**Name: Raj Pagui**

**Roll no: 239**

**Class: Sy MCA**

**1.Wrie c++ program to implement mid-square digit hashing function**

#include <iostream>

#include <vector>

#include <cmath>

#include <string>

using namespace std;

// Function to calculate the hash value using mid-square hashing

int midSquareHashing(int key, int tableSize) {

// Step 1: Square the key

long long square = static\_cast<long long>(key) \* key;

// Step 2: Convert the squared value to string to extract the middle digits

string squareStr = to\_string(square);

// Step 3: Determine the middle digits

int len = squareStr.length();

int mid = len / 2;

// Extract 2 or 3 middle digits based on the length

string midDigits = len % 2 == 0

? squareStr.substr(mid - 1, 2)

: squareStr.substr(mid, 2);

// Step 4: Convert the middle digits back to an integer

int hashValue = stoi(midDigits);

// Step 5: Use modulo operation to fit into the hash table

return hashValue % tableSize;

}

int main() {

// Example keys

vector<int> keys = {123, 456, 789, 101, 202};

// Define the hash table size

int tableSize = 10;

// Display the hash values

cout << "Key\tSquared\t\tHash Value" << endl;

cout << "---------------------------------" << endl;

for (int key : keys) {

long long square = static\_cast<long long>(key) \* key;

int hashValue = midSquareHashing(key, tableSize);

cout << key << "\t" << square << "\t" << hashValue << endl;

}

return 0;

}

**2.Write c++ program to implement hashing function (k mod 7)**

#include <iostream>

#include <vector>

using namespace std;

// Function to calculate hash value using k mod 7

int hashFunction(int key) {

return key % 7;

}

int main() {

// Example keys to hash

vector<int> keys = {15, 28, 35, 49, 63, 77, 91};

// Display the hash values

cout << "Key\tHash Value" << endl;

cout << "----------------" << endl;

for (int key : keys) {

int hashValue = hashFunction(key);

cout << key << "\t" << hashValue << endl;

}

return 0;

}

**3.Write C++ program to implement AVL tree with LR-rotations.**

#include <iostream>

using namespace std;

// Define the structure of a tree node

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

// Function to get the height of a node

int height(Node\* node) {

return node ? node->height : 0;

}

// Function to calculate the balance factor of a node

int getBalance(Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

// Function to create a new node

Node\* createNode(int key) {

Node\* node = new Node();

node->key = key;

node->left = node->right = nullptr;

node->height = 1;

return node;

}

// Right Rotation

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

// Left Rotation

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

// LR Rotation (Left-Right Rotation)

Node\* leftRightRotate(Node\* z) {

// Perform left rotation on left child

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

// Perform right rotation on the unbalanced node

return rightRotate(z);

}

// Insert a node into the AVL tree

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

if (!node)

return createNode(key);

// Insert key as per BST property

if (key < node->key)

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

else if (key > node->key)

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

else

return node; // Duplicates not allowed

// Update the height of the current node

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

// Get the balance factor

int balance = getBalance(node);

// Left-Right (LR) case

if (balance > 1 && key > node->left->key)

return leftRightRotate(node);

// Left-Left (LL) case

if (balance > 1 && key < node->left->key)

return rightRotate(node);

// Right-Right (RR) case

if (balance < -1 && key > node->right->key)

return leftRotate(node);

// Right-Left (RL) case

if (balance < -1 && key < node->right->key) {

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

return leftRotate(node);

}

return node;

}

// Function to print the tree (In-order Traversal)

void inOrder(Node\* root) {

if (root) {

inOrder(root->left);

cout << root->key << " ";

inOrder(root->right);

}

}

int main() {

Node\* root = nullptr;

// Insert nodes into the AVL tree

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 10);

root = insert(root, 25); // Triggers LR rotation

cout << "In-order traversal of the AVL tree:" << endl;

inOrder(root);

return 0;

}

**4. Write C++ program to implement Heapify .**

#include <iostream>

#include <vector>

using namespace std;

// Function to heapify a subtree rooted at index `i`

// `n` is the size of the heap (or array)

void heapify(vector<int>& arr, int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child index

int right = 2 \* i + 2; // Right child index

// If left child is larger than root

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

largest = left;

// If right child is larger than the largest so far

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

largest = right;

