

**Name:**M Zubair Khan

**ROLL NO:** BS-AI-006

**LAB MANUAL**

**Lab #01**

#include <iostream>

using namespace std;

void display(int arr[], int size) {

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

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

}

cout << endl;

}

void insertFront(int arr[], int &size, int value) {

for (int i = size; i > 0; i--) {

arr[i] = arr[i - 1];

}

arr[0] = value;

size++;

}

void insertMid(int arr[], int &size, int value) {

int pos = size / 2;

for (int i = size; i > pos; i--) {

arr[i] = arr[i - 1];

}

arr[pos] = value;

size++;

}

void insertLast(int arr[], int &size, int value) {

arr[size] = value;

size++;

}

void deleteFront(int arr[], int &size) {

for (int i = 0; i < size - 1; i++) {

arr[i] = arr[i + 1];

}

size--;

}

void deleteMid(int arr[], int &size) {

int pos = size / 2;

for (int i = pos; i < size - 1; i++) {

arr[i] = arr[i + 1];

}

size--;

}

void deleteLast(int arr[], int &size) {

size--;

}

int search(int arr[], int size, int value) {

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

if (arr[i] == value) {

return i;

}

}

return -1;

}

void update(int arr[], int index, int value) {

arr[index] = value;

}

int main() {

int arr[100], size = 0;

// Initial array

insertLast(arr, size, 10);

insertLast(arr, size, 20);

insertLast(arr, size, 30);

cout << "Initial array: ";

display(arr, size);

// Insertions

insertFront(arr, size, 5);

cout << "After inserting 5 at front: ";

display(arr, size);

insertMid(arr, size, 15);

cout << "After inserting 15 at mid: ";

display(arr, size);

insertLast(arr, size, 35);

cout << "After inserting 35 at last: ";

display(arr, size);

// Deletions

deleteFront(arr, size);

cout << "After deleting from front: ";

display(arr, size);

deleteMid(arr, size);

cout << "After deleting from mid: ";

display(arr, size);

deleteLast(arr, size);

cout << "After deleting from last: ";

display(arr, size);

// Searching

int index = search(arr, size, 20);

if (index != -1)

cout << "Element 20 found at index: " << index << endl;

else

cout << "Element 20 not found!" << endl;

// Updating

update(arr, 1, 25);

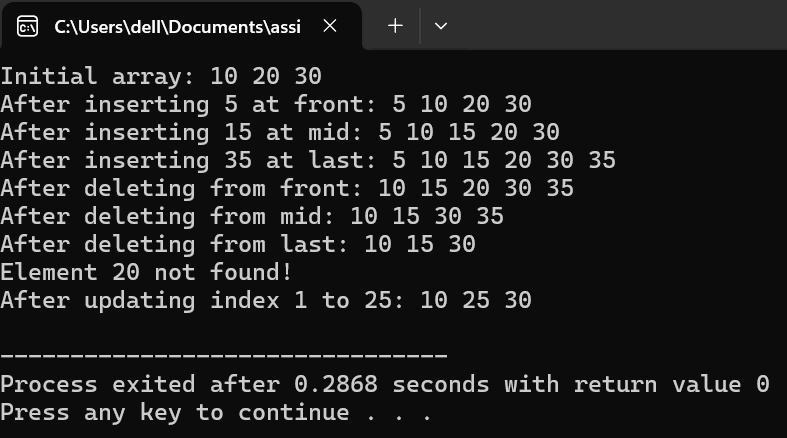
cout << "After updating index 1 to 25: ";

display(arr, size);

return 0;

}

**OUTPUT:**

****

**LAB 3:**

a stack is an abstract data type that follows the Last-In, First-Out (LIFO) principle.This means that the last element added to the stack is the first one to be removed.

