**QUS: 1**

**SOL:**

**// #include <stdio.h>**

**#include<iostream>**

**#include<vector>**

**#include <algorithm>**

**using namespace std;**

**void rotate(vector<int>& arr, int k) {**

**int n = arr.size();**

**k = k % n;**

**reverse(arr.begin(),arr.end());**

**reverse(arr.begin(),arr.begin()+k);**

**reverse(arr.begin()+k,arr.end());**

**}**

**int main()**

**{**

**vector<int> arr;**

**int n;**

**cout<<"enter size of an array:";**

**cin>>n;**

**int k;cout<<"enter the value of k:";**

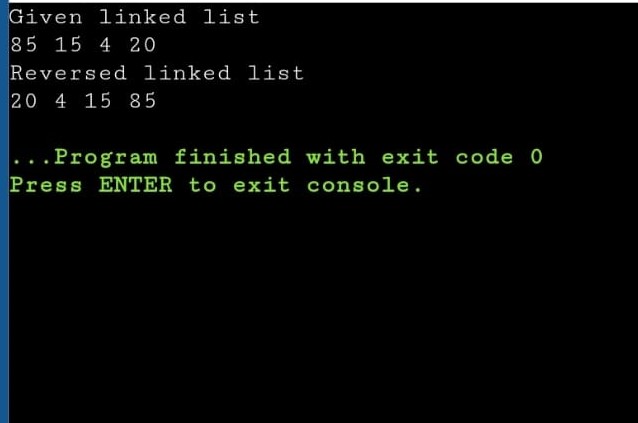
**cin>>k;**

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

**cout<<arr[i];**

**}**

**}**



**Qus: 2**

**Solution:**

**#include<iostream>**

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* reverseList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* current = head;

ListNode\* next = nullptr;

while (current != nullptr) {

// Store the next node

next = current->next;

// Reverse the current node's pointer

current->next = prev;

// Move pointers one position ahead

prev = current;

current = next;

}

// At the end, prev will be the new head

return prev;

}

// Helper function to print the linked list

void printList(ListNode\* head) {

while (head != nullptr) {

std::cout << head->val << " ";

head = head->next;

}

std::cout << std::endl;

}

// Helper function to create a new node

ListNode\* createNode(int value) {

return new ListNode(value);

}

int main() {

// Create a sample linked list: 1 -> 2 -> 3 -> 4 -> 5

ListNode\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

head->next->next->next->next = createNode(5);

head->next->next->next->next->next=createNode(6);

std::cout << "Original List: ";

printList(head);

// Reverse the linked list

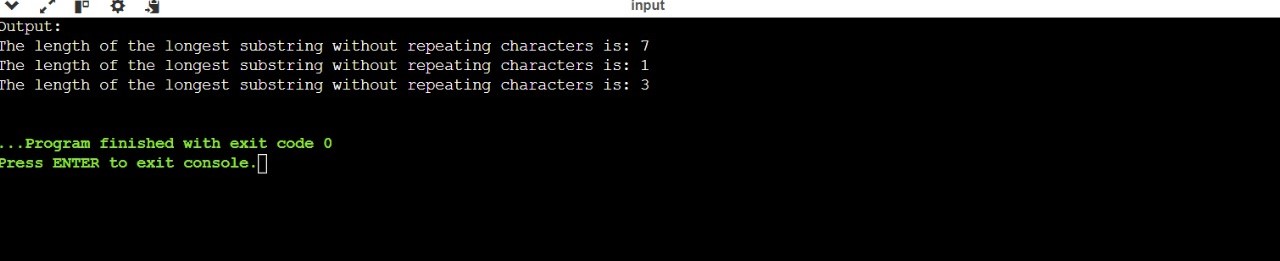
ListNode\* newHead = reverseList(head);

std::cout << "Reversed List: ";

printList(newHead);

    return 0;

}



Qus:3

Solution:

#include <iostream>

#include <unordered\_map>

#include <algorithm> // for std::max

int lengthOfLongestSubstring(const std::string& s) {

std::unordered\_map<char, int> charIndexMap;

int maxLength = 0;

int start = 0;

for (int end = 0; end < s.length(); ++end) {

char currentChar = s[end];