// If the largest is not the root

if (largest != i) {

swap(arr[i], arr[largest]); // Swap root and largest

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Function to build a max-heap from the array

void buildMaxHeap(vector<int>& arr) {

int n = arr.size();

// Start from the last non-leaf node and heapify each subtree

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

heapify(arr, n, i);

}

}

// Function to print the array

void printArray(const vector<int>& arr) {

for (int val : arr) {

cout << val << " ";

}

cout << endl;

}

int main() {

// Example array

vector<int> arr = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};

cout << "Original array:" << endl;

printArray(arr);

// Build a max-heap from the array

buildMaxHeap(arr);

cout << "Array after heapify (Max-Heap):" << endl;

printArray(arr);

return 0;

}

**5. Write C++ program to implement Dijkstras shortest path algorithm with suitable**

**example.**

#include <iostream>

#include <vector>

#include <limits.h>

using namespace std;

// Function to find the vertex with the minimum distance value

int minDistance(vector<int>& dist, vector<bool>& sptSet, int V) {

int min = INT\_MAX, minIndex;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && dist[v] <= min) {

min = dist[v];

minIndex = v;

}

}

return minIndex;

}

// Function to implement Dijkstra's algorithm

void dijkstra(vector<vector<int>>& graph, int src) {

int V = graph.size();

vector<int> dist(V, INT\_MAX); // Distance values

vector<bool> sptSet(V, false); // Shortest path tree set

dist[src] = 0; // Distance to source is 0

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

sptSet[u] = true;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

// Print the distance array

cout << "Vertex\tDistance from Source" << endl;

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

cout << i << "\t" << dist[i] << endl;

}

}

int main() {

// Example graph (Adjacency Matrix)

vector<vector<int>> graph = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

int src = 0; // Source vertex

dijkstra(graph, src);

return 0;

}

**6. Write C++ program to implement Floyd Warshall algorithm with suitable example.**

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

#define INF numeric\_limits<int>::max()

void printSolution(const vector<vector<int>>& dist) {

int V = dist.size();

cout << "Shortest distances between every pair of vertices:" << endl;

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

for (int j = 0; j < V; j++) {

if (dist[i][j] == INF)

cout << "INF" << "\t";

else

cout << dist[i][j] << "\t";

}

cout << endl;

}

}

void floydWarshall(vector<vector<int>>& graph) {

int V = graph.size();

vector<vector<int>> dist = graph;

// Update the solution matrix by considering all vertices as intermediate vertices

for (int k = 0; k < V; k++) {

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

for (int j = 0; j < V; j++) {

// Skip if k is not on a valid path between i and j

if (dist[i][k] != INF && dist[k][j] != INF &&

dist[i][j] > dist[i][k] + dist[k][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

printSolution(dist);

}

int main() {

// Example graph with 4 vertices

vector<vector<int>> graph = {

{0, 3, INF, 5},

{2, 0, INF, 4},

{INF, 1, 0, INF},

{INF, INF, 2, 0}

};

cout << "Graph represented as adjacency matrix (INF means no direct edge):" << endl;

for (const auto& row : graph) {

for (const auto& value : row) {

if (value == INF)

cout << "INF\t";

else

cout << value << "\t";

}

cout << endl;

}

cout << endl;

floydWarshall(graph);

return 0;

}

**7. Write C++ program to perform the following operations on SET: a) Union**

#include <iostream>

#include <set>

#include <iterator>

#include <algorithm> // Include this for set\_union

using namespace std;

void displaySet(const set<int>& s) {

for (const auto& elem : s) {

cout << elem << " ";

}

cout << endl;

}

int main() {

set<int> setA = {1, 2, 3, 4};

set<int> setB = {3, 4, 5, 6};

set<int> setUnion;

cout << "Set A: ";

displaySet(setA);

cout << "Set B: ";

displaySet(setB);

// Perform union operation

set\_union(setA.begin(), setA.end(), setB.begin(), setB.end(),

inserter(setUnion, setUnion.begin()));

cout << "Union of Set A and Set B: ";

displaySet(setUnion);

return 0;

}

**8. Write C++ programs to implement: a) Sequential File.**#include <iostream>

#include <fstream>

#include <string>

using namespace std;

