|  |  |
| --- | --- |
| A picture of a winding road and trees  **Data structures**  **Miss IRSHA QURESHI** | **M.Abdullah Khalid**  **2023-BS-AI-035** |

**Single Link List**

**Question 1**

#include <iostream>

using namespace std;

// Define a Node of the doubly linked list

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// Function to delete a node by its value in a doubly linked list

bool deleteNodeByValue(Node\*& head, int value) {

if (!head) { // If the list is empty

cout << "The list is empty." << endl;

return false;

}

Node\* temp = head;

// Search for the node with the specified value

while (temp && temp->data != value) {

temp = temp->next;

}

if (!temp) { // Value not found

cout << "Value " << value << " not found in the list." << endl;

return false;

}

// Node with the value found

if (temp == head) { // If it's the head node

head = head->next;

if (head) {

head->prev = nullptr;

}

} else if (!temp->next) { // If it's the last node

temp->prev->next = nullptr;

} else { // If it's a middle node

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

delete temp; // Free the memory of the node

cout << "Node with value " << value << " deleted successfully." << endl;

return true;

}

// Function to display the doubly linked list

void displayList(Node\* head) {

if (!head) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

cout << "List: ";

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

// Function to insert a node at the end of the doubly linked list

void appendNode(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (!head) { // If the list is empty

head = newNode;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

// Add nodes to the doubly linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original list: ";

displayList(head);

// Delete a node by value

deleteNodeByValue(head, 20);

cout << "List after deleting the node with value 20: ";

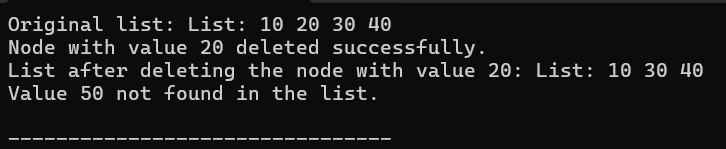
displayList(head);

// Try deleting a non-existent value

deleteNodeByValue(head, 50);

return 0;

}

**Output:**  


**Question 2**

#include <iostream>

using namespace std;

// Define a Node of the doubly linked list

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// Function to delete the first node of a doubly linked list

void deleteFirstNode(Node\*& head) {

if (head == nullptr) { // If the list is empty

cout << "The list is already empty." << endl;

return;

}

Node\* temp = head; // Store the current head node

head = head->next; // Move the head pointer to the next node

if (head != nullptr) { // If the list is not empty after deletion

head->prev = nullptr;

}

delete temp; // Free the memory of the old head node

cout << "First node deleted successfully." << endl;

}

// Function to display the doubly linked list

void displayList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

// Function to insert a node at the end of the doubly linked list

void appendNode(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == nullptr) { // If the list is empty

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

// Add nodes to the doubly linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

cout << "Original list: ";

displayList(head);

// Delete the first node

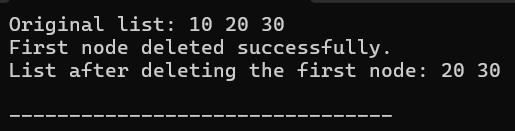
deleteFirstNode(head);

cout << "List after deleting the first node: ";

displayList(head);

return 0;

}

Output:  


**Question 3**

#include <iostream>

using namespace std;

// Define a Node of the doubly linked list

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// Function to delete the last node of a doubly linked list

void deleteLastNode(Node\*& head) {

if (head == nullptr) { // If the list is empty

cout << "The list is already empty." << endl;

return;

}

if (head->next == nullptr) { // If the list has only one node

delete head;

head = nullptr;

cout << "Last node deleted successfully." << endl;

return;

}

Node\* temp = head;

// Traverse to the last node

while (temp->next != nullptr) {

temp = temp->next;

}

// Update the previous node's next pointer

temp->prev->next = nullptr;

delete temp; // Free the memory of the last node

cout << "Last node deleted successfully." << endl;

}

// Function to display the doubly linked list

void displayList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

// Function to insert a node at the end of the doubly linked list

void appendNode(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == nullptr) { // If the list is empty

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

// Add nodes to the doubly linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

cout << "Original list: ";

displayList(head);

// Delete the last node

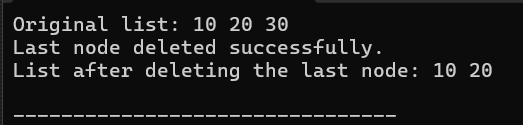
deleteLastNode(head);

cout << "List after deleting the last node: ";

displayList(head);

return 0;

}

Output:  