// If the character is already in the map, move the start position

// right after the last occurrence of the current character.

if (charIndexMap.find(currentChar) != charIndexMap.end()) {

start = std::max(start, charIndexMap[currentChar] + 1);

}

// Update the last seen index of the current character.

charIndexMap[currentChar] = end;

// Calculate the maximum length of the substring without repeating characters.

maxLength = std::max(maxLength, end - start + 1);

}

return maxLength;

}

int main() {

std::string s = "abcabcbb";

std::cout << "The length of the longest substring without repeating characters is: " << lengthOfLongestSubstring(s) << std::endl;

s = "bbbbb";

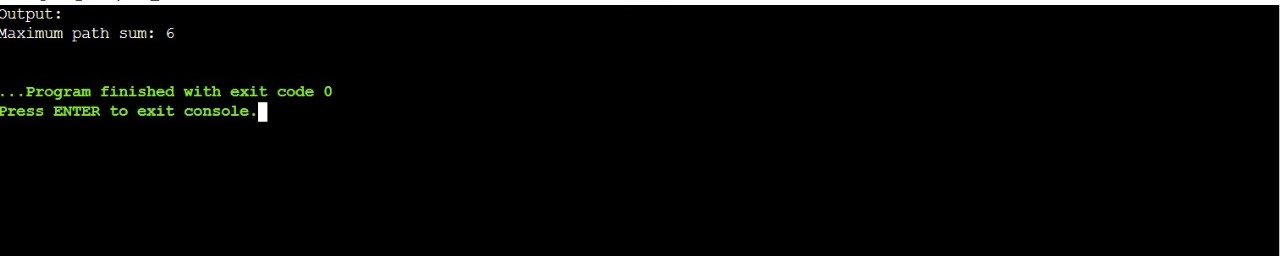
std::cout << "The length of the longest substring without repeating characters is: " << lengthOfLongestSubstring(s) << std::endl;

s = "pwwkew";

std::cout << "The length of the longest substring without repeating characters is: " << lengthOfLongestSubstring(s) << std::endl;

    return 0;

}



Qus:4

Solution:

#include <iostream>

#include <algorithm>

#include <bits/stdc++.h>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

int maxPathSumHelper(TreeNode\* root, int& maxSum) {

if (!root) return 0;

int left = std::max(0, maxPathSumHelper(root->left, maxSum));

int right = std::max(0, maxPathSumHelper(root->right, maxSum));

maxSum = std::max(maxSum, root->val + left + right);

return root->val + std::max(left, right);

}

int maxPathSum(TreeNode\* root) {

int maxSum = INT\_MIN;

maxPathSumHelper(root, maxSum);

return maxSum;

}

int main() {

TreeNode\* root = new TreeNode(1);

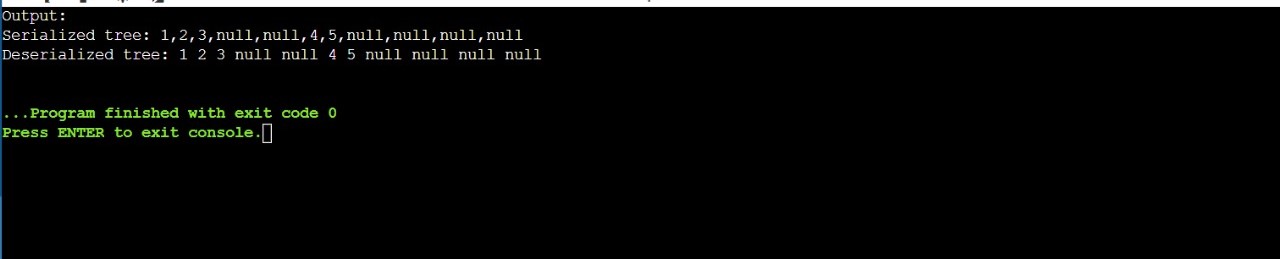
root->left = new TreeNode(2);

root->right = new TreeNode(3);

std::cout << "Maximum path sum: " << maxPathSum(root) << std::endl;

    return 0;

}



Qus:5

Solution:

#include <iostream>

#include <sstream>

#include <string>

#include <queue>

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

std::string serialize(TreeNode\* root) {

if (!root) return "null";

std::queue<TreeNode\*> q;

q.push(root);

std::string result;

while (!q.empty()) {

TreeNode\* node = q.front();

q.pop();

if (node) {

result += std::to\_string(node->val) + ",";

q.push(node->left);

q.push(node->right);

} else {

result += "null,";

}

}

return result.substr(0, result.size() - 1); // Remove trailing comma

}

TreeNode\* deserialize(const std::string& data) {

if (data == "null") return nullptr;

std::stringstream s(data);

std::string str;

std::getline(s, str, ',');

TreeNode\* root = new TreeNode(std::stoi(str));

std::queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* node = q.front();

q.pop();

if (std::getline(s, str, ',')) {

if (str != "null") {

node->left = new TreeNode(std::stoi(str));

q.push(node->left);

}

}

if (std::getline(s, str, ',')) {

if (str != "null") {

node->right = new TreeNode(std::stoi(str));

q.push(node->right);

}

}

}

return root;

}

void printTree(TreeNode\* root) {

if (!root) return;

std::queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* node = q.front();

q.pop();

if (node) {

std::cout << node->val << " ";

q.push(node->left);

q.push(node->right);

} else {

std::cout << "null ";

}

}

std::cout << std::endl;

}

int main() {

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->right->left = new TreeNode(4);

root->right->right = new TreeNode(5);

std::string serialized = serialize(root);

std::cout << "Serialized tree: " << serialized << std::endl;

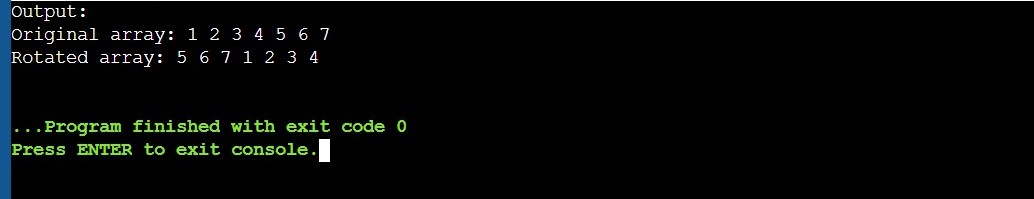
TreeNode\* deserialized = deserialize(serialized);

std::cout << "Deserialized tree: ";

printTree(deserialized);

    return 0;

}



Qus:6

Solution:

#include <iostream>

int factorial(int n) {

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

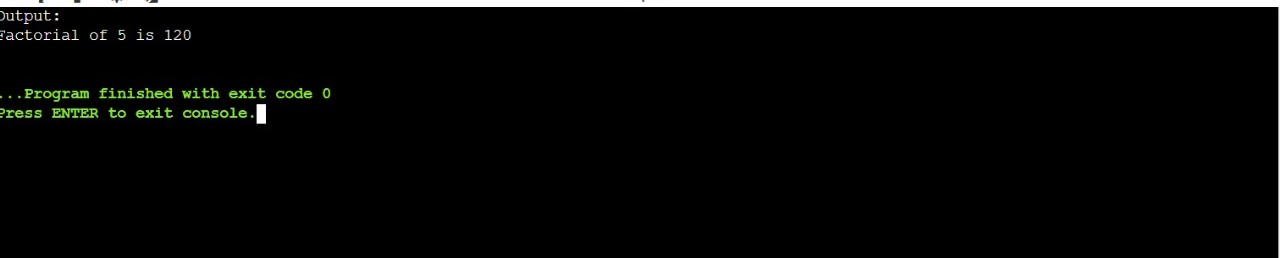
int main() {

int n = 5;

std::cout << "Factorial of " << n << " is " << factorial(n) << std::endl;

    return 0;

}



Qus:7

Solution:

#include <iostream>

int sumOfDigits(int n) {

int sum = 0;

while (n > 0) {

sum += n % 10;

n /= 10;

}

return sum;