**Program:**

#include <iostream>using namespace std;

#define MAX 100

struct Stack {

int arr[MAX];

int top;

};

// Initialize stackvoid initStack(Stack &s) {

s.top = -1;

}

// Check if the stack is emptybool isEmpty(Stack &s) {

return s.top == -1;

}

// Check if the stack is fullbool isFull(Stack &s) {

return s.top == MAX - 1;

}

// Push an element onto the stackvoid push(Stack &s, int value) {

if (isFull(s)) {

cout << "Stack Overflow! Cannot push " << value << endl;

return;

}

s.top++;

s.arr[s.top] = value;

cout << value << " pushed onto stack." << endl;

}

// Pop an element from the stackint pop(Stack &s) {

if (isEmpty(s)) {

cout << "Stack Underflow! Cannot pop." << endl;

return -1;

}

int poppedValue = s.arr[s.top];

s.top--;

cout << poppedValue << " popped from stack." << endl;

return poppedValue;

}

// Peek the top element of the stackint peek(Stack &s) {

if (isEmpty(s)) {

cout << "Stack is empty. No top element." << endl;

return -1;

}

return s.arr[s.top];

}

// Display the stack elementsvoid display(Stack &s) {

if (isEmpty(s)) {

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

return;

}

cout << "Stack elements: ";

for (int i = s.top; i >= 0; i--) {

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

}

cout << endl;

}

int main() {

Stack s;

initStack(s);

// Perform stack operations

push(s, 10);

push(s, 20);

push(s, 30);

display(s);

cout << "Top element: " << peek(s) << endl;

pop(s);

display(s);

pop(s);

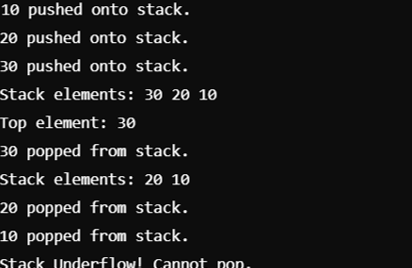
pop(s);

pop(s); // Attempt to pop from an empty stack

return 0;

}

**OUTPUT:**

****

**LAB 4**

**Program:**

#include <iostream>

#include <cstring>

// For strlen()

using namespace std;

#define MAX 100

struct Stack {

char arr[MAX];

int top;

};

// Initialize stackvoid initStack(Stack &s) {

s.top = -1;

}

// Check if the stack is emptybool isEmpty(Stack &s) {

return s.top == -1;

}

// Push an element onto the stackvoid push(Stack &s, char value) {

if (s.top == MAX - 1) {

cout << "Stack Overflow! Cannot push " << value << endl;

return;

}

s.arr[++s.top] = value;

}

// Pop an element from the stackchar pop(Stack &s) {

if (isEmpty(s)) {

cout << "Stack Underflow! Cannot pop." << endl;

return '\0';

}

return s.arr[s.top--];

}

// Peek the top element of the stackchar peek(Stack &s) {

if (isEmpty(s)) {

return '\0';

}

return s.arr[s.top];

}

// Check if a character is an operatorbool isOperator(char ch) {

return ch == '+' || ch == '-' || ch == '\*' || ch == '/';

}

// Get precedence of an operatorint precedence(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

return 0;

}

// Convert infix to postfixvoid infixToPostfix(char infix[], char postfix[]) {

Stack s;

initStack(s);

int j = 0;

for (int i = 0; infix[i] != '\0'; i++) {

char ch = infix[i];

if (isalnum(ch)) { // Operand

postfix[j++] = ch;

} else if (ch == '(') { // Left parenthesis

push(s, ch);

} else if (ch == ')') { // Right parenthesis

while (!isEmpty(s) && peek(s) != '(') {

postfix[j++] = pop(s);

}

pop(s); // Pop the '('

} else if (isOperator(ch)) { // Operator

while (!isEmpty(s) && precedence(peek(s)) >= precedence(ch)) {

postfix[j++] = pop(s);

}

push(s, ch);

}

}

// Pop all remaining operators from the stack

while (!isEmpty(s)) {

postfix[j++] = pop(s);

}

postfix[j] = '\0'; // Null terminate the postfix expression

}

int main() {

char infix[MAX], postfix[MAX];

cout << "Enter an infix expression (e.g., A+B\*C): ";

cin >> infix;

