**Submitted To:**

**Miss Irsha Qureshi**

**Submitted By:**

**Ayesha Imran**

**Registration no#:**

**2023-BS-AI-057**

**Degree Program:**

**BS - Artificial Intelligence**

**Semester:**

**03**

# LAB MANUAl


Lab 01-Arrays ........ 4
Implementation of Arrays ........ 4
What is an Array? ........ 4
Advantages of Arrays ........ 4
Types of Arrays ........ 4
Examples: ........ 4
1. Array Traversing ........ 4
2. Insertion (Front, Mid, Last) ........ 5
3. Deletion (Front, Mid, Last) ........ 8
4. Searching ........ 12
5. Update ........ 13
Lab 02- Stack ........ 14
Implementation of Stack ........ 14
What is a Stack? ........ 14
Advantages: ........ 14
Types of Stacks: ........ 14
1. Based on Implementation: ........ 14
2. Based on Use: ........ 14
Applications: ........ 14
Examples: ........ 15
1. Browser Navigation (Back/Forward History) ........ 15
2. Undo/Redo in Text Editor ........ 16
3. Function Call Stack (Recursion) ........ 18
4. Balancing Parentheses ........ 19
5. Infix to Postfix Conversion ........ 20
Lab 03 – Queue ........ 22
Implementation of Queue ........ 22
What is a Queue? ........ 22
Advantages of a Queue: ........ 22
Types of Queues: ........ 22
Applications: ........ 22
Examples: ........ 23
1. Linear Queue (Array-Based) ........ 23
2. Circular Queue (Array-Based) ........ 25
i = (i + 1) % size; ........ 26
Lab 04 - Single Linked List (SLL) ........ 28
Implementation of  Single Linked List (SLL) ........ 28
What is a Single Linked List (SLL)? ........ 28
Advantages of SLL: ........ 28
Real-Life Applications: ........ 28
Examples: ........ 28
1. Insertion at Front: ........ 28
2. Insertion at End: ........ 29
3. Insertion at Middle (after a specific position): ........ 31
4. Deletion from Front: ........ 32
5. Deletion from Last ........ 33
6.Deletion from Middle (Specific Position) ........ 34
7. Searching for an Element ........ 36

8. Updating an Element ........ 37
9. Finding Index of an Element ........ 38
10. Traversing the List ........ 39
11. Deleting the List ........ 40
Lab 05 - Double Linked List (DLL) ........ 42
Implementation of Double Linked List (DLL) ........ 42
What is Double Linked List (DLL)? ........ 42
1. Data ........ 42
Advantages: ........ 42
Disadvantages: ........ 42
Types: ........ 42
Real-life Uses: ........ 42
Example: ........ 43
1. Insertion at the Front ........ 43
2. Insertion at the Last ........ 44
3. Insertion at Middle (Specific Position) ........ 46
4. Deletion from the Front ........ 48
5. Deletion from the Last ........ 49
6. Deletion from the Middle (Specific Position) ........ 51
7. Searching ........ 54
8. Traversing the List ........ 55
9. Update a Node ........ 56
Lab 06 - Circular Linked List ........ 58
Implementation of Circular Linked List ........ 58
What is Circular Linked List? ........ 58
Advantages: ........ 58
Applications: ........ 58
Example: ........ 58
1. Insertion at Front ........ 58
2. Insertion at Last ........ 60
3.Insertion at a Specific Position ........ 61
4. Deletion from Front ........ 64
5.Deletion from Last ........ 66
6.Deletion from a Specific Position ........ 67
7.Searching ........ 70
8. Traversing ........ 71
9.Update ........ 73
Lab 07 - Binary Search Tree (BST) ........ 75
Implementation of Circular Linked List ........ 75
What is Binary Search Tree (BST)? ........ 75
Advantages: ........ 75
Types of BST: ........ 75
Examples: ........ 76
1. Insertion in Binary Search Tree ........ 76
2. Deletion in Binary Search Tree ........ 77
3. Searching in Binary Search Tree ........ 80
4.Traversal (In-order, Pre-order, Post-order) ........ 82
Main Program (Traversals) ........ 83

Lab 01-Arrays

# Implementation of Arrays

## What is an Array?

**An array is a collection of elements of the same type stored in contiguous memory locations.**

**It allows accessing elements using their index (starting from 0).**

## Advantages of Arrays

1. **Random Access**: Direct access to elements using index.
2. **Efficient Storage**: Compact memory usage.
3. **Easy Traversal**: Simple loops for processing elements.
4. **Sorting/Searching**: Works well with algorithms.
5. **Fixed Size**: Useful for known-size data.

## Types of Arrays

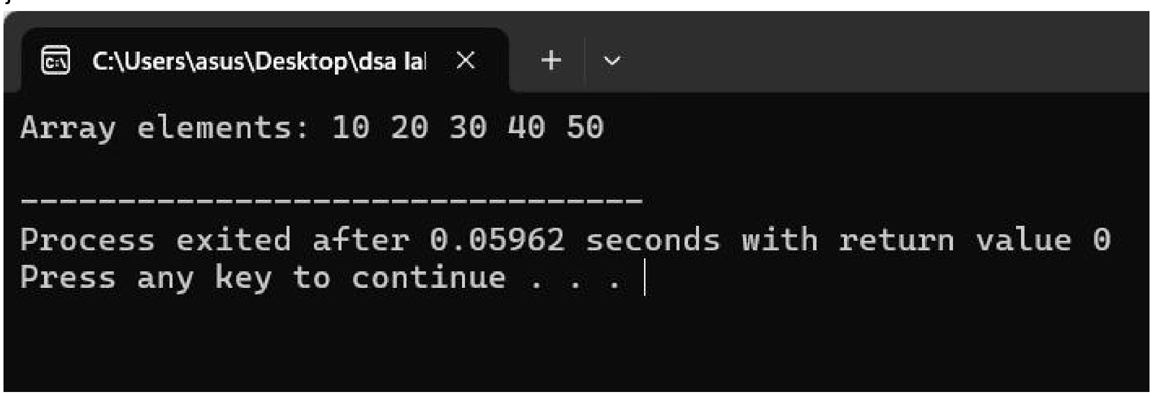
1. **One-Dimensional**: Linear collection (e.g., int arr[5];).
2. **Multi-Dimensional**: Arrays within arrays (e.g., int mat[3][3];).
3. **Dynamic Array**: Size adjusted at runtime (e.g., new int[n]; or std::vector<int>).