**Question 4**

#include <iostream>

using namespace std;

// Define a Node of the doubly linked list

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// Function to perform forward traversal of the doubly linked list

void forwardTraversal(Node\* head) {

cout << "Forward Traversal: ";

Node\* temp = head;

while (temp != nullptr) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

// Function to perform reverse traversal of the doubly linked list

void reverseTraversal(Node\* head) {

if (head == nullptr) { // If the list is empty

cout << "Reverse Traversal: List is empty." << endl;

return;

}

// Move to the last node

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

// Traverse backward from the last node

cout << "Reverse Traversal: ";

while (temp != nullptr) {

cout << temp->data << " ";

temp = temp->prev;

}

cout << endl;

}

// Function to append a node to the end of the doubly linked list

void appendNode(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == nullptr) { // If the list is empty

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

// Function to delete a node at a specific position in the doubly linked list

void deleteNodeAtPosition(Node\*& head, int position) {

if (head == nullptr) { // If the list is empty

cout << "The list is empty." << endl;

return;

}

if (position <= 0) { // Invalid position

cout << "Invalid position. Position should be greater than 0." << endl;

return;

}

Node\* temp = head;

int currentIndex = 1;

// Traverse to the node at the specified position

while (temp != nullptr && currentIndex < position) {

temp = temp->next;

currentIndex++;

}

if (temp == nullptr) { // Position exceeds the list size

cout << "Position " << position << " exceeds the list size." << endl;

return;

}

if (temp == head) { // Deleting the head node

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

} else if (temp->next == nullptr) { // Deleting the last node

temp->prev->next = nullptr;

} else { // Deleting a middle node

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

delete temp; // Free the memory of the node

cout << "Node at position " << position << " deleted successfully." << endl;

}

// Main function to demonstrate forward and reverse traversal

int main() {

Node\* head = nullptr;

// Add nodes to the doubly linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

// Perform forward and reverse traversal before deletion

forwardTraversal(head);

reverseTraversal(head);

// Delete a node at position 2

deleteNodeAtPosition(head, 2);

// Perform forward and reverse traversal after deletion

forwardTraversal(head);

reverseTraversal(head);

return 0;

}

**Output:  
A screen shot of a computer

Description automatically generated**

**SINGLE CIRCULAR**

**QUESTION 1**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

};

// Function to delete a node at a specific position in a circular linked list

void deleteNodeAtPosition(Node\*& head, int position) {

if (head == nullptr) { // List is empty

cout << "List is empty. Nothing to delete." << endl;

return;

}

// If the position is 0, delete the head node

if (position == 0) {

// If there's only one node

if (head->next == head) {

delete head;

head = nullptr;

} else {

Node\* last = head;

while (last->next != head) { // Find the last node

last = last->next;

}

Node\* temp = head;

head = head->next; // Move the head pointer

last->next = head; // Adjust the last node's next pointer

delete temp; // Delete the old head

}

return;

}

Node\* current = head;

Node\* previous = nullptr;

int count = 0;

// Traverse to the desired position

while (current->next != head && count < position) {

previous = current;

current = current->next;

count++;

}

// If position is out of bounds

if (current->next == head && count < position) {

cout << "Position out of bounds." << endl;

return;

}

// Delete the node

previous->next = current->next;

delete current;

}

// Function to insert a node at the end of the circular linked list

void insert(Node\*& head, int data) {

Node\* newNode = new Node();

newNode->data = data;

if (head == nullptr) {

head = newNode;

newNode->next = head;

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

// Function to display the circular linked list

void display(Node\* head) {

if (head == nullptr) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Main function

int main() {

Node\* head = nullptr;

// Insert some nodes

insert(head, 10);

insert(head, 20);

insert(head, 30);

insert(head, 40);

cout << "Original list: ";

display(head);

// Delete node at position 2

deleteNodeAtPosition(head, 2);

cout << "After deleting node at position 2: ";

display(head);

// Try to delete node at an invalid position

deleteNodeAtPosition(head, 5);

cout << "After attempting to delete node at position 5: ";

display(head);

return 0;

}

**Output:  
A black screen with white text

Description automatically generated**

**Question 2**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

};

// Function to delete a node by its value in a circular linked list

void deleteNodeByValue(Node\*& head, int value) {

if (head == nullptr) { // List is empty

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* current = head;

Node\* previous = nullptr;

// Case 1: The node to be deleted is the only node in the list

if (head->data == value && head->next == head) {

delete head;

head = nullptr;

cout << "Node with value " << value << " deleted." << endl;

return;