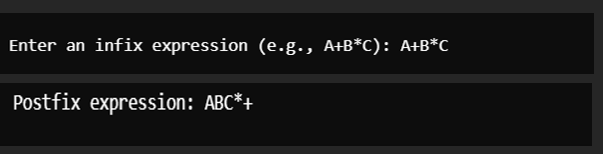
infixToPostfix(infix, postfix);

cout << "Postfix expression: " << postfix << endl;

return 0;

}

**OUTPUT:**



**LAB 5:**

**Program:**

**1. Linear Queue**

#include <iostream>using namespace std;

#define MAX 5

struct LinearQueue {

int arr[MAX];

int front, rear;

};

// Initialize the queuevoid initQueue(LinearQueue &q) {

q.front = -1;

q.rear = -1;

}

// Check if the queue is emptybool isEmpty(LinearQueue &q) {

return q.front == -1;

}

// Check if the queue is fullbool isFull(LinearQueue &q) {

return q.rear == MAX - 1;

}

// Enqueue operationvoid enqueue(LinearQueue &q, int value) {

if (isFull(q)) {

cout << "Queue Overflow! Cannot enqueue " << value << endl;

return;

}

if (isEmpty(q)) {

q.front = 0;

}

q.rear++;

q.arr[q.rear] = value;

cout << value << " enqueued into the queue." << endl;

}

// Dequeue operationint dequeue(LinearQueue &q) {

if (isEmpty(q)) {

cout << "Queue Underflow! Cannot dequeue." << endl;

return -1;

}

int value = q.arr[q.front];

if (q.front == q.rear) { // Single element

q.front = q.rear = -1;

} else {

q.front++;

}

cout << value << " dequeued from the queue." << endl;

return value;

}

// Display the queuevoid display(LinearQueue &q) {

if (isEmpty(q)) {

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

return;

}

cout << "Queue elements: ";

for (int i = q.front; i <= q.rear; i++) {

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

}

cout << endl;

}

int main() {

LinearQueue q;

initQueue(q);

enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

enqueue(q, 40);

enqueue(q, 50);

enqueue(q, 60); // Overflow case

display(q);

dequeue(q);

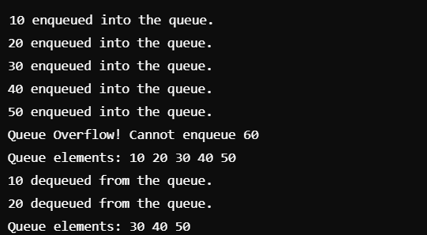
dequeue(q);

display(q);

return 0;

}

**OUTPUT:**

****

**2. Circular Queue**

#include <iostream>using namespace std;

#define MAX 5

struct CircularQueue {

int arr[MAX];

int front, rear;

};

// Initialize the queuevoid initQueue(CircularQueue &q) {

q.front = -1;

q.rear = -1;

}

// Check if the queue is emptybool isEmpty(CircularQueue &q) {

return q.front == -1;

}

// Check if the queue is fullbool isFull(CircularQueue &q) {

return (q.rear + 1) % MAX == q.front;

}

// Enqueue operationvoid enqueue(CircularQueue &q, int value) {

if (isFull(q)) {

cout << "Queue Overflow! Cannot enqueue " << value << endl;

return;

}

if (isEmpty(q)) {

q.front = q.rear = 0;

} else {

q.rear = (q.rear + 1) % MAX;

}

q.arr[q.rear] = value;

cout << value << " enqueued into the circular queue." << endl;

}

// Dequeue operationint dequeue(CircularQueue &q) {

if (isEmpty(q)) {

cout << "Queue Underflow! Cannot dequeue." << endl;

return -1;

}

int value = q.arr[q.front];

if (q.front == q.rear) { // Single element

q.front = q.rear = -1;

} else {

q.front = (q.front + 1) % MAX;

}

cout << value << " dequeued from the circular queue." << endl;

return value;