## Examples:

### 1. Array Traversing

```
#include <iostream>
using namespace std;

void traverseArray(int arr[], int size) {
```

```
    cout << "Array elements: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = 5;
    traverseArray(arr, size);
    return 0;
}
```

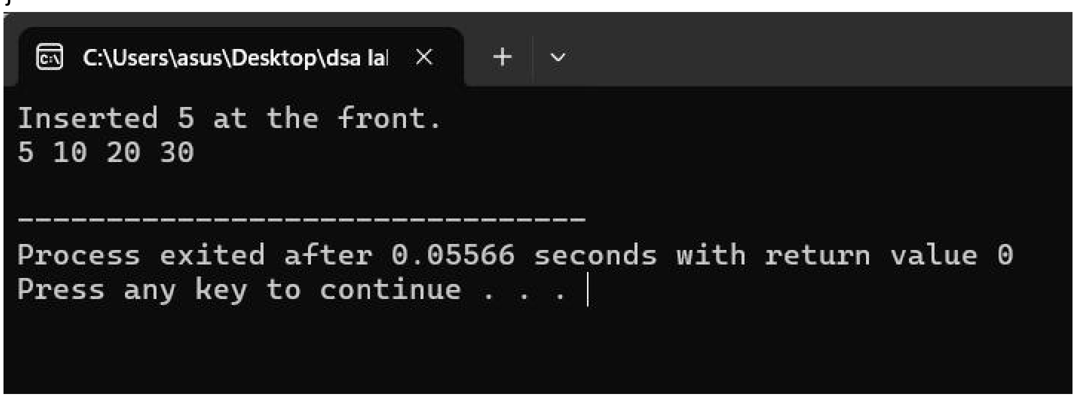



## 2. Insertion (Front, Mid, Last)

- **Insert at Front**

```
#include <iostream>
using namespace std;

void insertFront(int arr[], int &size, int element, int capacity) {
    if (size >= capacity) {
        cout << "Array is full. Cannot insert." << endl;
        return;
    }
    for (int i = size; i > 0; i--) {
        arr[i] = arr[i - 1];
    }
```

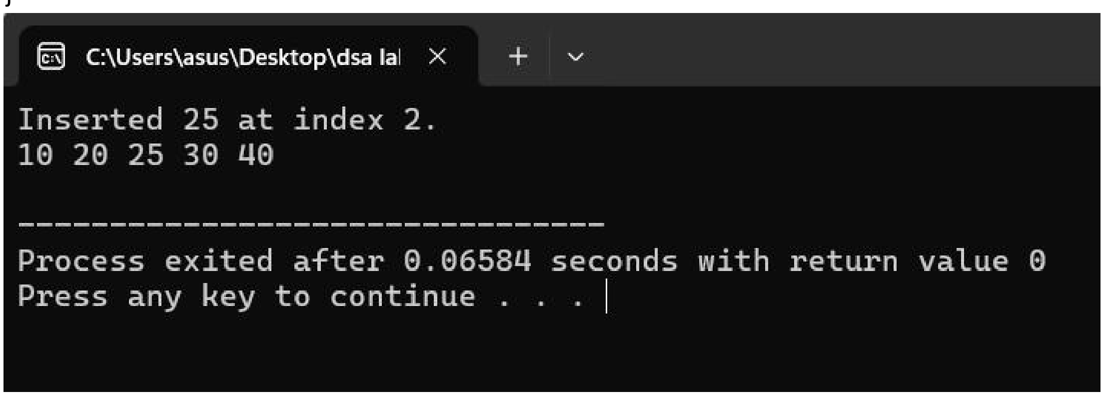
```
    arr[0] = element;
    size++;
    cout << "Inserted " << element << " at the front." << endl;
}

int main() {
    const int capacity = 10;
    int arr[capacity] = {10, 20, 30};
    int size = 3;

    insertFront(arr, size, 5, capacity);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```




- **Insert in Middle**

```
#include <iostream>
using namespace std;

void insertMid(int arr[], int &size, int element, int index, int capacity) {
    if (size >= capacity) {
        cout << "Array is full. Cannot insert." << endl;
        return;
    }
```

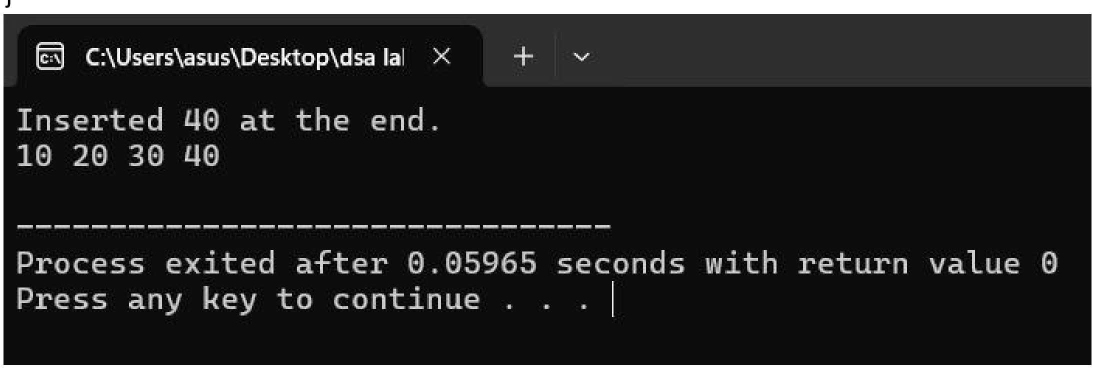
```
    for (int i = size; i > index; i--) {
        arr[i] = arr[i - 1];
    }
    arr[index] = element;
    size++;
    cout << "Inserted " << element << " at index " << index << "." << endl;
}

int main() {
    const int capacity = 10;
    int arr[capacity] = {10, 20, 30, 40};
    int size = 4;

    insertMid(arr, size, 25, 2, capacity);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```




- **Insert at Last**

```
#include <iostream>
using namespace std;

void insertLast(int arr[], int &size, int element, int capacity) {
    if (size >= capacity) {
```

```
        cout << "Array is full. Cannot insert." << endl;
        return;
    }
    arr[size++] = element;
    cout << "Inserted " << element << " at the end." << endl;
}

int main() {
    const int capacity = 10;
    int arr[capacity] = {10, 20, 30};
    int size = 3;

    insertLast(arr, size, 40, capacity);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

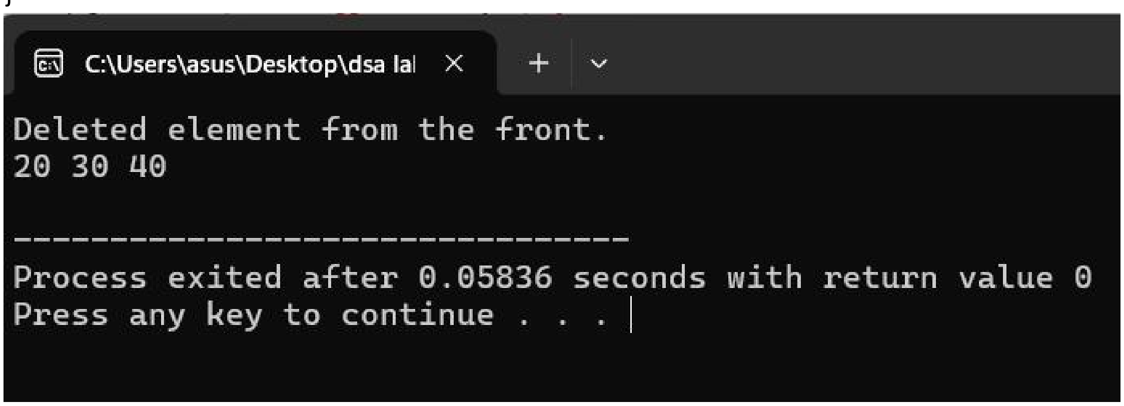



## 3. Deletion (Front, Mid, Last)

- **Delete from Front**