}

// Case 2: The node to be deleted is the head node

if (head->data == value) {

// Find the last node

while (current->next != head) {

current = current->next;

}

Node\* temp = head;

head = head->next;

current->next = head;

delete temp;

cout << "Node with value " << value << " deleted." << endl;

return;

}

// Case 3: The node to be deleted is in the middle or end of the list

do {

previous = current;

current = current->next;

if (current->data == value) {

previous->next = current->next;

delete current;

cout << "Node with value " << value << " deleted." << endl;

return;

}

} while (current != head);

// If the value was not found

cout << "Value " << value << " not found in the list." << endl;

}

// Function to insert a node at the end of the circular linked list

void insert(Node\*& head, int data) {

Node\* newNode = new Node();

if (!newNode) {

cout << "Memory allocation failed!" << endl;

return;

}

newNode->data = data;

if (head == nullptr) {

head = newNode;

newNode->next = head;

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

// Function to display the circular linked list

void display(Node\* head) {

if (head == nullptr) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Main function

int main() {

Node\* head = nullptr;

// Insert some nodes

insert(head, 10);

insert(head, 20);

insert(head, 30);

insert(head, 40);

cout << "Original list: ";

display(head);

// Delete a node by value

deleteNodeByValue(head, 20);

cout << "After deleting node with value 20: ";

display(head);

// Try to delete a node not in the list

deleteNodeByValue(head, 50);

cout << "After attempting to delete node with value 50: ";

display(head);

return 0;

}

Output:  
A black screen with white text

Description automatically generated

**QUESTION 3**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

};

// Function to delete the last node of a circular linked list

void deleteLastNode(Node\*& head) {

if (head == nullptr) { // List is empty

cout << "List is empty. Nothing to delete." << endl;

return;

}

if (head->next == head) { // Only one node in the list

delete head;

head = nullptr;

return;

}

// Traverse the list to find the second last node

Node\* current = head;

while (current->next->next != head) {

current = current->next;

}

// Adjust pointers and delete the last node

Node\* last = current->next;

current->next = head;

delete last;

}

// Function to insert a node at the end of the circular linked list

void insert(Node\*& head, int data) {

Node\* newNode = new Node();

newNode->data = data;

if (head == nullptr) {

head = newNode;

newNode->next = head;

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

// Function to display the circular linked list

void display(Node\* head) {

if (head == nullptr) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Main function

int main() {

Node\* head = nullptr;

// Insert some nodes

insert(head, 10);

insert(head, 20);

insert(head, 30);

insert(head, 40);

cout << "Original list: ";

display(head);

// Delete the last node

deleteLastNode(head);

cout << "After deleting the last node: ";

display(head);

return 0;

}

Output:

A black background with white text

Description automatically generated

**QUESTION 4**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

};

// Function to delete the first node of a circular linked list

void deleteFirstNode(Node\*& head) {

if (head == nullptr) { // List is empty

cout << "List is empty. Nothing to delete." << endl;

return;

}

if (head->next == head) { // Only one node in the list

delete head;

head = nullptr;

return;

}

// Find the last node in the list

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Point the last node to the second node

Node\* temp = head;

head = head->next;

last->next = head;

// Delete the first node

delete temp;

}

// Function to insert a node at the end of the circular linked list

void insert(Node\*& head, int data) {

Node\* newNode = new Node();

newNode->data = data;

if (head == nullptr) {

head = newNode;

newNode->next = head;

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

// Function to display the circular linked list

void display(Node\* head) {

if (head == nullptr) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Main function

int main() {

Node\* head = nullptr;

// Insert some nodes

insert(head, 10);

insert(head, 20);

insert(head, 30);

insert(head, 40);

cout << "Original list: ";

display(head);

// Delete the first node

deleteFirstNode(head);

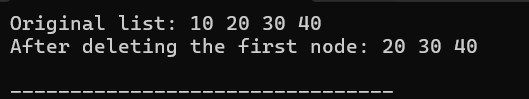
cout << "After deleting the first node: ";

display(head);

return 0;

}

Output:



**Question 5**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* next;

};

// Function to delete a node at a specific position in a circular linked list

void deleteNodeAtPosition(Node\*& head, int position) {

if (head == nullptr) { // List is empty

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* current = head;

// Case 1: Delete head node (position 0)

if (position == 0) {

// If there's only one node

if (head->next == head) {

delete head;

head = nullptr;

} else {

// Find the last node

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Update head and last node's next pointer

Node\* temp = head;

head = head->next;

last->next = head;

delete temp;

}

return;