// Function to add a new record to the file

void addRecord(const string& filename) {

ofstream file(filename, ios::app); // Open in append mode

if (!file) {

cerr << "Error opening file!" << endl;

return;

}

string name;

int age;

cout << "Enter name: ";

cin >> name;

cout << "Enter age: ";

cin >> age;

file << name << " " << age << endl;

file.close();

cout << "Record added successfully!" << endl;

}

// Function to display all records

void displayRecords(const string& filename) {

ifstream file(filename);

if (!file) {

cerr << "Error opening file!" << endl;

return;

}

string name;

int age;

cout << "Records in the file:" << endl;

while (file >> name >> age) {

cout << "Name: " << name << ", Age: " << age << endl;

}

file.close();

}

// Function to search for a specific record

void searchRecord(const string& filename, const string& searchName) {

ifstream file(filename);

if (!file) {

cerr << "Error opening file!" << endl;

return;

}

string name;

int age;

bool found = false;

while (file >> name >> age) {

if (name == searchName) {

cout << "Record found: Name: " << name << ", Age: " << age << endl;

found = true;

break;

}

}

if (!found) {

cout << "Record not found!" << endl;

}

file.close();

}

int main() {

string filename = "records.txt";

int choice;

do {

cout << "\n--- Sequential File Operations ---" << endl;

cout << "1. Add Record" << endl;

cout << "2. Display All Records" << endl;

cout << "3. Search for a Record" << endl;

cout << "4. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

addRecord(filename);

break;

case 2:

displayRecords(filename);

break;

case 3: {

string searchName;

cout << "Enter name to search: ";

cin >> searchName;

searchRecord(filename, searchName);

break;

}

case 4:

cout << "Exiting..." << endl;

break;

default:

cout << "Invalid choice! Try again." << endl;

}

} while (choice != 4);

return 0;

}

**9. Write C++ program to implement AVL tree with RR-rotations.**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int key;

Node\* left;

Node\* right;

int height;

Node(int val) : key(val), left(nullptr), right(nullptr), height(1) {}

};

// Function to get the height of the tree

int getHeight(Node\* node) {

return node ? node->height : 0;

}

// Calculate the balance factor of a node

int getBalanceFactor(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

// Update the height of a node

void updateHeight(Node\* node) {

if (node) {

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

}

}

// Right-Right (RR) rotation

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

updateHeight(y);

updateHeight(x);

return x;

}

// Left-Left (LL) rotation

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

updateHeight(x);

updateHeight(y);

return y;

}

// Insert a node into the AVL tree

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

// Perform standard BST insertion

if (!node) {

return new Node(key);

}

if (key < node->key) {

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

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

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

} else {

return node; // Duplicates are not allowed

}

// Update the height of the node

updateHeight(node);

// Get the balance factor

int balance = getBalanceFactor(node);

// Balance the node if it becomes unbalanced

if (balance > 1 && key < node->left->key) {

return rightRotate(node); // Right-Right rotation

}

if (balance < -1 && key > node->right->key) {

return leftRotate(node); // Left-Left rotation

}

if (balance > 1 && key > node->left->key) {

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

return rightRotate(node);

}

if (balance < -1 && key < node->right->key) {

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

return leftRotate(node);

}

return node; // Return the unchanged node pointer

}

// In-order traversal of the AVL tree

void inOrderTraversal(Node\* root) {

if (root) {

inOrderTraversal(root->left);

cout << root->key << " ";

inOrderTraversal(root->right);

}

}

// Main function

int main() {

Node\* root = nullptr;

// Insert elements into the AVL tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

cout << "In-order traversal of the AVL tree: ";

inOrderTraversal(root);

cout << endl;

return 0;

}

**10. Write C++ program to implement AVL tree with LL-rotations.**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int key;

Node\* left;

Node\* right;

int height;

Node(int val) : key(val), left(nullptr), right(nullptr), height(1) {}

};

// Function to get the height of a node

int getHeight(Node\* node) {

return node ? node->height : 0;

}

// Calculate the balance factor of a node

int getBalanceFactor(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

// Update the height of a node

void updateHeight(Node\* node) {

if (node) {

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

}

}

// Left-Left (LL) rotation

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

updateHeight(x);

updateHeight(y);

return y;

