Practical-7

Name:-Vishwajit kate

Reg. No:-2020BIT011

# Write a code with complete simulation of the following

# AVL tree

# Binary Heap

# Max Heap

# Min Heap

# Heapyfy

# 1) AVL tree

# Code:-

// Reg No:2020BIT011

// AVL tree implementation in C++

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node \*left;

Node \*right;

int height;

};

int max(int a, int b);

// Calculate height

int height(Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

// New node creation

Node \*newNode(int key) {

Node \*node = new Node();

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

// Rotate right

Node \*rightRotate(Node \*y) {

Node \*x = y->left;

Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left),

height(y->right)) +

1;

x->height = max(height(x->left),

height(x->right)) +

1;

return x;

}

// Rotate left

Node \*leftRotate(Node \*x) {

Node \*y = x->right;

Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left),

height(x->right)) +

1;

y->height = max(height(y->left),

height(y->right)) +

1;

return y;

}

// Get the balance factor of each node

int getBalanceFactor(Node \*N) {

if (N == NULL)

return 0;

return height(N->left) -

height(N->right);

}

// Insert a node

Node \*insertNode(Node \*node, int key) {

// Find the correct postion and insert the node

if (node == NULL)

return (newNode(key));

if (key < node->key)

node->left = insertNode(node->left, key);

else if (key > node->key)

node->right = insertNode(node->right, key);

else

return node;

// Update the balance factor of each node and

// balance the tree

node->height = 1 + max(height(node->left),

height(node->right));

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1) {

if (key < node->left->key) {

return rightRotate(node);

} else if (key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

}

if (balanceFactor < -1) {

if (key > node->right->key) {

return leftRotate(node);

} else if (key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

}

return node;

}

// Node with minimum value

Node \*nodeWithMimumValue(Node \*node) {

Node \*current = node;

while (current->left != NULL)

current = current->left;

return current;

}

// Delete a node

Node \*deleteNode(Node \*root, int key) {

// Find the node and delete it

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

if ((root->left == NULL) ||

(root->right == NULL)) {

Node \*temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

Node \*temp = nodeWithMimumValue(root->right);

root->key = temp->key;

root->right = deleteNode(root->right,

temp->key);

}

}

if (root == NULL)

return root;

// Update the balance factor of each node and

// balance the tree

root->height = 1 + max(height(root->left),

height(root->right));

int balanceFactor = getBalanceFactor(root);

if (balanceFactor > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rightRotate(root);

} else {

root->left = leftRotate(root->left);

return rightRotate(root);

}

}

if (balanceFactor < -1) {

if (getBalanceFactor(root->right) <= 0) {

return leftRotate(root);

} else {

root->right = rightRotate(root->right);

return leftRotate(root);

}

}

return root;

}

// Print the tree

void printTree(Node \*root, string indent, bool last) {

if (root != nullptr) {

cout << indent;

if (last) {

cout << "R----";

indent += " ";

} else {

cout << "L----";

indent += "| ";

}

cout << root->key << endl;

printTree(root->left, indent, false);

printTree(root->right, indent, true);

}

}

int main() {

Node \*root = NULL;

root = insertNode(root, 33);

root = insertNode(root, 13);

root = insertNode(root, 53);

root = insertNode(root, 9);

root = insertNode(root, 21);

root = insertNode(root, 61);

root = insertNode(root, 8);

root = insertNode(root, 11);

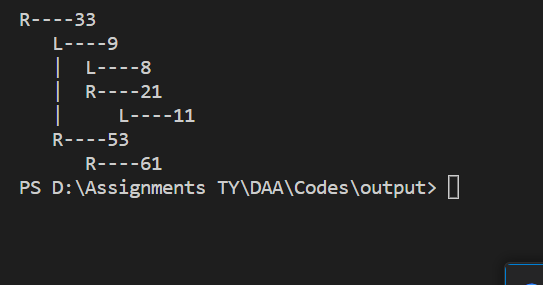
printTree(root, "", true);