}

// Case 2: Traverse to the position and delete the node

Node\* prev = nullptr;

current = head;

int count = 0;

while (current->next != head && count < position) {

prev = current;

current = current->next;

count++;

}

// If position is out of bounds

if (current->next == head && count < position) {

cout << "Position out of bounds." << endl;

return;

}

// Delete the node

prev->next = current->next;

delete current;

}

// Function to insert a node at the end of the circular linked list

void insert(Node\*& head, int data) {

Node\* newNode = new Node();

newNode->data = data;

// If the list is empty

if (head == nullptr) {

head = newNode;

newNode->next = head; // Circular link

return;

}

// Traverse to the last node

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Maintain circular link

}

// Function to display the circular linked list

void display(Node\* head) {

if (head == nullptr) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Main function

int main() {

Node\* head = nullptr;

// Insert some nodes

insert(head, 10);

insert(head, 20);

insert(head, 30);

insert(head, 40);

cout << "Original list: ";

display(head);

// Delete node at position 2

deleteNodeAtPosition(head, 2);

cout << "After deleting node at position 2: ";

display(head);

// Delete node at position 0 (head node)

deleteNodeAtPosition(head, 0);

cout << "After deleting node at position 0: ";

display(head);

return 0;

}

Output:  
A black background with white text

Description automatically generated

**TREE**

**QUESTION 1**

#include <iostream>

using namespace std;

// Node structure for Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Function to insert a node in the Binary Search Tree

Node\* insert(Node\* root, int value) {

if (root == nullptr) {

return new Node(value);

}

if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}

return root;

}

// Function to count the nodes in the Binary Search Tree

int countNodes(Node\* root) {

if (root == nullptr) {

return 0;

}

// Recursively count nodes in the left and right subtrees, and add 1 for the current node

return 1 + countNodes(root->left) + countNodes(root->right);

}

// Function to perform an in-order traversal and print the tree

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Main function

int main() {

Node\* root = nullptr;

// Insert nodes into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "In-order traversal of the Binary Search Tree: ";

inorderTraversal(root);

cout << endl;

// Count the nodes in the BST

int nodeCount = countNodes(root);

cout << "Total number of nodes in the BST: " << nodeCount << endl;

return 0;

}

Output:

A black screen with white text

Description automatically generated

**QUESTION 2**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

// Constructor to initialize a new node

Node(int val) : data(val), left(NULL), right(NULL) {}

};

// Function to find the minimum value node (leftmost node)

Node\* findMin(Node\* root) {

while (root && root->left) {

root = root->left;

}

return root;

}

// Function to delete a node from the binary search tree

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

// If the tree is empty

if (!root) return nullptr;

// Traverse the tree to find the node to delete

if (key < root->data) {

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

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

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

} else {

// Node found to delete

if (!root->left) {

// No left child, so we can just return the right subtree

Node\* temp = root->right;

delete root;

return temp;

} else if (!root->right) {

// No right child, so we return the left subtree

Node\* temp = root->left;

delete root;

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

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

root->data = temp->data; // Copy the inorder successor's value to this node

root->right = deleteNode(root->right, temp->data); // Delete the inorder successor

}

return root;

}

// Inorder traversal to print the tree (for verification)

void inorderTraversal(Node\* root) {

if (root) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Main function to test deletion

int main() {

// Creating a binary search tree

Node\* root = new Node(10);

root->left = new Node(5);

root->right = new Node(15);

root->left->left = new Node(3);

root->left->right = new Node(7);

root->right->left = new Node(12);

root->right->right = new Node(18);

// Print the original tree

cout << "Original tree (Inorder Traversal): ";

inorderTraversal(root);

cout << endl;

// Delete a node (value 5)

cout << "Deleting node with value 5." << endl;

root = deleteNode(root, 5);

// Print the tree after deletion

cout << "Tree after deletion (Inorder Traversal): ";

inorderTraversal(root);

cout << endl;

return 0;

}

Output:

**A black screen with white text

Description automatically generated**

**Question 3**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = nullptr;

right = nullptr;

}

};

// Function to insert a value (with duplicate check)

Node\* insert(Node\* root, int val) {

if (root == nullptr) {

return new Node(val);

}

if (val < root->data) {

root->left = insert(root->left, val);

} else if (val > root->data) {

root->right = insert(root->right, val);

} else {

cout << "Duplicate value " << val << " not allowed." << endl;

}

return root;

}

// Function for inorder traversal to display the tree

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Main function to test duplication handling and display the tree

int main() {

Node\* root = nullptr;