}

// Right-Right (RR) rotation

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

updateHeight(y);

updateHeight(x);

return x;

}

// Insert a node into the AVL tree

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

// Perform standard BST insertion

if (!node) {

return new Node(key);

}

if (key < node->key) {

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

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

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

} else {

return node; // Duplicates are not allowed

}

// Update the height of the node

updateHeight(node);

// Get the balance factor

int balance = getBalanceFactor(node);

// Balance the node if it becomes unbalanced

if (balance > 1 && key < node->left->key) {

return rightRotate(node); // Left-Left rotation

}

if (balance < -1 && key > node->right->key) {

return leftRotate(node); // Right-Right rotation

}

if (balance > 1 && key > node->left->key) {

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

return rightRotate(node);

}

if (balance < -1 && key < node->right->key) {

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

return leftRotate(node);

}

return node; // Return the unchanged node pointer

}

// In-order traversal of the AVL tree

void inOrderTraversal(Node\* root) {

if (root) {

inOrderTraversal(root->left);

cout << root->key << " ";

inOrderTraversal(root->right);

}

}

// Main function

int main() {

Node\* root = nullptr;

// Insert elements into the AVL tree

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 2);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 1);

cout << "In-order traversal of the AVL tree: ";

inOrderTraversal(root);

cout << endl;

return 0;

}

**11. Write C++ program to implement AVL tree with RL-rotations.**#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

int height;

};

int getHeight(Node\* n) {

return n ? n->height : 0;

}

int getBalanceFactor(Node\* n) {

return n ? getHeight(n->left) - getHeight(n->right) : 0;

}

Node\* createNode(int data) {

Node\* node = new Node();

node->data = data;

node->left = node->right = nullptr;

node->height = 1;

return node;

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T = x->right;

x->right = y;

y->left = T;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T = y->left;

y->left = x;

x->right = T;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

return y;

}

Node\* insert(Node\* node, int data) {

if (!node)

return createNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

else

return node;

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

int balance = getBalanceFactor(node);

if (balance > 1 && data > node->left->data) {

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

return rightRotate(node);

}

if (balance < -1 && data < node->right->data) {

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

return leftRotate(node);

}

return node;

}

void preOrder(Node\* root) {

if (root) {

cout << root->data << " ";

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 5);

root = insert(root, 8);

root = insert(root, 15);

cout << "Pre-order Traversal of AVL Tree with RL Rotations:\n";

preOrder(root);

return 0;

}

12.Write c++ program to create max-heap  
#include <iostream>

#include <vector>

using namespace std;

void heapify(vector<int>& heap, int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

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

largest = right;

if (largest != i) {

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

heapify(heap, n, largest);

}

}

void buildMaxHeap(vector<int>& heap) {

int n = heap.size();

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

heapify(heap, n, i);

}

void printHeap(const vector<int>& heap) {

for (int val : heap)

cout << val << " ";

cout << endl;

}

int main() {

vector<int> heap = {3, 9, 2, 1, 4, 5};

buildMaxHeap(heap);

cout << "Max-Heap: ";

printHeap(heap);

return 0;

}

13.Write C++ program to create min –heap.  
#include <iostream>

#include <vector>

using namespace std;

void heapify(vector<int>& heap, int n, int i) {

int smallest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && heap[left] < heap[smallest])

smallest = left;

if (right < n && heap[right] < heap[smallest])

smallest = right;

if (smallest != i) {

swap(heap[i], heap[smallest]);

heapify(heap, n, smallest);

}

}

void buildMinHeap(vector<int>& heap) {

int n = heap.size();

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

heapify(heap, n, i);

}

void printHeap(const vector<int>& heap) {

for (int val : heap)

cout << val << " ";

cout << endl;

}

int main() {

vector<int> heap = {9, 4, 7, 1, -2, 6, 5};

buildMinHeap(heap);

cout << "Min-Heap: ";

printHeap(heap);

return 0;

}

14 Write C++ program to perform the following operations on SET: Intersection .  
#include <iostream>

#include <set>

using namespace std;

void printSet(const set<int>& s) {

for (int elem : s)

cout << elem << " ";

cout << endl;