}

// Display the queuevoid display(CircularQueue &q) {

if (isEmpty(q)) {

cout << "Circular Queue is empty." << endl;

return;

}

cout << "Circular Queue elements: ";

int i = q.front;

while (true) {

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

if (i == q.rear) break;

i = (i + 1) % MAX;

}

cout << endl;

}

int main() {

CircularQueue q;

initQueue(q);

enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

enqueue(q, 40);

enqueue(q, 50);

enqueue(q, 60); // Overflow case

display(q);

dequeue(q);

dequeue(q);

enqueue(q, 60);

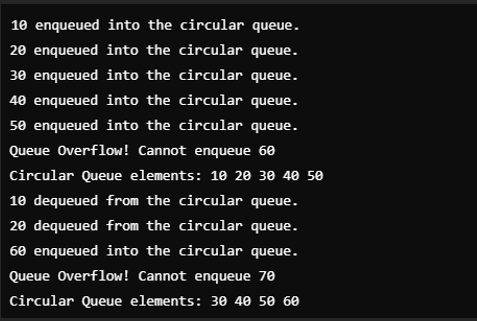
enqueue(q, 70); // Overflow case

display(q);

return 0;

}

**OUTPUT:**

****

**LAB 6:**

**Singly Linked List**

A singly linked list is a linear data structure where each element (called a node) contains two parts:

**Data:** The actual information being stored.

**Pointer (Next):** A reference to the next node in the sequence

**Program:**

#include <iostream>using namespace std;

struct Node {

int data;

Node \*next;

};

// Initialize the head pointerNode\* createNode(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = nullptr;

return newNode;

}

// Insert at the frontvoid insertFront(Node\* &head, int value) {

Node\* newNode = createNode(value);

newNode->next = head;

head = newNode;

cout << value << " inserted at the front." << endl;

}

// Insert at the middlevoid insertMid(Node\* &head, int value) {

if (!head || !head->next) {

insertFront(head, value);

return;

}

Node\* slow = head, \*fast = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

Node\* newNode = createNode(value);

newNode->next = slow->next;

slow->next = newNode;

cout << value << " inserted at the middle." << endl;

}

// Insert at the lastvoid insertLast(Node\* &head, int value) {

Node\* newNode = createNode(value);

if (!head) {

head = newNode;

cout << value << " inserted at the last." << endl;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

cout << value << " inserted at the last." << endl;

}

// Delete from the frontvoid deleteFront(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the front." << endl;

return;

}

Node\* temp = head;

head = head->next;

cout << temp->data << " deleted from the front." << endl;

delete temp;

}

// Delete from the middlevoid deleteMid(Node\* &head) {

if (!head || !head->next) {

deleteFront(head);

return;

}

Node\* slow = head, \*prev = nullptr, \*fast = head;

while (fast->next && fast->next->next) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

prev->next = slow->next;

cout << slow->data << " deleted from the middle." << endl;

delete slow;

}

// Delete from the lastvoid deleteLast(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the last." << endl;

return;

}

if (!head->next) {

cout << head->data << " deleted from the last." << endl;

delete head;

head = nullptr;

return;

}

Node\* temp = head;

while (temp->next->next) {

temp = temp->next;

}

cout << temp->next->data << " deleted from the last." << endl;

delete temp->next;

temp->next = nullptr;

}

// Search for an elementint search(Node\* head, int value) {

int index = 0;

while (head) {

if (head->data == value) {

return index;

}

head = head->next;

index++;

}

return -1;

}

// Update an element at a given indexvoid update(Node\* head, int index, int newValue) {

int currentIndex = 0;

while (head) {

if (currentIndex == index) {

cout << "Updated index " << index << " from " << head->data << " to " << newValue << "." << endl;

head->data = newValue;

return;

}

head = head->next;

currentIndex++;

}

cout << "Index " << index << " out of bounds!" << endl;

}

// Display the listvoid traverse(Node\* head) {

if (!head) {

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

return;