// Insert some values into the tree

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 12);

root = insert(root, 10); // Duplicate Values

root = insert(root, 18);

// Display the tree in inorder traversal

cout << "Inorder Traversal of the Tree: ";

inorderTraversal(root);

cout << endl;

return 0;

}

Output:  
A black screen with white text

Description automatically generated

**QUESTION 4**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

// Constructor to initialize the node

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Inorder Traversal function to print the tree

void inorder(Node\* root) {

if (root == nullptr) return;

inorder(root->left);

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

inorder(root->right);

}

// Function to insert a new value into the binary search tree

Node\* insert(Node\* root, int val) {

if (root == nullptr) {

return new Node(val); // Insert the new node

}

if (val < root->data) {

root->left = insert(root->left, val); // Insert in the left subtree

} else if (val > root->data) {

root->right = insert(root->right, val); // Insert in the right subtree

} else {

cout << "Duplicate value " << val << " not allowed." << endl; // Handle duplicates

}

return root;

}

// Main function to test insertion and inorder traversal

int main() {

Node\* root = nullptr; // Initialize the root as NULL

// Insert values into the binary search tree

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 3); // Adding more values for testing

root = insert(root, 7);

root = insert(root, 12);

// Print the inorder traversal of the tree

cout << "Inorder Traversal of the Tree: ";

inorder(root);

cout << endl;

cout << "Values inserted successfully!" << endl;

return 0;

}

Output:  
A black screen with white text

Description automatically generated

**QUESTION 5**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

// Function to search for a value

bool search(Node\* root, int key) {

if (root == NULL) return false;

if (root->data == key) return true;

if (key < root->data) return search(root->left, key);

return search(root->right, key);

}

// Main function to test searching

int main() {

Node\* root = new Node(10);

root->left = new Node(5);

root->right = new Node(15);

int searchKey = 5;

if (search(root, searchKey)) {

cout << "Value " << searchKey << " found in the tree." << endl;

} else {

cout << "Value " << searchKey << " not found in the tree." << endl;

}

return 0;

}

Output:  
A black screen with white text

Description automatically generated

**QUESTION 6**

#include <iostream>

using namespace std;

// Node class definition

class Node {

public:

int data;

Node\* left;

Node\* right;

// Constructor to initialize the node

Node(int val) {

data = val;

left = nullptr;

right = nullptr;

}

};

// BinarySearchTree class to encapsulate tree functions

class BinarySearchTree {

private:

Node\* root;

// Helper function to insert a new node

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

if (node == nullptr) {

return new Node(val); // If node is null, create a new node

}

if (val < node->data) {

node->left = insert(node->left, val); // Insert in the left subtree

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

node->right = insert(node->right, val); // Insert in the right subtree

}

return node; // Return the unchanged node pointer

}

// Helper function for inorder traversal

void inorder(Node\* node) {

if (node == nullptr) return;

inorder(node->left); // Visit left child

cout << node->data << " "; // Print node data

inorder(node->right); // Visit right child

}

// Helper function for preorder traversal

void preorder(Node\* node) {

if (node == nullptr) return;

cout << node->data << " "; // Print node data

preorder(node->left); // Visit left child

preorder(node->right); // Visit right child

}

// Helper function for postorder traversal

void postorder(Node\* node) {

if (node == nullptr) return;

postorder(node->left); // Visit left child

postorder(node->right); // Visit right child

cout << node->data << " "; // Print node data

}

public:

// Constructor to initialize the root to null

BinarySearchTree() {

root = nullptr;

}

// Function to insert a value into the tree

void insert(int val) {

root = insert(root, val); // Call the recursive insert helper function

}

// Function to print inorder traversal

void inorder() {

inorder(root); // Start from root

cout << endl;

}

// Function to print preorder traversal

void preorder() {

preorder(root); // Start from root

cout << endl;

}

// Function to print postorder traversal

void postorder() {

postorder(root); // Start from root

cout << endl;

}

};

// Main function to test the tree

int main() {

BinarySearchTree bst;

// Inserting values into the binary search tree

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(3);

bst.insert(7);

bst.insert(12);

bst.insert(18);

// Print the different tree traversals

cout << "Inorder Traversal: ";

bst.inorder(); // Output: 3 5 7 10 12 15 18

cout << "Preorder Traversal: ";

bst.preorder(); // Output: 10 5 3 7 15 12 18

cout << "Postorder Traversal: ";

bst.postorder(); // Output: 3 7 5 12 18 15 10

return 0;

}

Output:  
A screenshot of a computer

Description automatically generated