```
#include <iostream>
using namespace std;
```

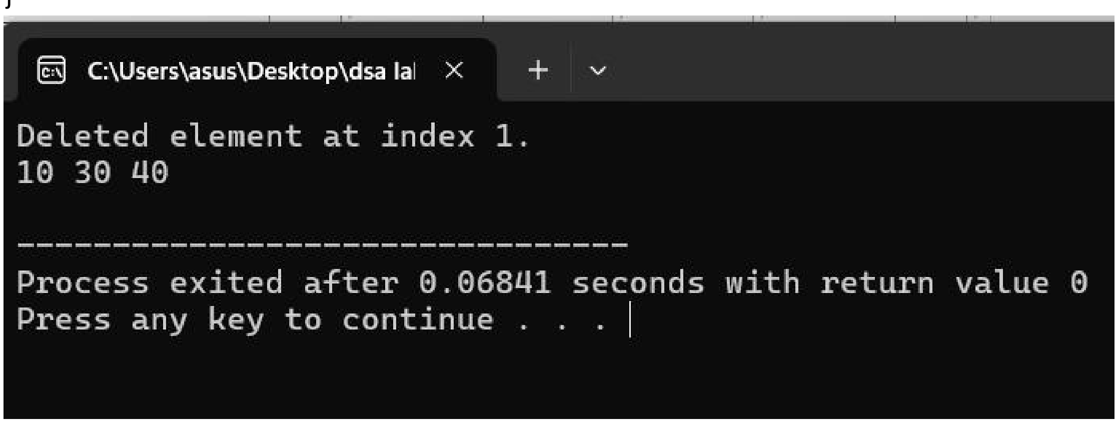
```
void deleteFront(int arr[], int &size) {
    if (size <= 0) {
        cout << "Array is empty. Cannot delete." << endl;
        return;
    }
    for (int i = 0; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
    cout << "Deleted element from the front." << endl;
}

int main() {
    int arr[] = {10, 20, 30, 40};
    int size = 4;

    deleteFront(arr, size);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```




- **Delete from Middle**

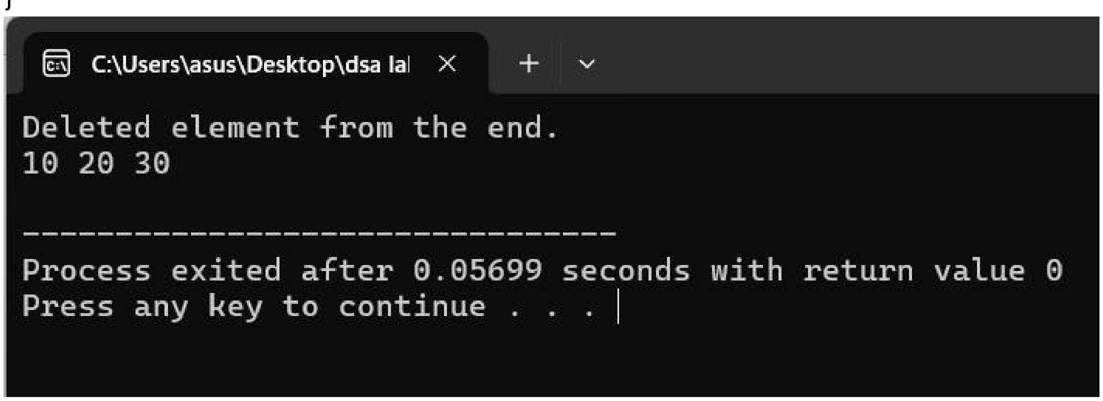
```
#include <iostream>
using namespace std;
```

```
void deleteMid(int arr[], int &size, int index) {
    if (index < 0 || index >= size) {
        cout << "Invalid index." << endl;
        return;
    }
    for (int i = index; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
    cout << "Deleted element at index " << index << "." << endl;
}

int main() {
    int arr[] = {10, 20, 30, 40};
    int size = 4;

    deleteMid(arr, size, 1);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```



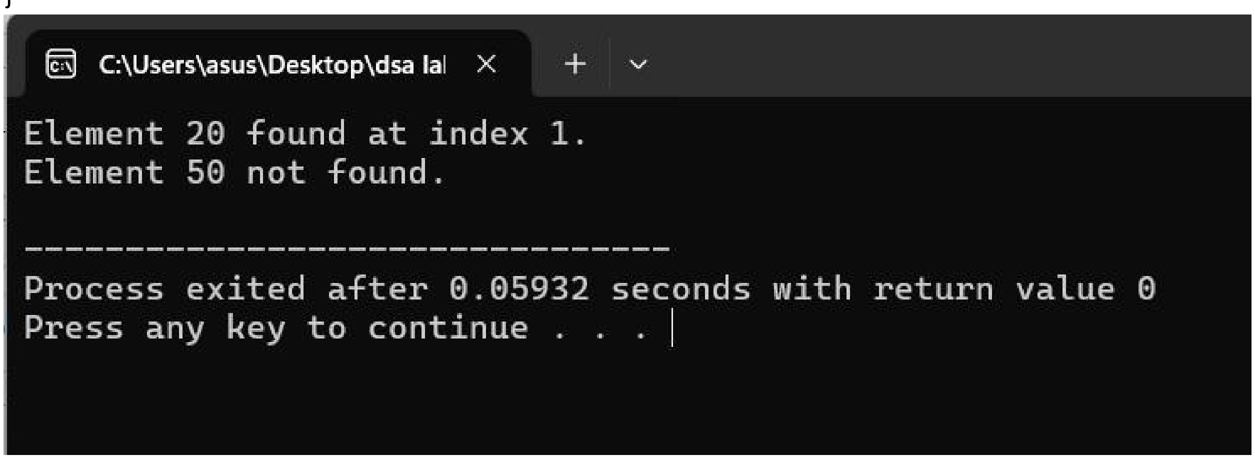
- **Delete from Last**

```
#include <iostream>
using namespace std;

void deleteLast(int arr[], int &size) {
    if (size <= 0) {
        cout << "Array is empty. Cannot delete." << endl;
        return;
    }
    size--;
    cout << "Deleted element from the end." << endl;
}

int main() {
    int arr[] = {10, 20, 30, 40};
    int size = 4;

    deleteLast(arr, size);

    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```



## 4. Searching

```cpp
#include <iostream>
using namespace std;

void searchElement(int arr[], int size, int element) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == element) {
            cout << "Element " << element << " found at index " << i << "." << endl;
            return;
        }
    }
    cout << "Element " << element << " not found." << endl;
}

int main() {
    int arr[] = {10, 20, 30, 40};
    int size = 4;

    searchElement(arr, size, 20);
    searchElement(arr, size, 50);

    return 0;
}
```



## 5. Update

```cpp
#include <iostream>
using namespace std;

void updateElement(int arr[], int size, int index, int newValue) {
   if (index < 0 || index >= size) {
      cout << "Invalid index." << endl;
      return;
   }
   arr[index] = newValue;
   cout << "Updated index " << index << " to " << newValue << "." << endl;
}

int main() {
   int arr[] = {10, 20, 30, 40};
   int size = 4;

   updateElement(arr, size, 2, 35);

   for (int i = 0; i < size; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;

   return 0;
}
```

```
C:\Users\asus\Desktop\dsa lal

Updated index 2 to 35.
10 20 35 40

--------------------------------
Process exited after 0.0768 seconds with return value 0
Press any key to continue . . .
```

Lab 02- Stack

## Implementation of Stack

## What is a Stack?

- A stack is a **LIFO (Last In, First Out)** data structure.
- The last item added is the first to be removed.
- Commn operations: **Push** (add), **Pop** (remove), and **Peek** (view top element).

