smallest element of array
def sumBetweenTwoKth(arr, n, k1, k2):

    # Sort the given array
    arr.sort()

    result = 0
    for i in range(k1, k2-1):
        result += arr[i]
    return result

# Driver code
arr = [ 20, 8, 22, 4, 12, 10, 14 ]
k1 = 3; k2 = 6
n = len(arr)
print(sumBetweenTwoKth(arr, n, k1, k2))


# This code is contributed by Anant Agarwal.
```

C#

```
 // C# program to find sum of all element
// between to K1'th and k2'th smallest
// elements in array
using System;

class GFG {

    // Returns sum between two kth smallest
    // element of array
    static int sumBetweenTwoKth(int[] arr, int n,
                                int k1, int k2)
    {
        // Sort the given array
        Array.Sort(arr);

        // Below code is equivalent to
        int result = 0;

        for (int i = k1; i < k2 - 1; i++)
            result += arr[i];

        return result;
    }

    // Driver code
    public static void Main()
    {
        int[] arr = {20, 8, 22, 4, 12, 10, 14};
        int k1 = 3, k2 = 6;
        int n = arr.Length;

        Console.Write(sumBetweenTwoKth(arr, n, k1, k2));
    }
}

// This code is contributed by nitin mittal.
```

Output:

```
 26
```

Time Complexity: O(n log n)

Method 2 (Using Min Heap)
We can optimize above solution be using a min heap.
1) Create a min heap of all array elements. (This step takes O(n) time)
2) Do extract minimum k1 times (This step takes O(K1 Log n) time)

3) Do extract minimum k2 – k1 – 1 times and sum all extracted elements. (This step takes O ((K2 – k1) * Log n) time)

Overall time complexity of this method is O(n + k2 Log n) which is better than sorting based method.

Improved By : nitin mittal, AbhishekVats3

Source

https://www.geeksforgeeks.org/sum-elements-k1th-k2th-smallest-elements/

Chapter 69

Time Complexity of building a heap

Time Complexity of building a heap - GeeksforGeeks

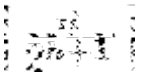
Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1
        do HEAPIFY(A, i);
    end for
END
```

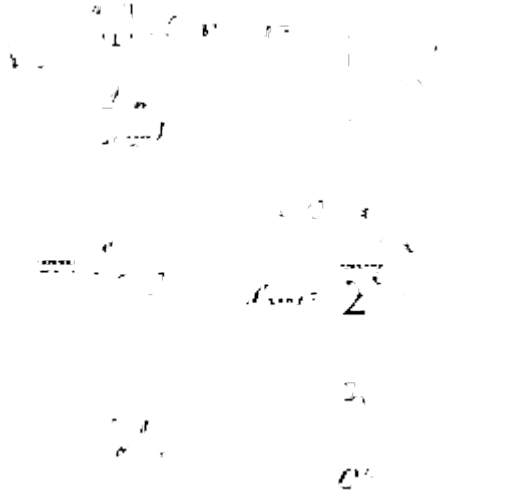
A quick look over the above algorithm suggests that the running time is $O(nlg(n))$, since each call to **Heapify** costs $O(lg(n))$ and **Build-Heap** makes $O(n)$ such calls.
This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the running time of **Heapify** depends on the height of the tree 'h' (which is equal to lg(n), where n is number of nodes) and the heights of most sub-trees are small.
The height 'h' increases as we move upwards along the tree. Line-3 of **Build-Heap** runs a loop from the index of the last internal node (heapsize/2) with height=1, to the index of root(1) with height = lg(n). Hence, **Heapify** takes different time for each node, which is $O(h)$

For finding the Time Complexity of building a heap, we must know the number of nodes having height h.

For this we use the fact that, A heap of size n has at most  nodes with height h.

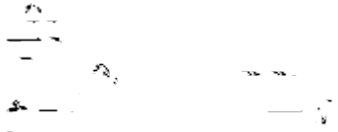
Now to derive the time complexity, we express the total cost of **Build-Heap** as-



(1)

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2(). Similarly in Step three, the upper limit of the summation can be increased to infinity since we are using Big-Oh notation.

Sum of infinite G.P. (x < 1)

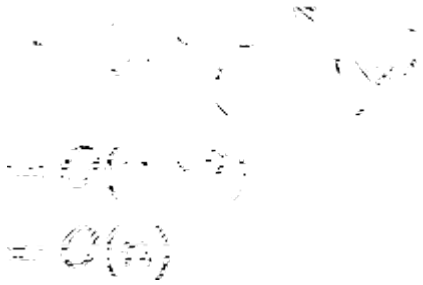


(2)

On differentiating both sides and multiplying by x, we get

(3)

Putting the result obtained in (3) back in our derivation (1), we get



(4)

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

Reference :
http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf

Source

https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/

Chapter 70

Tournament Tree (Winner Tree) and Binary Heap

Tournament Tree (Winner Tree) and Binary Heap - GeeksforGeeks

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.