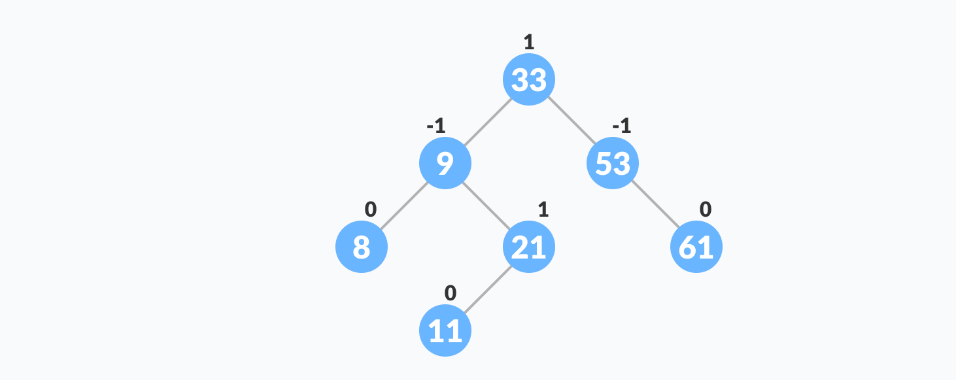
root = deleteNode(root, 13);

cout << "After deleting " << endl;

printTree(root, "", true);

}**Output:**

****

**Simulation:** ****

# 2) Binary Heap

# Code:-

# // Reg No:2020BIT011

// A C++ program to demonstrate common Binary Heap Operations

#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:

// Constructor

MinHeap(int capacity);

// to heapify a subtree with the root at given index

void MinHeapify(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();

// Decreases key value of key at index i to new\_val

void decreaseKey(int i, int new\_val);

// Returns the minimum key (key at root) from min heap

int getMin() { return harr[0]; }

// Deletes a key stored at index i

void deleteKey(int i);

// Inserts a new key 'k'

void insertKey(int k);

};

// 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);

}

}

// Decreases value of key at index 'i' to new\_val. It is assumed that

// new\_val is smaller than harr[i].

void MinHeap::decreaseKey(int i, int new\_val)

{

harr[i] = new\_val;

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--;

MinHeapify(0);

return root;

}

// This function deletes key at index i. It first reduced value to minus

// infinite, then calls extractMin()

void MinHeap::deleteKey(int i)

{

decreaseKey(i, INT\_MIN);

extractMin();

}

// A recursive method to heapify a subtree with the 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;

}

// Driver program to test above functions

int main()

{

MinHeap h(11);

h.insertKey(3);

h.insertKey(2);

h.deleteKey(1);

h.insertKey(15);

h.insertKey(5);

h.insertKey(4);

h.insertKey(45);

cout << h.extractMin() << " ";

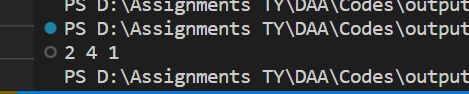
cout << h.getMin() << " ";

h.decreaseKey(2, 1);

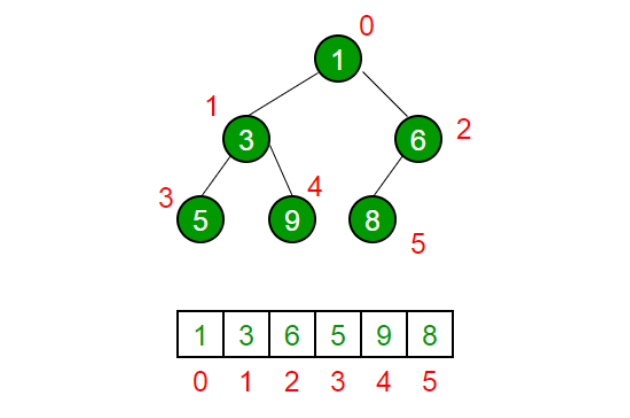
cout << h.getMin();

return 0;

}}**Output:**

****

**Simulation:**

****

# 3) Max Heap

# Code:-

// Reg No:2020BIT011

#include <iostream>

using namespace std;

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

// Find largest among root, left child and right child

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

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

largest = right;

// Swap and continue heapifying if root is not largest

if (largest != i) {

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

heapify(arr, n, largest);

}

}

// main function to do heap sort

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

// Build max heap

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// Heap sort

for (int i = n - 1; i >= 0; i--) {

swap(arr[0], arr[i]);

// Heapify root element to get highest element at root again

heapify(arr, i, 0);