## Advantages:

1. **Fast Operations**: Push and pop are O(1)O(1)O(1).
2. **Reversing Data**: Helpful in reversing strings or numbers.
3. **Memory Management**: Used in function calls and recursion.
4. **Simplifies Algorithms**: Balancing parentheses, infix-to-postfix conversion, undo/redo, etc.

## Types of Stacks:

1. **Based on Implementation**:
   - **Array-Based**: Fixed size, simple.
   - **Linked List-Based**: Dynamic size.
2. **Based on Use**:
   - **Call Stack**: Handles function calls.
   - **Undo Stack**: For undo/redo operations.
   - **Expression Stack**: Evaluates mathematical expressions.

## Applications:

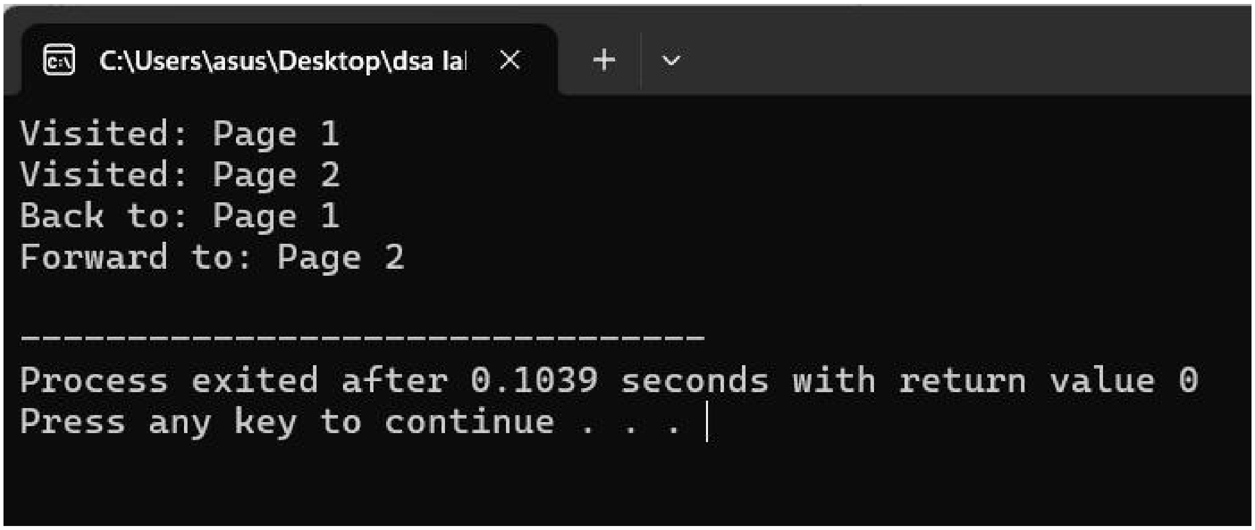
- Backtracking (e.g., solving mazes).
- Browser history navigation.
- Managing function calls (recursion).
- Expression evaluation (infix to postfix).

## Examples:

## 1. Browser Navigation (Back/Forward History)

```
#include <iostream>
#include <stack>
using namespace std;

class BrowserHistory {
    stack<string> backStack, forwardStack;

public:
    void visitPage(string page) {
        backStack.push(page);
        while (!forwardStack.empty()) forwardStack.pop();  // Clear forward stack
        cout << "Visited: " << page << endl;
    }

    void back() {
        if (backStack.empty()) {
            cout << "No pages in history!" << endl;
            return;
        }
        forwardStack.push(backStack.top());
        backStack.pop();
        cout << "Back to: " << (backStack.empty() ? "No page" : backStack.top()) << endl;
    }

    void forward() {
        if (forwardStack.empty()) {
            cout << "No forward page!" << endl;
            return;
        }
        backStack.push(forwardStack.top());
        forwardStack.pop();
        cout << "Forward to: " << backStack.top() << endl;
    }
};
```

```
int main() {
   BrowserHistory browser;
   browser.visitPage("Page 1");
   browser.visitPage("Page 2");
   browser.back();
   browser.forward();
   return 0;
}
```




## 2. Undo/Redo in Text Editor

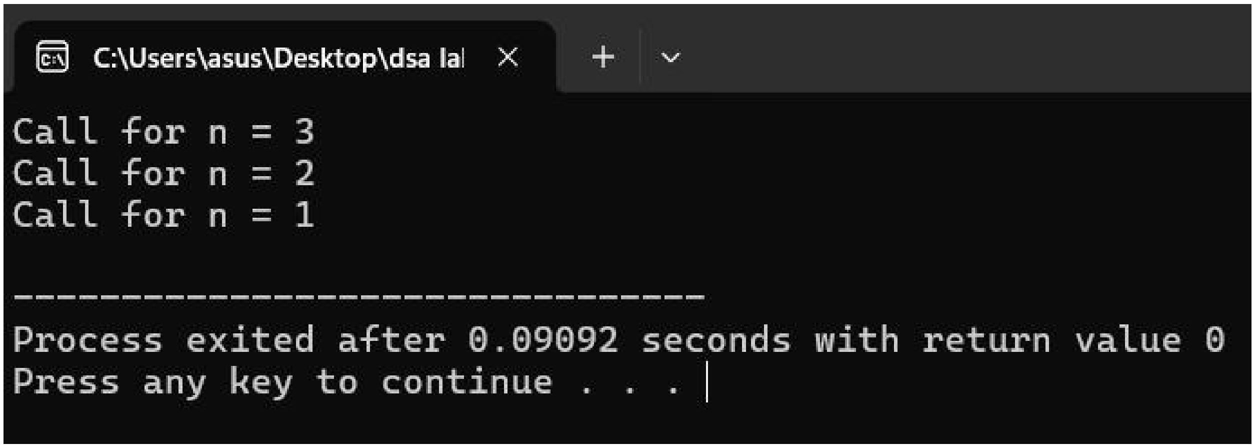
```
#include <iostream>
#include <stack>
using namespace std;

class TextEditor {
   stack<string> undoStack, redoStack;

public:
   void type(string text) {
      undoStack.push(text);
      while (!redoStack.empty()) redoStack.pop();
      cout << "Typed: " << text << endl;
   }

   void undo() {
```

```
        if (undoStack.empty()) {
            cout << "Nothing to undo!" << endl;
            return;
        }
        string lastText = undoStack.top();
        undoStack.pop();
        redoStack.push(lastText);
        cout << "Undid: " << lastText << endl;
    }

    void redo() {
        if (redoStack.empty()) {
            cout << "Nothing to redo!" << endl;
            return;
        }
        string lastText = redoStack.top();
        redoStack.pop();
        undoStack.push(lastText);
        cout << "Redid: " << lastText << endl;
    }
};

int main() {
    TextEditor editor;
    editor.type("Hello");
    editor.type("World");
    editor.undo();
    editor.redo();
    return 0;
}
```

```
Typed: Hello
Typed: World
Undid: World
Redid: World

--------------------------------
Process exited after 0.09321 seconds with return value 0
Press any key to continue . . .
```