}

int main() {

set<int> set1 = {1, 2, 3, 4, 5};

set<int> set2 = {3, 4, 5, 6, 7};

set<int> intersection;

for (int elem : set1) {

if (set2.find(elem) != set2.end())

intersection.insert(elem);

}

cout << "Intersection: ";

printSet(intersection);

return 0;

}

15. Write C++ program to traverse AVL tree: Pre-order .

#include <iostream>

using namespace std;

// AVL Tree Node Structure

struct Node {

int data;

Node\* left;

Node\* right;

int height;

Node(int val) {

data = val;

left = right = nullptr;

height = 1;

}

};

// Utility function to get the height of a node

int height(Node\* node) {

if (node == nullptr) return 0;

return node->height;

}

// Utility function to perform right rotation

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;

}

// Utility function to perform left rotation

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 balance factor of node

int getBalance(Node\* node) {

if (node == nullptr) return 0;

return height(node->left) - height(node->right);

}

// Insert node into AVL tree

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

if (node == nullptr) return new Node(key);

if (key < node->data) {

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

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

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

} else {

return node; // Duplicates not allowed

}

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

int balance = getBalance(node);

// Left heavy situation

if (balance > 1 && key < node->left->data)

return rightRotate(node);

// Right heavy situation

if (balance < -1 && key > node->right->data)

return leftRotate(node);

// Left-right heavy situation

if (balance > 1 && key > node->left->data) {

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

return rightRotate(node);

}

// Right-left heavy situation

if (balance < -1 && key < node->right->data) {

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

return leftRotate(node);

}

return node;

}

// Pre-order traversal of AVL tree

void preOrder(Node\* root) {

if (root == nullptr) return;

cout << root->data << " ";

preOrder(root->left);

preOrder(root->right);

}

int main() {

Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 15);

cout << "Pre-order traversal of AVL tree: ";

preOrder(root);

return 0;

}

16. . Write C++ program to traverse AVL tree: In-order .

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

int height;

Node(int val) {

data = val;

left = right = nullptr;

height = 1; // New node is initially at height 1

}

};

// Utility function to get the height of the tree

int height(Node\* node) {

return node ? node->height : 0;

}

// Utility function to get the balance factor of a node

int getBalance(Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

// Right rotate utility

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

// Return the new root

return x;

}

// Left rotate utility

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

// Return the new root

return y;

}

// Insert a node into the AVL tree and balance it

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

// Step 1: Perform the normal BST insert

if (!node) return new Node(key);

if (key < node->data)

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

else if (key > node->data)

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

else // Duplicate keys are not allowed

return node;

// Step 2: Update the height of this ancestor node

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

// Step 3: Get the balance factor and balance the tree if needed

int balance = getBalance(node);

// Left heavy (left-left case)

if (balance > 1 && key < node->left->data)

return rightRotate(node);

// Right heavy (right-right case)

if (balance < -1 && key > node->right->data)

return leftRotate(node);

// Left-right case

if (balance > 1 && key > node->left->data) {

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

return rightRotate(node);

}

// Right-left case

if (balance < -1 && key < node->right->data) {

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

return leftRotate(node);

}

return node; // Return the (unchanged) node pointer

}

// In-order traversal

void inorder(Node\* root) {

if (root == nullptr)

return;

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

// Main function

int main() {

Node\* root = nullptr;

// Insert nodes into the AVL tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 15);

root = insert(root, 25);

root = insert(root, 5);

// In-order traversal

cout << "In-order traversal of AVL tree: ";

inorder(root);

cout << endl;

return 0;

}

17. Write C++ program to traverse AVL tree: Post-order .

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

int height;

Node(int val) {

data = val;

left = right = nullptr;

height = 1; // New node is initially at height 1

}

};

// Utility function to get the height of the tree

int height(Node\* node) {

return node ? node->height : 0;

}

// Utility function to get the balance factor of a node

int getBalance(Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

// Right rotate utility

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

// Return the new root

return x;

}

// Left rotate utility

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

// Return the new root

return y;

}

// Insert a node into the AVL tree and balance it

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

// Step 1: Perform the normal BST insert

if (!node) return new Node(key);

if (key < node->data)

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

else if (key > node->data)

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

else // Duplicate keys are not allowed

return node;

// Step 2: Update the height of this ancestor node

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

// Step 3: Get the balance factor and balance the tree if needed

int balance = getBalance(node);