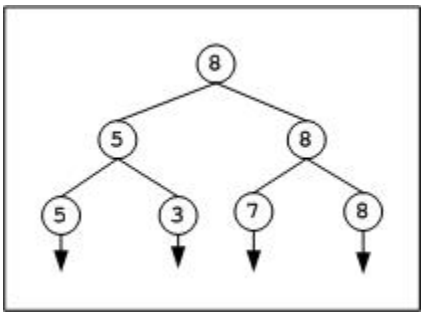
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in equation given in the post).

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 $\mathbf{(N + \log_2 N - 2)}$ 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.

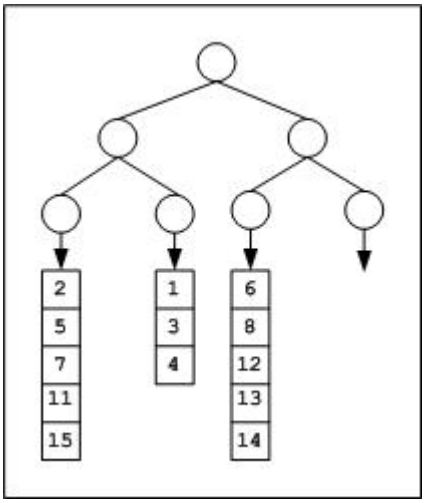
Median of Sorted Arrays

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.

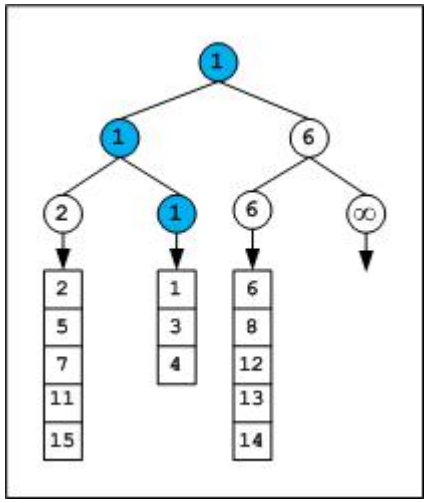
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
{ 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.



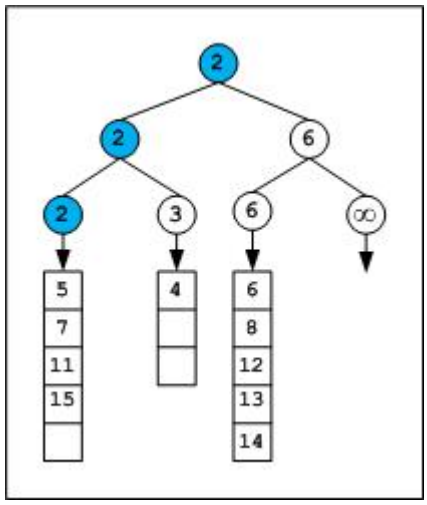
After the first tournament, the tree appears as below,



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((L₁ + L₂ + … + Lₘ) * logM)*** to merge all the arrays, and ***O(m*logM)*** 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

Second minimum element using minimum comparisons

Related Posts
Find the smallest and second smallest element in an array.
Second minimum element using minimum comparisons

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

https://www.geeksforgeeks.org/tournament-tree-and-binary-heap/

Chapter 71

Where is Heap Sort used practically?

Where is Heap Sort used practically? - GeeksforGeeks

Although QuickSortworks better in practice, the advantage of HeapSortworst case upper bound of O(nLogn).

MergeSortalso has upper bound as O(nLogn) and works better in practice when compared to HeapSort. But MergeSort requires O(n) extra space

HeapSort is not used much in practice, but can be useful in real time (or time bound where QuickSort doesn't fit) embedded systems where less space is available (MergeSort doesn't fit)

Source

https://www.geeksforgeeks.org/where-is-heap-sort-used-practically/

Chapter 72

Why is Binary Heap Preferred over BST for Priority Queue?