## 3. Function Call Stack (Recursion)

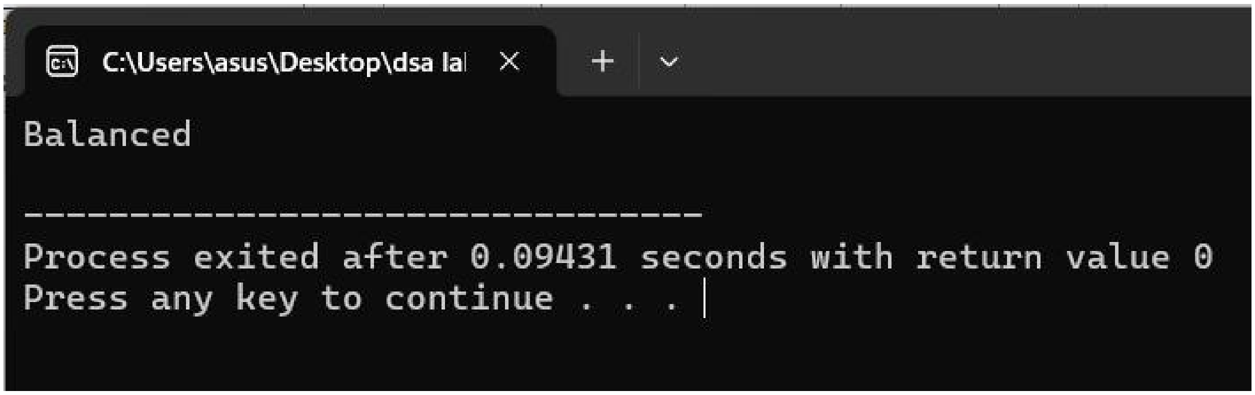
```cpp
#include <iostream>
#include <stack>
using namespace std;

void recursiveFunction(int n) {
    stack<int> callStack;
    callStack.push(n);
    if (n > 0) {
        cout << "Call for n = " << n << endl;
        recursiveFunction(n - 1);
    }
    callStack.pop();
}

int main() {
    recursiveFunction(3);
    return 0;
}
```




## 4. Balancing Parentheses

```
#include <iostream>
#include <stack>
using namespace std;

bool isBalanced(string expression) {
   stack<char> s;
   for (char c : expression) {
      if (c == '(' || c == '{' || c == '[') {
         s.push(c);
      } else if (c == ')' || c == '}' || c == ']') {
         if (s.empty()) return false;
         char top = s.top();
         if ((c == ')' && top == '(') || (c == '}' && top == '{') || (c == ']' && top == '[')) {
            s.pop();
         } else {
            return false;
         }
      }
   }
   return s.empty();
}

int main() {
   string expression = "({[()]})";
   cout << (isBalanced(expression) ? "Balanced" : "Not Balanced") << endl;
```

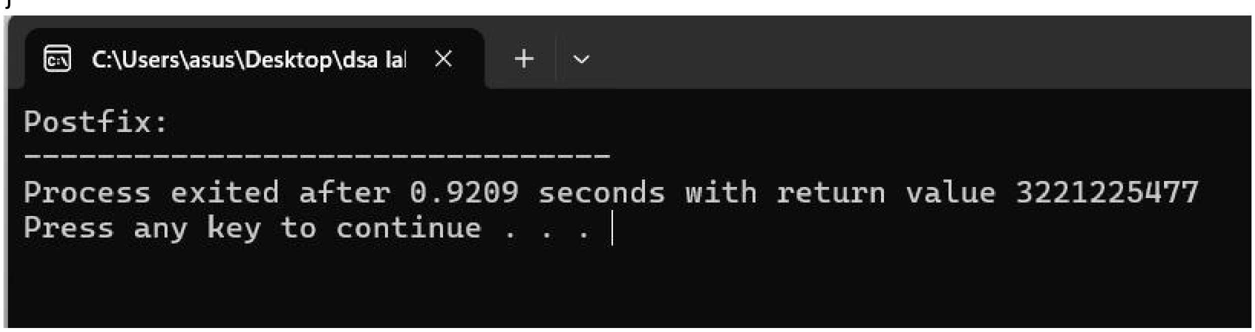
```
    return 0;
}
```




## 5. Infix to Postfix Conversion

```
#include <iostream>
#include <stack>
#include <cctype>
using namespace std;

int precedence(char c) {
    if (c == '+' || c == '-') return 1;
    if (c == '*' || c == '/') return 2;
    return 0;
}

string infixToPostfix(string infix) {
    stack<char> s;
    string postfix = "";

    for (char c : infix) {
        if (isalnum(c)) {
            postfix += c;
        } else if (c == '(') {
            s.push(c);
        } else if (c == ')') {
            while (!s.empty() && s.top() != '(') {
                postfix += s.top();
                s.pop();
```

```
        }
        s.pop();  // Remove '('
      } else {
        while (!s.empty() && precedence(s.top()) >= precedence(c)) {
          postfix += s.top();
          s.pop();
        }
        s.push(c);
      }
    }

    while (!s.empty()) {
      postfix += s.top();
      s.pop();
    }
    return postfix;
}

int main() {
    string infix = "A+B*(C^D-E)^(F+G*H)-I";
    cout << "Postfix: " << infixToPostfix(infix) << endl;
    return 0;
}
```



Lab 03 – Queue

## Implementation of Queue

## What is a Queue?

- A **queue** is a **FIFO (First In, First Out)** data structure.
- The first element added is the first one to be removed.
- Common operations: **Enqueue** (add), **Dequeue** (remove), **Front** (view front element), **Rear** (view last element).

## Advantages of a Queue:

1. **FIFO Access**: Processes elements in the order they arrive.
2. **Efficient Memory Use**: Manages resources effectively.
3. **Real-Time Systems**: Used for task scheduling and managing requests.
4. **Broad Applicability**: Useful in many scenarios like traffic management, task execution, and CPU scheduling.

## Types of Queues:

1. **Linear Queue**: Basic FIFO queue.
2. **Circular Queue**: Wraps around when it reaches the end, optimizing space.
3. **Priority Queue**: Elements are processed based on priority, not arrival order.
4. **Deque**: Double-ended queue, allows adding/removing from both ends.
5. **Queue using Two Stacks**: A queue simulated using two stacks.

## Applications:

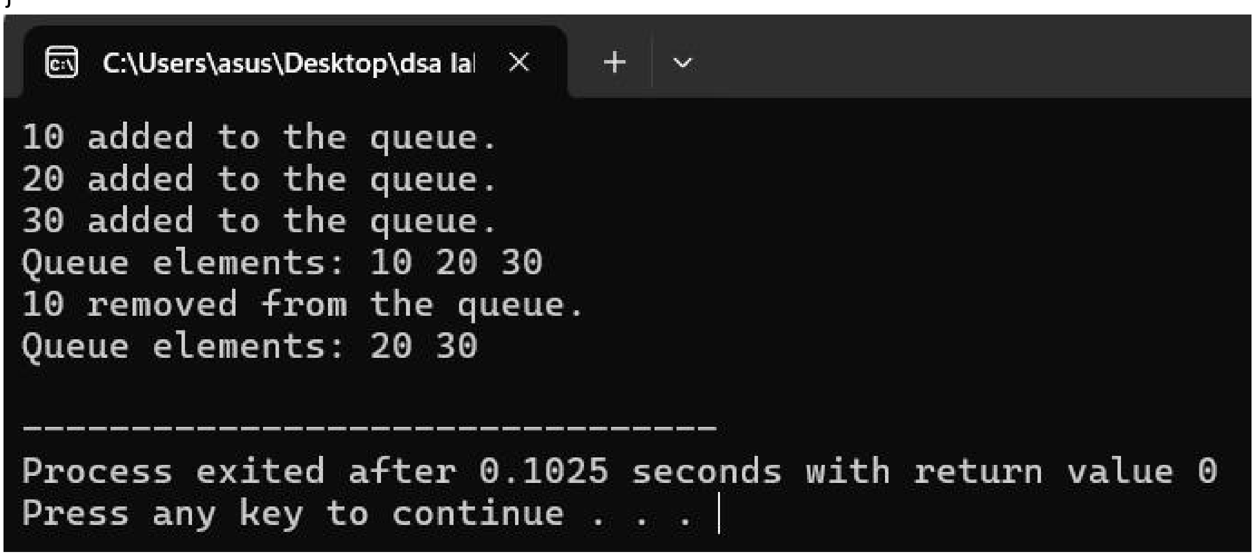
- Task scheduling
- Print jobs management
- Call centers handling
- Traffic signals
- Graph traversal (BFS)