}

cout << "List elements: ";

while (head) {

cout << head->data << " ";

head = head->next;

}

cout << endl;

}

int main() {

Node\* head = nullptr;

// Insertion

insertFront(head, 10);

insertFront(head, 20);

insertLast(head, 30);

insertMid(head, 25);

traverse(head);

// Deletion

deleteFront(head);

traverse(head);

deleteMid(head);

traverse(head);

deleteLast(head);

traverse(head);

// Searching

int index = search(head, 10);

if (index != -1)

cout << "Element 10 found at index " << index << "." << endl;

else

cout << "Element 10 not found!" << endl;

// Updating

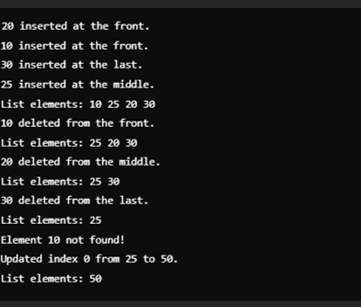
update(head, 0, 50);

traverse(head);

return 0;

}

**OUTPUT:**

****

D

**LAB 7:**

In computer science, a **doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

**Program:**

#include <iostream>using namespace std;

struct Node {

int data;

Node \*prev;

Node \*next;

};

// Initialize a new nodeNode\* createNode(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->prev = nullptr;

newNode->next = nullptr;

return newNode;

}

// Insert at the frontvoid insertFront(Node\* &head, int value) {

Node\* newNode = createNode(value);

if (head) {

newNode->next = head;

head->prev = newNode;

}

head = newNode;

cout << value << " inserted at the front." << endl;

}

// Insert at the lastvoid insertLast(Node\* &head, int value) {

Node\* newNode = createNode(value);

if (!head) {

head = newNode;

cout << value << " inserted at the last." << endl;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

cout << value << " inserted at the last." << endl;

}

// Insert at the middlevoid insertMid(Node\* &head, int value) {

if (!head || !head->next) {

insertFront(head, value);

return;

}

Node\* slow = head;

Node\* fast = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

Node\* newNode = createNode(value);

newNode->next = slow->next;

newNode->prev = slow;

if (slow->next) {

slow->next->prev = newNode;

}

slow->next = newNode;

cout << value << " inserted at the middle." << endl;

}

// Delete from the frontvoid deleteFront(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the front." << endl;

return;

}

Node\* temp = head;

head = head->next;

if (head) {

head->prev = nullptr;

}

cout << temp->data << " deleted from the front." << endl;

delete temp;

}

// Delete from the lastvoid deleteLast(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the last." << endl;

return;

}

if (!head->next) {

cout << head->data << " deleted from the last." << endl;

delete head;

head = nullptr;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->prev->next = nullptr;

cout << temp->data << " deleted from the last." << endl;

delete temp;

}

// Delete from the middlevoid deleteMid(Node\* &head) {

if (!head || !head->next) {

deleteFront(head);

return;

}

Node\* slow = head;

Node\* fast = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

if (slow->prev) {

slow->prev->next = slow->next;

}

if (slow->next) {

slow->next->prev = slow->prev;

}

cout << slow->data << " deleted from the middle." << endl;

delete slow;

}

// Search for an elementint search(Node\* head, int value) {

int index = 0;

while (head) {

if (head->data == value) {

return index;

}

head = head->next;

index++;

}

return -1;

}

// Update an element at a given indexvoid update(Node\* head, int index, int newValue) {

int currentIndex = 0;

while (head) {

if (currentIndex == index) {

cout << "Updated index " << index << " from " << head->data << " to " << newValue << "." << endl;

head->data = newValue;

return;

}

head = head->next;

currentIndex++;

}

cout << "Index " << index << " out of bounds!" << endl;

}

// Display the list forwardvoid traverseForward(Node\* head) {

if (!head) {

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

return;

}

cout << "List elements (forward): ";

while (head) {

cout << head->data << " ";

head = head->next;

}

cout << endl;