}

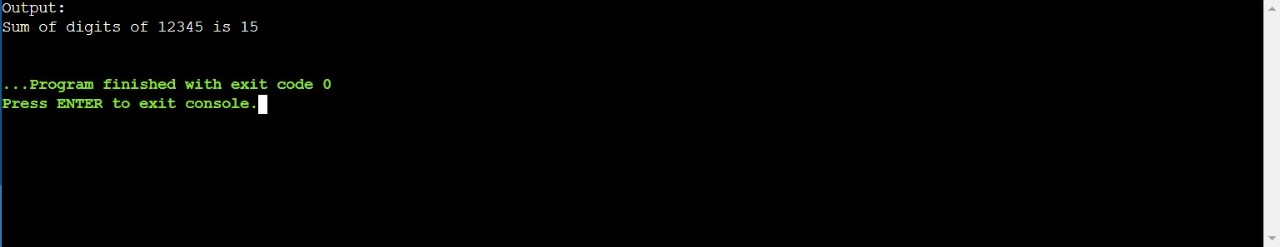
int main() {

int n = 12345;

std::cout << "Sum of digits of " << n << " is " << sumOfDigits(n) << std::endl;

    return 0;

}



Qus:8

Solution:

#include <iostream>

int gcd(int a, int b) {

while (b != 0) {

int temp = b;

b = a % b;

a = temp;

}

return a;

}

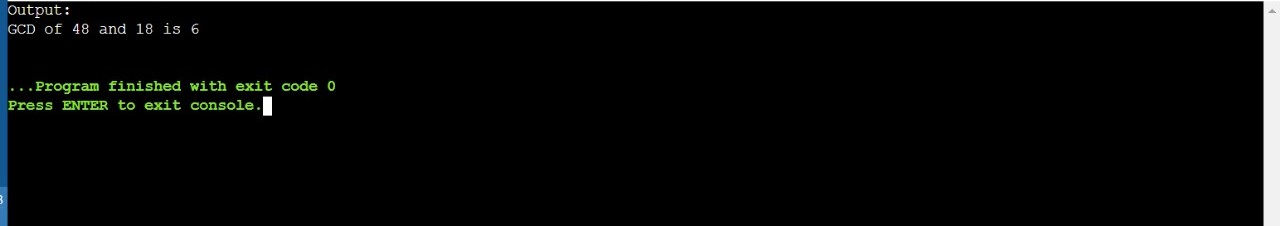
int main() {

int a = 48, b = 18;

std::cout << "GCD of " << a << " and " << b << " is " << gcd(a, b) << std::endl;

    return 0;

}



Qus:9

Solution:

#include <bits/stdc++.h>

using namespace std;

int maxAbsDiff(int arr[], int n)

{

//Sorting the array

sort(arr,arr+n);

//returning the difference between last and first element

return arr[n-1]-arr[0];

}

// Driver code

int main()

{

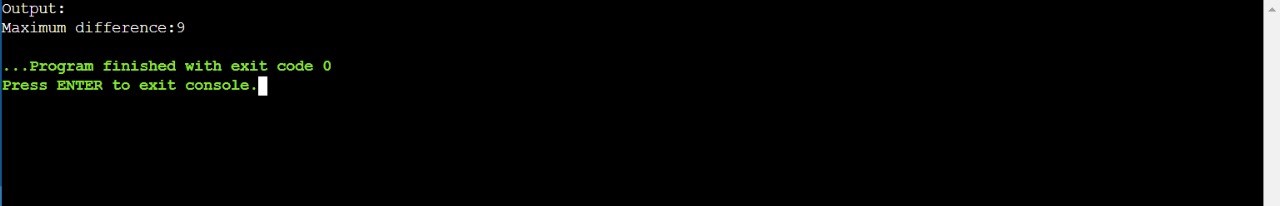
int arr[] = { 2, 3, 10, 2, 4, 8, 1};

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

cout <<"Maximum difference:"<< maxAbsDiff(arr, n);

    return 0;

}



Qus:10

Solution:

#include <iostream>

#include <cctype>

bool isAlphabetic(const std::string& s) {

for (char c : s) {

if (!std::isalpha(c)) {

return false;

}

}

return true;

}

int main() {

std::string s = "HelloWorld";

std::cout << "The string \"" << s << "\" contains only alphabetic characters: " << std::boolalpha << isAlphabetic(s) << std::endl;

s = "Hello Worl256d!";

std::cout << "The string \"" << s << "\" contains only alphabetic characters: " << std::boolalpha << isAlphabetic(s) << std::endl;

    return 0;

}