## Examples:

### 1. Linear Queue (Array-Based)

```
#include <iostream>
using namespace std;

class LinearQueue {
    int* queue;
    int front, rear, size;

public:
    LinearQueue(int s) {
        size = s;
        queue = new int[size];
        front = rear = -1;
    }

    // Enqueue
    void enqueue(int value) {
        if (rear == size - 1) {
            cout << "Queue is full!" << endl;
        } else {
            if (front == -1) front = 0;
            queue[++rear] = value;
            cout << value << " added to the queue." << endl;
        }
    }

    // Dequeue
    void dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue is empty!" << endl;
        } else {
            cout << queue[front++] << " removed from the queue." << endl;
        }
    }

    // Display Queue
```

```
  void display() {
    if (front == -1 || front > rear) {
      cout << "Queue is empty!" << endl;
    } else {
      cout << "Queue elements: ";
      for (int i = front; i <= rear; ++i) {
        cout << queue[i] << " ";
      }
      cout << endl;
    }
  }
};

int main() {
  LinearQueue q(5);

  q.enqueue(10);
  q.enqueue(20);
  q.enqueue(30);
  q.display();

  q.dequeue();
  q.display();

  return 0;
}
```



## 2. Circular Queue (Array-Based)

```
#include <iostream>
using namespace std;

class CircularQueue {
    int* queue;
    int front, rear, size;

public:
    CircularQueue(int s) {
        size = s;
        queue = new int[size];
        front = rear = -1;
    }

    // Enqueue
    void enqueue(int value) {
        if ((rear + 1) % size == front) {
            cout << "Queue is full!" << endl;
        } else {
            if (front == -1) front = 0;
            rear = (rear + 1) % size;
            queue[rear] = value;
            cout << value << " added to the queue." << endl;
        }
    }

    // Dequeue
    void dequeue() {
        if (front == -1) {
            cout << "Queue is empty!" << endl;
        } else {
            cout << queue[front] << " removed from the queue." << endl;
            if (front == rear) {
                front = rear = -1;  // Reset the queue after last element is dequeued
            } else {
                front = (front + 1) % size;
            }
        }
```

```
    }

    // Display Queue
    void display() {
        if (front == -1) {
            cout << "Queue is empty!" << endl;
        } else {
            cout << "Queue elements: ";
            int i = front;
            while (i != rear) {
                cout << queue[i] << " ";
                i = (i + 1) % size;
            }
            cout << queue[rear] << endl;
        }
    }
};

int main() {
    CircularQueue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.display();

    q.enqueue(40);
    q.enqueue(50);
    q.display();

    return 0;
}
```

```
C:\Users\asus\Desktop\dsa lal

10 added to the queue.
20 added to the queue.
30 added to the queue.
Queue elements: 10 20 30
10 removed from the queue.
Queue elements: 20 30
40 added to the queue.
50 added to the queue.
Queue elements: 20 30 40 50

--------------------------------
Process exited after 0.09171 seconds with return value 0
Press any key to continue . . .
```

Lab 04 - Single Linked List (SLL)

## Implementation of Single Linked List (SLL)

## What is a Single Linked List (SLL)?

A **Single Linked List** is a linear data structure where each node contains:

- **Data**: The information stored in the node.
- **Next**: A reference to the next node. The last node points to nullptr.

## Advantages of SLL:

1. **Dynamic Size**: Grows or shrinks as needed.
2. **Efficient Insertions/Deletions**: Especially at the beginning.
3. **Memory Efficient**: Allocates memory as needed.
4. **Flexible Memory Management**: Can use non-contiguous memory.

## Real-Life Applications:

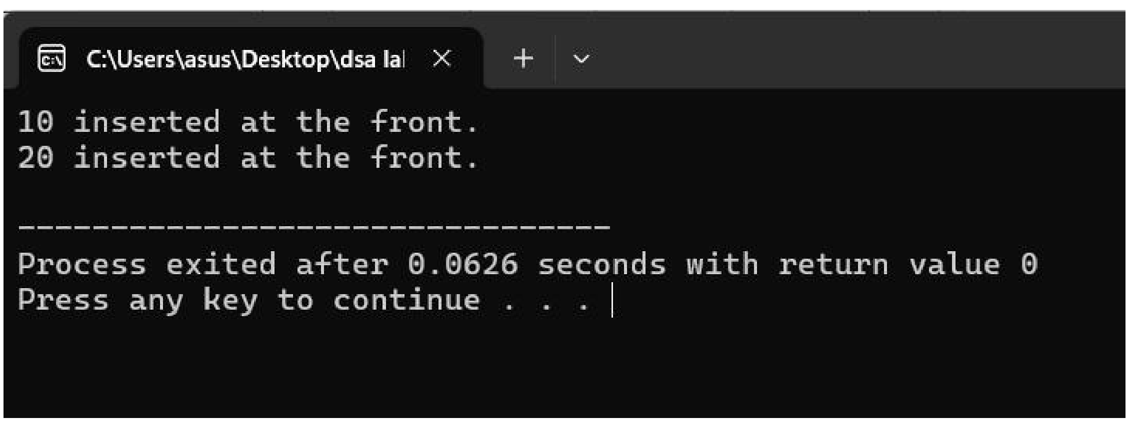
1. **Dynamic Memory Allocation**: Memory management in operating systems.
2. **Stacks and Queues**: Implemented using linked lists.
3. **Browser History**: Web browsers use it to store visited pages.
4. **Polynomial Representation**: Representing polynomials in math.
5. **Graph Representation**: Used for adjacency lists in graphs.
6. **Playlist Management**: Music players use it to manage playlists.

