

The University of Faisalabad

Name: Mohammad Mukedas

Roll no: 2023-bs-ai-008

Submitted to: Mam Irsha Qureshi

Degree: AI - SEC (A)



LAB 2:

Array: An array is a fundamental data structure that stores a collection of elements of the same data type in contiguous memory locations. Each element in an array is accessed using an index or key.

Program: Arrays (Insertion at Front, Mid, Last, Update, search)

```
#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:

```
Initial array: 10 20 30
After inserting 5 at front: 5 10 20 30
After inserting 15 at mid: 5 10 15 20 30
After inserting 35 at last: 5 10 15 20 30 35
After deleting from front: 10 15 20 30 35
After deleting from mid: 10 15 30 35
After deleting from last: 10 15 30
Element 20 not found!
After updating index 1 to 25: 10 25 30
```

LAB 3:

In computer science, 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: Stack Implementation in C++

This program demonstrates stack operations including push, pop, peek (top element), and checking if the stack is empty.

```
#include <iostream> using namespace std;
#define MAX 100
struct Stack {
    int arr[MAX];
    int top;
};
// Initialize stack void initStack(Stack &s) {
    s.top = -1;
}
// Check if the stack is empty bool isEmpty(Stack &s) {
    return s.top == -1;
}
// Check if the stack is full bool isFull(Stack &s) {
    return s.top == MAX - 1;
}
// Push an element onto the stack void 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 stack
int 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 stack
int peek(Stack &s) {
    if (isEmpty(s)) {
        cout << "Stack is empty. No top element." << endl;
        return -1;
    }
    return s.arr[s.top];
}

// Display the stack elements
void 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);
}

```

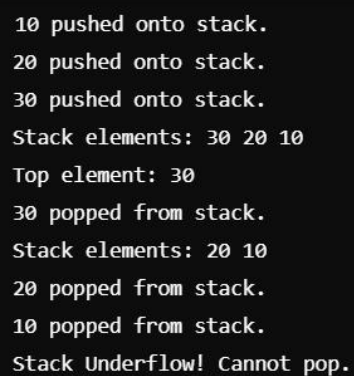
```
cout << "Top element: " << peek(s) << endl;

pop(s);
display(s);

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

return 0;
}
```

OUTPUT:

A terminal window with a dark background and light-colored text. It shows the output of a C++ program that uses a stack. The output includes pushing three numbers (10, 20, 30) onto the stack, displaying the stack elements (30 20 10), peeking at the top element (30), and then popping each element. The final line shows an attempt to pop from an empty stack, resulting in an underflow error.

```
10 pushed onto stack.
20 pushed onto stack.
30 pushed onto stack.
Stack elements: 30 20 10
Top element: 30
30 popped from stack.
Stack elements: 20 10
20 popped from stack.
10 popped from stack.
Stack Underflow! Cannot pop.
```


LAB 4:

Program: Stack Application - Infix to Postfix Conversion

This program converts an infix expression (e.g., A+B*C) to a postfix expression (e.g., ABC*+) using stack operations.

```
#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 stack
char peek(Stack &s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s.arr[s.top];
}

// Check if a character is an operator
bool isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
}

// Get precedence of an operator
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// Convert infix to postfix
void 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:

```
Enter an infix expression (e.g., A+B*C): A+B*C
```

```
Postfix expression: ABC*+
```

LAB 5:

Program: Queue (Linear and Circular)

This program demonstrates the implementation of both Linear Queue and Circular Queue.

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 full
bool isFull(LinearQueue &q) {
    return q.rear == MAX - 1;
}
// Enqueue operation
void 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 operation
int 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 queue
void 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:

```

10 enqueued into the queue.
20 enqueued into the queue.
30 enqueued into the queue.
40 enqueued into the queue.
50 enqueued into the queue.
Queue Overflow! Cannot enqueue 60
Queue elements: 10 20 30 40 50
10 dequeued from the queue.
20 dequeued from the queue.
Queue elements: 30 40 50