}

}

// Print an array

void printArray(int arr[], int n) {

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

cout << arr[i] << " ";

cout << "\n";

}

// Driver code

int main() {

int arr[] = {1, 12, 9, 5, 6, 10};

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

heapSort(arr, n);

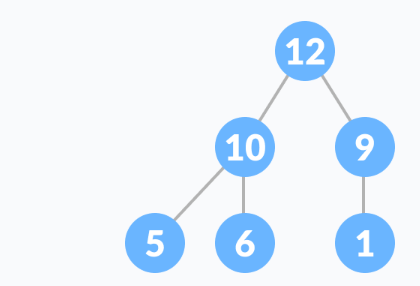
cout << "Sorted array is \n";

printArray(arr, n);

} **Output:-**

****

**Simulation:**

****

# 4) Min Heap

# Code:-

// Reg No:2020BIT011

#include <bits/stdc++.h>

using namespace std;

// Driver code

int main ()

{

// Creates a max heap

priority\_queue <int> pq;

pq.push(5);

pq.push(1);

pq.push(10);

pq.push(30);

pq.push(20);

// One by one extract items from max heap

while (pq.empty() == false)

{

cout << pq.top() << " ";

pq.pop();

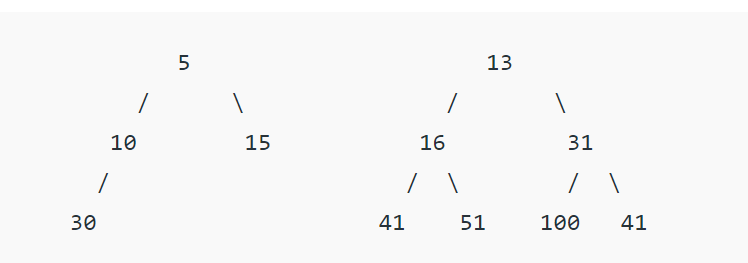
}

return 0;

} **Output:-**

****

**Simulation:**

****

# 5)Heapyfy

# Code:-

// Reg No:2020BIT011

#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)

{

// Initialize largest as root

int largest = i;

// left = 2\*i + 1

int l = 2 \* i + 1;

// right = 2\*i + 2

int r = 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's code

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

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

// Function call

heapSort(arr, N);

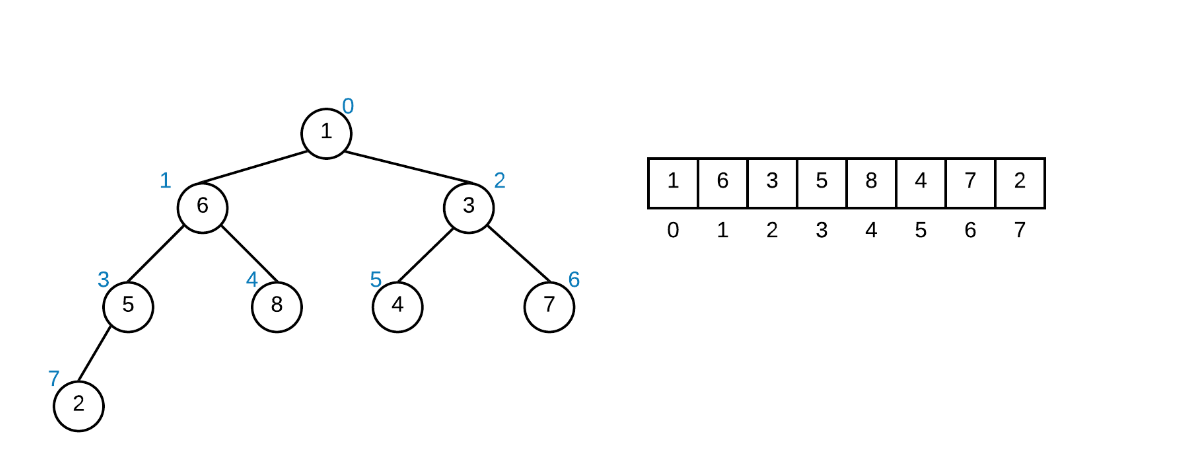
cout << "Sorted array is \n";

printArray(arr, N);

} **Output:-**

****

**Simulation:**

****