}

// Display the list backwardvoid traverseBackward(Node\* head) {

if (!head) {

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

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

cout << "List elements (backward): ";

while (temp) {

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

temp = temp->prev;

}

cout << endl;

}

int main() {

Node\* head = nullptr;

// Insertion

insertFront(head, 10);

insertFront(head, 20);

insertLast(head, 30);

insertMid(head, 25);

traverseForward(head);

// Deletion

deleteFront(head);

traverseForward(head);

deleteMid(head);

traverseForward(head);

deleteLast(head);

traverseForward(head);

// Searching

int index = search(head, 10);

if (index != -1)

cout << "Element 10 found at index " << index << "." << endl;

else

cout << "Element 10 not found!" << endl;

// Updating

update(head, 0, 50);

traverseForward(head);

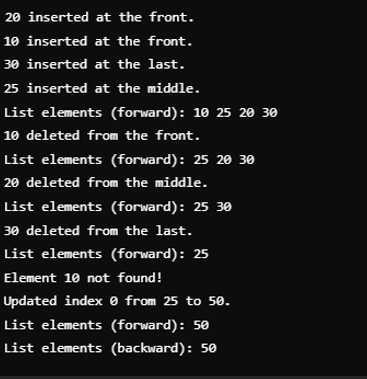
// Backward traversal

traverseBackward(head);

return 0;

}

**OUTPUT:**

****

D

**LAB 8:**

A **circular linked list** is a type of linked list where the last node's "next" pointer points back to the first node in the list. This creates a circular structure, where there is no beginning or end.

**Program:**

#include <iostream>using namespace std;

struct Node {

int data;

Node \*next;

};

// Initialize a new nodeNode\* createNode(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = nullptr;

return newNode;

}

// Insert at the frontvoid insertFront(Node\* &head, int value) {

Node\* newNode = createNode(value);

if (!head) {

head = newNode;

head->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

newNode->next = head;

temp->next = newNode;

head = newNode;

}

cout << value << " inserted at the front." << endl;

}

// Insert at the lastvoid insertLast(Node\* &head, int value) {

Node\* newNode = createNode(value);

if (!head) {

head = newNode;

head->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

cout << value << " inserted at the last." << endl;

}

// Insert at the middlevoid insertMid(Node\* &head, int value) {

if (!head || !head->next) {

insertFront(head, value);

return;

}

Node\* slow = head, \*fast = head;

while (fast->next != head && fast->next->next != head) {

slow = slow->next;

fast = fast->next->next;

}

Node\* newNode = createNode(value);

newNode->next = slow->next;

slow->next = newNode;

cout << value << " inserted at the middle." << endl;

}

// Delete from the frontvoid deleteFront(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the front." << endl;

return;

}

if (head->next == head) {

cout << head->data << " deleted from the front." << endl;

delete head;

head = nullptr;

} else {

Node\* temp = head;

Node\* last = head;

while (last->next != head) {

last = last->next;

}

head = head->next;

last->next = head;

cout << temp->data << " deleted from the front." << endl;

delete temp;

}

}

// Delete from the lastvoid deleteLast(Node\* &head) {

if (!head) {

cout << "List is empty! Cannot delete from the last." << endl;

return;

}

if (head->next == head) {

cout << head->data << " deleted from the last." << endl;

delete head;

head = nullptr;

} else {

Node\* temp = head;

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

temp = temp->next;

}

cout << temp->next->data << " deleted from the last." << endl;

delete temp->next;

temp->next = head;

}

}

// Delete from the middlevoid deleteMid(Node\* &head) {

if (!head || head->next == head) {

deleteFront(head);

return;

}

Node\* slow = head;

Node\* fast = head;

Node\* prev = nullptr;

while (fast->next != head && fast->next->next != head) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

prev->next = slow->next;

cout << slow->data << " deleted from the middle." << endl;

delete slow;

}

// Search for an elementint search(Node\* head, int value) {

if (!head) return -1;

Node\* temp = head;

int index = 0;

do {

if (temp->data == value) {

return index;

}

temp = temp->next;

index++;