// Left heavy (left-left case)

if (balance > 1 && key < node->left->data)

return rightRotate(node);

// Right heavy (right-right case)

if (balance < -1 && key > node->right->data)

return leftRotate(node);

// Left-right case

if (balance > 1 && key > node->left->data) {

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

return rightRotate(node);

}

// Right-left case

if (balance < -1 && key < node->right->data) {

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

return leftRotate(node);

}

return node; // Return the (unchanged) node pointer

}

// Post-order traversal

void postOrder(Node\* root) {

if (root == nullptr)

return;

postOrder(root->left); // Traverse left subtree

postOrder(root->right); // Traverse right subtree

cout << root->data << " "; // Visit node

}

// Main function

int main() {

Node\* root = nullptr;

// Insert nodes into the AVL tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 15);

root = insert(root, 25);

root = insert(root, 5);

// Post-order traversal

cout << "Post-order traversal of AVL tree: ";

postOrder(root);

cout << endl;

return 0;

}

18. C++ program to demonstrate the creation of descending

order set container

#include <iostream>

#include <set>

#include <functional> // For std::greater

using namespace std;

int main() {

// Create a set of integers with descending order using std::greater

set<int, greater<int>> descSet;

// Inserting elements into the set

descSet.insert(10);

descSet.insert(20);

descSet.insert(5);

descSet.insert(15);

descSet.insert(25);

// Display the elements of the set in descending order

cout << "Elements in descending order: ";

for (const auto& element : descSet) {

cout << element << " ";

}

cout << endl;

// Demonstrating that duplicate elements are not allowed

descSet.insert(15); // This will not be inserted again since 15 is already in the set

cout << "After trying to insert duplicate 15: ";

for (const auto& element : descSet) {

cout << element << " ";

}

cout << endl;

return 0;

}

19.Write a cpp program to implement Naive Pattern Searching algorithm.

#include <iostream>

#include <string>

using namespace std;

// Naive Pattern Searching algorithm

void naiveSearch(const string& text, const string& pattern) {

int n = text.length();

int m = pattern.length();

// Loop through the text and check for pattern at every position

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

int j = 0;

// Check if the pattern matches at position i in the text

while (j < m && text[i + j] == pattern[j]) {

j++;

}

// If the pattern is fully matched, print the position (index)

if (j == m) {

cout << "Pattern found at index " << i << endl;

}

}

}

int main() {

string text, pattern;

// Input the text and pattern

cout << "Enter the text: ";

getline(cin, text);

cout << "Enter the pattern to search: ";

getline(cin, pattern);

// Call the naive search function

naiveSearch(text, pattern);

return 0;

}

20. A C++ program to print topological sorting of a DAG

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Function to perform topological sort

void topologicalSort(int V, vector<vector<int>> &adj) {

vector<int> inDegree(V, 0);

// Calculate in-degree of each vertex

for (int u = 0; u < V; u++) {

for (int v : adj[u]) {

inDegree[v]++;

}

}

// Queue to store vertices with in-degree 0

queue<int> q;

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

if (inDegree[i] == 0) {

q.push(i);

}

}

vector<int> topoOrder;

while (!q.empty()) {

int u = q.front();

q.pop();

topoOrder.push\_back(u);

// Reduce in-degree of adjacent vertices

for (int v : adj[u]) {

inDegree[v]--;

if (inDegree[v] == 0) {

q.push(v);

}

}

}

// Check for cycles

if (topoOrder.size() != V) {

cout << "The graph contains a cycle and cannot be topologically sorted." << endl;

return;

}

// Print the topological order

cout << "Topological Sorting: ";

for (int v : topoOrder) {

cout << v << " ";

}

cout << endl;

}

int main() {

int V = 6; // Number of vertices

vector<vector<int>> adj(V);

// Adding edges to the graph

adj[5].push\_back(2);

adj[5].push\_back(0);

adj[4].push\_back(0);

adj[4].push\_back(1);

adj[2].push\_back(3);

adj[3].push\_back(1);

cout << "Graph edges:\n";

for (int u = 0; u < V; u++) {

for (int v : adj[u]) {

cout << u << " -> " << v << endl;

}

}

cout << endl;

topologicalSort(V, adj);

return 0;

}