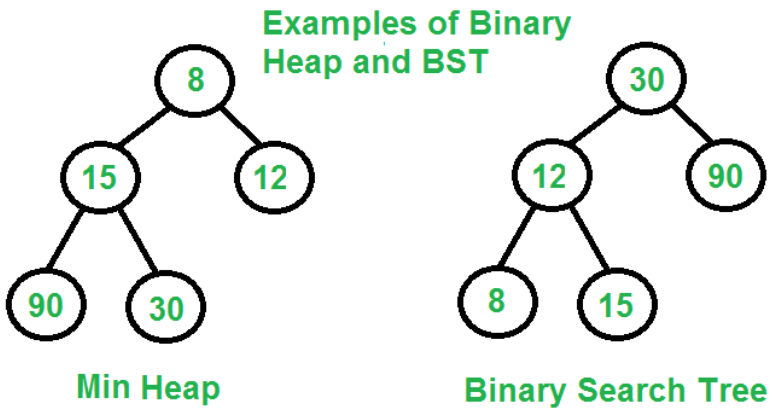
Why is Binary Heap Preferred over BST for Priority Queue? - GeeksforGeeks

A typical Priority Queue requires following operations to be efficient.

1. Get Top Priority Element (Get minimum or maximum)
2. Insert an element
3. Remove top priority element
4. Decrease Key

A Binary Heapsupports above operations with following time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)



A Self Balancing Binary Search Tree like AVL Tree, Red-Black Tree, etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally O(1), but can be easily implemented in O(1) by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.
2. Inserting an element is naturally O(Logn)
3. Removing maximum or minimum are also O(Logn)
4. Decrease key can be done in O(Logn) by doing a deletion followed by insertion. See this for details.

So why is Binary Heap Preferred for Priority Queue?

- Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
- Although operations are of same time complexity, constants in Binary Search Tree are higher.
- We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.
- Binary Heap doesn't require extra space for pointers.
- Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in $\Theta(1)$ time

Is Binary Heap always better?
Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

- Searching an element in self-balancing BST is O(Logn) which is O(n) in Binary Heap.
- We can print all elements of BST in sorted order in O(n) time, but Binary Heap requires O(nLogn) time.
- Floor and ceil can be found in O(Logn) time.
- K'th largest/smallest elementbe found in O(Logn) time by augmenting tree with an additional field.

This article is contributed by **Vivek Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/why-is-binary-heap-preferred-over-bst-for-priority-queue/

Chapter 73

heapq in Python to print all elements in sorted order from row and column wise sorted matrix

heapq in Python to print all elements in sorted order from row and column wise sorted matrix - GeeksforGeeks

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.

Examples:

```
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]
```

This problem has existing solution please refer link. We will solve this problem in python with the same approach of merging two sorted arrays using **heapq module**.

```
 # Function to print all elements in sorted order
# from row and column wise sorted matrix
from heapq import merge
```

```
def sortedMatrix(mat):

    # initialize result variable with first row of matrix
    result=mat[0]

    # now traverse through complete matrix
    # after first row and merge each row with
    # result one by one
    # after last operation result will contain
    # list of sorted elements of matrix
    for row in mat[1:]:
        result=list(merge(result,row))

    return result

if __name__ == "__main__":
    mat = [[10, 20, 30, 40],[15, 25, 35, 45], \
          [27, 29, 37, 48],[32, 33, 39, 50]]
    print 'Elements of matrix in sorted order'
    print sortedMatrix(mat)
```

Output:

```
Elements of matrix in sorted order
[10, 15, 20, 25, 27, 29, 30, 32, 33, 35, 37, 39, 40, 45, 48, 50]
```

Source

https://www.geeksforgeeks.org/heapq-python-print-elements-sorted-order-row-column-wise-sorted-matrix/

Chapter 74

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

k largest(or smallest) elements in an array | added Min Heap method - GeeksforGeeks

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) For each element *x* in arr[k] to arr[n-1]
If *x* is greater than the min then remove *min* from temp[] and insert *x*.
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++

```
 // 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
```

Java

```
 // Java code for k largest elements in an array
import java.util.Arrays;
import java.util.Collections;

class GFG
{
    public static void kLargest(Integer [] arr, int k)
    {
        // Sort the given array arr in reverse order
        // This method doesn't work with primitive data
        // types. So, instead of int, Integer type
        // array will be used
```

```
        Arrays.sort(arr, Collections.reverseOrder());


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


    public static void main(String[] args)
    {
        Integer arr[] = new Integer[]{1, 23, 12, 9,
                                      30, 2, 50};
        int k = 3;
        kLargest(arr,k);
    }
}
// This code is contributed by Kamal Rawal
```

Python

```
 ''' Python3 code for k largest elements in an array'''

def kLargest(arr, k):
    # Sort the given array arr in reverse
    # order.
    arr.sort(reverse=True)
    #Print the first kth largest elements
    for i in range(k):
        print (arr[i],end=" ")

# Driver program
arr = [1, 23, 12, 9, 30, 2, 50]
#n = len(arr)
k = 3
kLargest(arr, k)

# This code is contributed by shreyanshi_arun.
```

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 heapifyfor 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.