} while (temp != head);

return -1;

}

// Update an element at a given indexvoid update(Node\* head, int index, int newValue) {

if (!head) {

cout << "List is empty! Cannot update." << endl;

return;

}

Node\* temp = head;

int currentIndex = 0;

do {

if (currentIndex == index) {

cout << "Updated index " << index << " from " << temp->data << " to " << newValue << "." << endl;

temp->data = newValue;

return;

}

temp = temp->next;

currentIndex++;

} while (temp != head);

cout << "Index " << index << " out of bounds!" << endl;

}

// Traverse the listvoid traverse(Node\* head) {

if (!head) {

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

return;

}

Node\* temp = head;

cout << "List elements: ";

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = nullptr;

// Insertion

insertFront(head, 10);

insertFront(head, 20);

insertLast(head, 30);

insertMid(head, 25);

traverse(head);

// Deletion

deleteFront(head);

traverse(head);

deleteMid(head);

traverse(head);

deleteLast(head);

traverse(head);

// Searching

int index = search(head, 10);

if (index != -1)

cout << "Element 10 found at index " << index << "." << endl;

else

cout << "Element 10 not found!" << endl;

// Updating

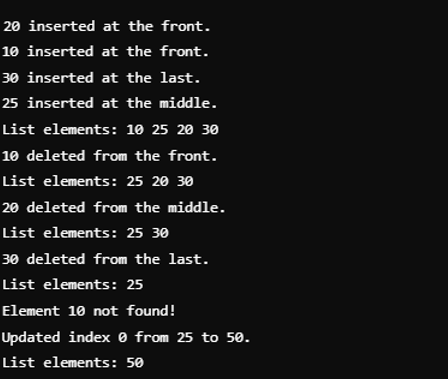
update(head, 0, 50);

traverse(head);

return 0;

}

**OUTPUT:**

****

**LAB 9:**

**Program:**

#include <iostream>using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

// Create a new nodeNode\* createNode(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->left = nullptr;

newNode->right = nullptr;

return newNode;

}

// Insert a node into the BSTNode\* insert(Node\* root, int value) {

if (!root) {

return createNode(value);

}

if (value < root->data) {

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

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

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

}

return root;

}

// Find the minimum value node in the BSTNode\* findMin(Node\* root) {

while (root && root->left) {

root = root->left;

}

return root;

}

// Delete a node from the BSTNode\* deleteNode(Node\* root, int value) {

if (!root) {

return root;

}

if (value < root->data) {

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

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

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

} else {

// Node with only one child or no child

if (!root->left) {

Node\* temp = root->right;

delete root;

return temp;

} else if (!root->right) {

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;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Search for a node in the BSTbool search(Node\* root, int value) {

if (!root) {

return false;

}

if (value == root->data) {

return true;

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

return search(root->left, value);

} else {

return search(root->right, value);

}

}

// In-order traversalvoid inOrder(Node\* root) {

if (root) {

inOrder(root->left);

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

inOrder(root->right);

}

}

// Pre-order traversalvoid preOrder(Node\* root) {

if (root) {

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

preOrder(root->left);

preOrder(root->right);

}

}

// Post-order traversalvoid postOrder(Node\* root) {

if (root) {

postOrder(root->left);

postOrder(root->right);

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

}

}

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 70);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 60);

root = insert(root, 80);

cout << "In-order traversal: ";

inOrder(root);

cout << endl;

cout << "Pre-order traversal: ";

preOrder(root);

cout << endl;

cout << "Post-order traversal: ";

postOrder(root);

cout << endl;

// Searching for a value

int searchValue = 40;

if (search(root, searchValue)) {

cout << "Value " << searchValue << " found in the BST." << endl;

} else {

cout << "Value " << searchValue << " not found in the BST." << endl;

}

// Deleting a node

int deleteValue = 50;

root = deleteNode(root, deleteValue);

cout << "After deleting " << deleteValue << ", in-order traversal: ";

inOrder(root);

cout << endl;

return 0;

}