## Examples:

### 1. Insertion at Front:

```
#include <iostream>
using namespace std;
```

```
struct Node {
   int data;
   Node* next;
};

void insertFront(Node*& head, int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = head;
   head = newNode;
   cout << value << " inserted at the front." << endl;
}

int main() {
   Node* head = nullptr;
   insertFront(head, 10);
   insertFront(head, 20);
   return 0;
}
```

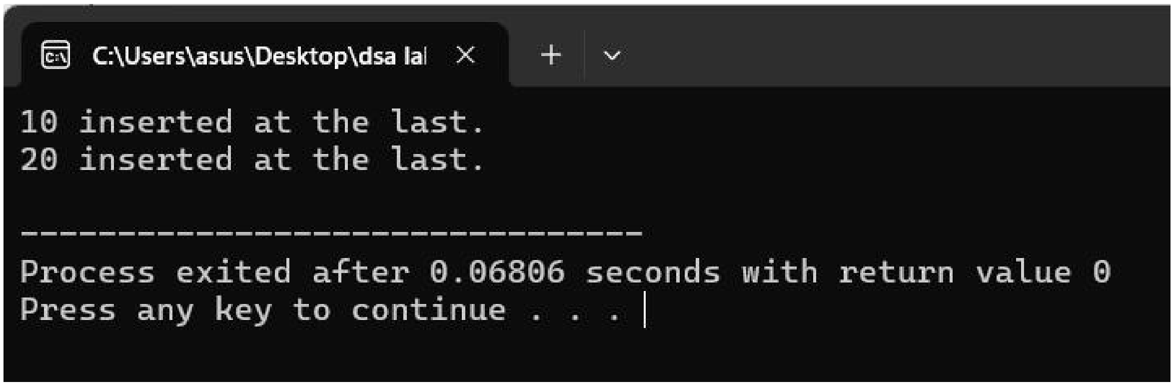



## 2. Insertion at End:

```
#include <iostream>

using namespace std;
```

```
struct Node {
    int data;
    Node* next;
};

void insertLast(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        cout << value << " inserted at the last." << endl;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    cout << value << " inserted at the last." << endl;
}

int main() {
    Node* head = nullptr;
    insertLast(head, 10);
```

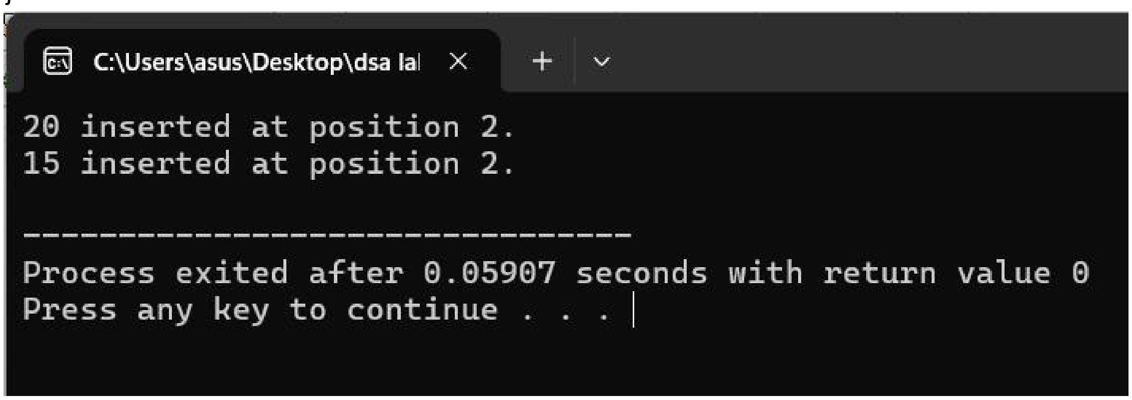
```
    insertLast(head, 20);

    return 0;

}
```




## 3. Insertion at Middle (after a specific position):

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void insertMid(Node*& head, int value, int position) {
    if (position == 1) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* newNode = new Node();
    newNode->data = value;
    Node* temp = head;

    for (int i = 1; i < position - 1 && temp != nullptr; i++) {
        temp = temp->next;
    }
```

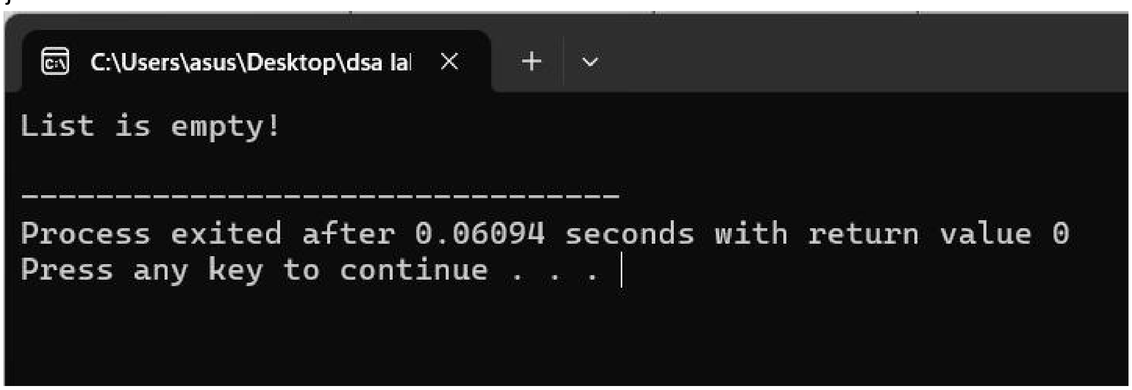
```
    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
    cout << value << " inserted at position " << position << "." << endl;
}

int main() {
    Node* head = nullptr;
    insertMid(head, 10, 1); // Insert at position 1
    insertMid(head, 20, 2); // Insert at position 2
    insertMid(head, 15, 2); // Insert at position 2
    return 0;
}
```




## 4. Deletion from Front:

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};
```

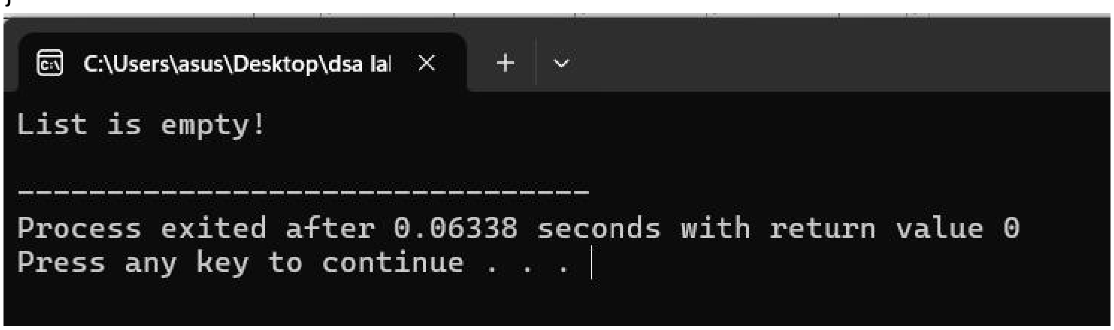
```
void deleteFront(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    head = head->next;
    delete temp;
    cout << "Node deleted from the front." << endl;
}

int main() {
    Node* head = nullptr;
    deleteFront(head); // Testing on empty list
    return 0;
}
```




## 5. Deletion from Last

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void deleteLast(Node*& head) {
    if (head == nullptr) {
```

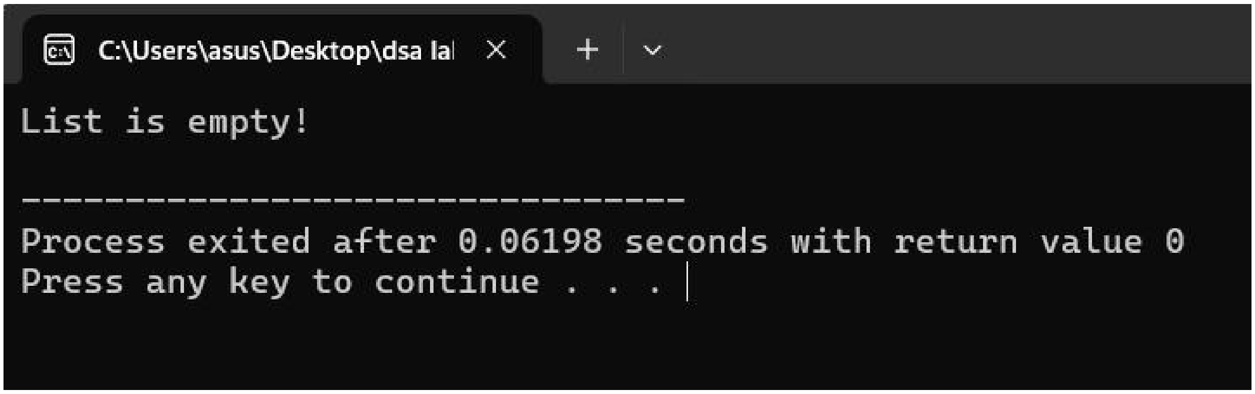
```
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    if (temp->next == nullptr) {
        head = nullptr;
        delete temp;
        cout << "Node deleted from the last." << endl;
        return;
    }