```

2. Circular Queue

```

#include <iostream>using namespace std;
#define MAX 5

```

```

struct CircularQueue {
    int arr[MAX];
    int front, rear;
};

// Initialize the queue
void initQueue(CircularQueue &q) {
    q.front = -1;
    q.rear = -1;
}

// Check if the queue is empty
bool isEmpty(CircularQueue &q) {
    return q.front == -1;
}

// Check if the queue is full
bool isFull(CircularQueue &q) {
    return (q.rear + 1) % MAX == q.front;
}

// Enqueue operation
void 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 operation
int 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:

```
10 enqueued into the circular queue.  
20 enqueued into the circular queue.  
30 enqueued into the circular queue.  
40 enqueued into the circular queue.  
50 enqueued into the circular queue.  
Queue Overflow! Cannot enqueue 60  
Circular Queue elements: 10 20 30 40 50  
10 dequeued from the circular queue.  
20 dequeued from the circular queue.  
60 enqueued into the circular queue.  
Queue Overflow! Cannot enqueue 70  
Circular Queue elements: 30 40 50 60
```

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: Single Linked List (Insertion, Deletion, Searching, Edit, Update, Find Index, Traversing)

This program demonstrates the functionalities of a single linked list in C++.

```
#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 front void 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 middle void 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 last void 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 element
int 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 index
void 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
void 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:

```

20 inserted at the front.
10 inserted at the front.
30 inserted at the last.
25 inserted at the middle.
List elements: 10 25 20 30
10 deleted from the front.
List elements: 25 20 30
20 deleted from the middle.
List elements: 25 30
30 deleted from the last.
List elements: 25
Element 10 not found!
Updated index 0 from 25 to 50.
List elements: 50

```

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: Double Linked List (Insertion, Deletion, Searching, Edit, Update, Find Index, Traversing)

This program demonstrates a double linked list implementation with various operations.

```
#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 middle void 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 front void 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 last
void 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 middle
void 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 element
int 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 index
void 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 forward
void 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 backward
void 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:

```

20 inserted at the front.
10 inserted at the front.
30 inserted at the last.
25 inserted at the middle.
List elements (forward): 10 25 20 30
10 deleted from the front.
List elements (forward): 25 20 30
20 deleted from the middle.
List elements (forward): 25 30
30 deleted from the last.
List elements (forward): 25
Element 10 not found!
Updated index 0 from 25 to 50.
List elements (forward): 50
List elements (backward): 50

```

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: Circular Linked List (Insertion, Deletion, Searching, Edit, Update, Find Index, Traversing)

This program demonstrates the implementation of a circular linked list with various operations in C++.

```
#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 last
void 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 middle
void 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 front
void 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 last
void 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 middle void 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 element int 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 index void 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 list
void 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:

```

20 inserted at the front.
10 inserted at the front.
30 inserted at the last.
25 inserted at the middle.
List elements: 10 25 20 30
10 deleted from the front.
List elements: 25 20 30
20 deleted from the middle.
List elements: 25 30
30 deleted from the last.
List elements: 25
Element 10 not found!
Updated index 0 from 25 to 50.
List elements: 50

```

LAB 9:

Binary Search Tree (BST) Definition

A binary search tree is a binary tree data structure where each node has at most two children (referred to as the left child and the right child)

Program: Binary Search Tree (Insertion, Deletion, Searching, and Traversal - In-order, Pre-order, Post-order)

This program implements a binary search tree (BST) and demonstrates its basic operations.

```
#include <iostream>using namespace std;  
struct Node {
```

```

    int data;
    Node* left;
    Node* right;
};

// Create a new node
Node* createNode(int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

// Insert a node into the BST
Node* 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 BST
Node* findMin(Node* root) {
    while (root && root->left) {
        root = root->left;
    }
    return root;
}

// Delete a node from the BST
Node* 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 BST
bool 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 traversal
void inOrder(Node* root) {
    if (root) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}

// Pre-order traversal
void preOrder(Node* root) {
    if (root) {

```

```

        cout << root->data << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Post-order traversal
void 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;
}

```

OUTPUT:

```

In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50
Value 40 found in the BST.
After deleting 50, in-order traversal: 20 30 40 60 70 80

```