Please write comments if you find any of the above explanations/algorithms incorrect, or find better ways to solve the same problem.

References:
http://en.wikipedia.org/wiki/Selection_algorithm

Source

https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/

Chapter 75

std::make_heap() in C++ STL

std::make_heap() in C++ STL - GeeksforGeeks

make_heap() is used to transform a sequence into a heap. A heap is a data structure which **points to highest**(or lowest) element and making its access in **O(1)** time. Order of all the other elements depends upon particular implementation, but remains consistent throughout. This function is defined in the header "**algorithm**". There are two implementations of make_heap() function. Both of them are explained through this article.

Syntax 1 : make_heap(iter_first, iter_last)

```
Template :
  void make_heap (RandomAccessIterator first, RandomAccessIterator last);
Parameters :
first : The pointer to the starting element of sequence that has to
be transformed into heap.
last : The pointer to the next address to last element of sequence that
has to be transformed into heap.
```

Below is the demonstrating code :

```
 // C++ code to demonstrate the working of
// make_heap() using syntax 1

#include<iostream>
#include<algorithm> // for heap
#include<vector>
using namespace std;

int main()
```

```
{
    // initializing vector;
    vector<int> vi = { 4, 6, 7, 9, 11, 4 };

    // using make_heap() to transform vector into
    // a max heap
    make_heap(vi.begin(),vi.end());

    //checking if heap using
    // front() function
    cout << "The maximum element of heap is : ";
    cout << vi.front() << endl;

}
```

Output:

```
The maximum element of heap is : 11
```

Syntax 2 : make_heap(iter_first, iter_last, comp)

```
Template :
  void make_heap (RandomAccessIterator first, RandomAccessIterator last, comp);
Parameters :
first : The pointer to the starting element of sequence that has to
be transformed into heap.
last : The pointer to the next address to last element of sequence that
has to be transformed into heap.
comp : The comparator function that returns a boolean
 true/false of the each elements compared. This function
accepts two arguments. This can be function pointer or
function object and cannot change values.
```

Below is the demonstrating code :

```
 // C++ code to demonstrate the working of
// make_heap() using syntax 2

#include<iostream>
#include<algorithm> // for heap
#include<vector>
using namespace std;
```

```
// comparator function to make min heap
struct greaters{
  bool operator()(const long& a,const long& b) const{
    return a>b;
  }
};

int main()
{
    // initializing vector;
    vector<int> vi = { 15, 6, 7, 9, 11, 45 };

    // using make_heap() to transform vector into
    // a min heap
    make_heap(vi.begin(),vi.end(), greaters());

    // checking if heap using
    // front() function
    cout << "The minimum element of heap is : ";
    cout << vi.front() << endl;

}
```

Output:

```
The minimum element of heap is : 6
```

Possible application : This function can be used in scheduling. In scheduling, a new element is inserted dynamically in iterations. Sorting again and again to get maximum takes much complexity O(nlogn), instead of that we use "push_heap()" function to heapify the heap made in O(logn) time . The code below depicts its implementation.

```
 // C++ code to demonstrate
// application of make_heap() (max_heap)
// priority scheduling

#include<iostream>
#include<algorithm> // for heap
#include<vector>
using namespace std;

int main()
{
    // initializing vector;
    // initial job priorities
    vector<int> vi = { 15, 6, 7, 9, 11, 19};
```

```
    // No. of incoming jobs.
    int k = 3;

    // using make_heap() to transform vector into
    // a min heap
    make_heap(vi.begin(),vi.end());

    // initializing job variable
    int a = 10;

    for ( int i=0; i<k; i++)
    {

    // push a job with priority level
    vi.push_back(a);

    // transform into heap ( using push_heap() )
    push_heap(vi.begin(), vi.end());

    //checking top priority job
    // front() funtion
    cout << "Job with maximum priority is : ";
    cout << vi.front() << endl;

    // increasing job priority level
    a = a + 10;

    }

}
```

Output:

```
ob with maximum priority is : 19
Job with maximum priority is : 20
Job with maximum priority is : 30
```

Source

https://www.geeksforgeeks.org/stdmake_heap-c-stl/