    while (temp->next != nullptr && temp->next->next != nullptr) {
        temp = temp->next;
    }
    delete temp->next;
    temp->next = nullptr;
    cout << "Node deleted from the last." << endl;
}

int main() {
    Node* head = nullptr;
    deleteLast(head); // Testing on empty list
    return 0;
}
```




## 6.Deletion from Middle (Specific Position)

```
#include <iostream>
using namespace std;
```

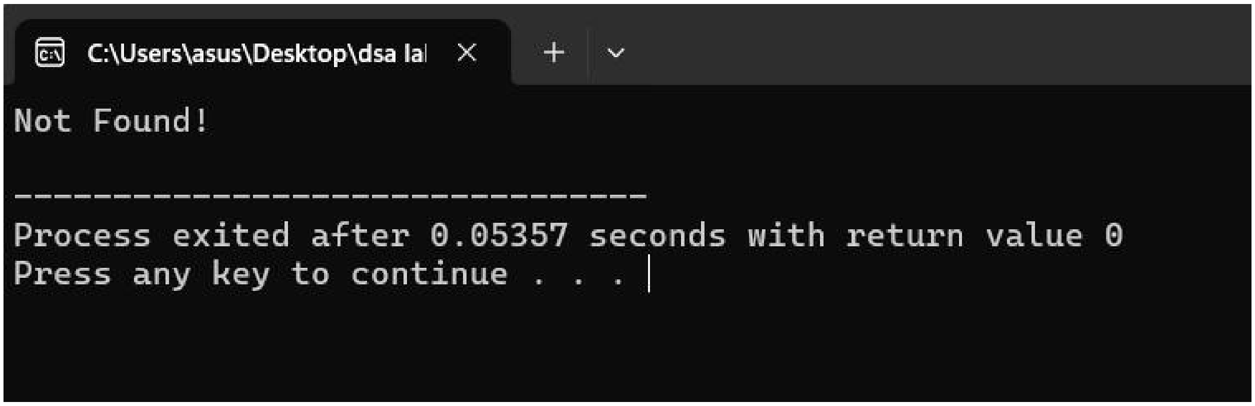
```cpp
struct Node {
    int data;
    Node* next;
};

void deleteMid(Node*& head, int position) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (position == 1) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr || temp->next == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }

    Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    delete nodeToDelete;
    cout << "Node deleted from position " << position << "." << endl;
}

int main() {
    Node* head = nullptr;
    deleteMid(head, 2); // Testing on empty list
    return 0;
}
```




## 7. Searching for an Element

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

bool search(Node* head, int value) {
    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == value) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

int main() {
    Node* head = nullptr;
    if (search(head, 10)) {
        cout << "Found!" << endl;
    } else {
        cout << "Not Found!" << endl;
    }
    return 0;
```

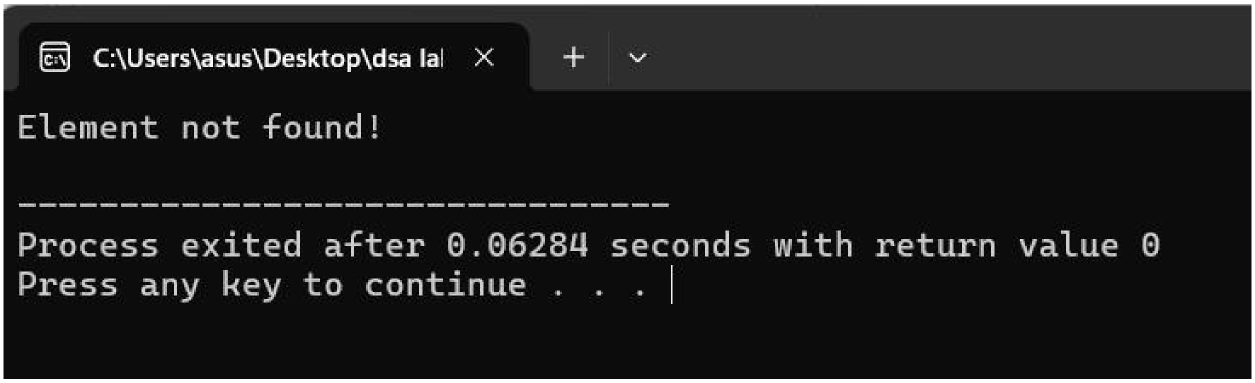
```
}
```




## 8. Updating an Element

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* next;
};

void update(Node* head, int oldValue, int newValue) {
   Node* temp = head;
   while (temp != nullptr) {
      if (temp->data == oldValue) {
         temp->data = newValue;
         cout << "Node with value " << oldValue << " updated to " << newValue << "." <<
endl;
         return;
      }
      temp = temp->next;
   }
   cout << "Element not found!" << endl;
}

int main() {
   Node* head = nullptr;
   update(head, 10, 20); // Testing on empty list
```

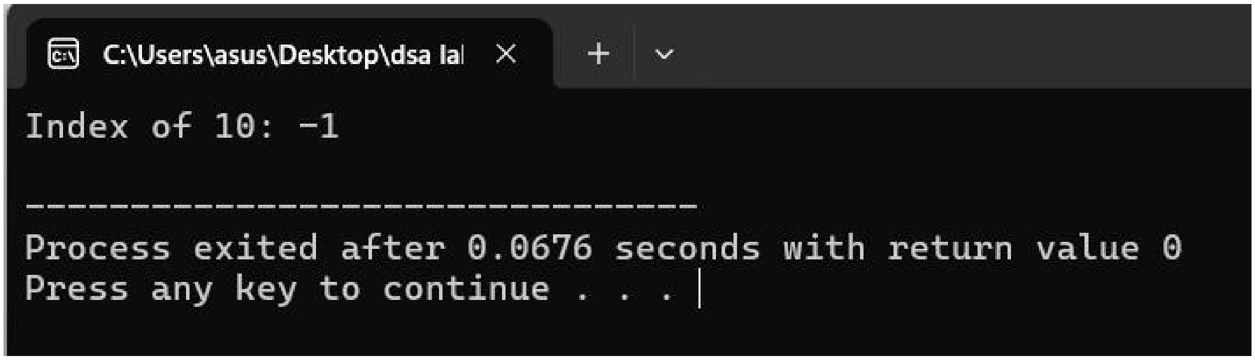
```
    return 0;
}
```




## 9. Finding Index of an Element

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

int findIndex(Node* head, int value) {
    Node* temp = head;
    int index = 0;

    while (temp != nullptr) {
        if (temp->data == value) {
            return index;
        }
        temp = temp->next;
        index++;
    }

    return -1; // Not found
}

int main() {
```

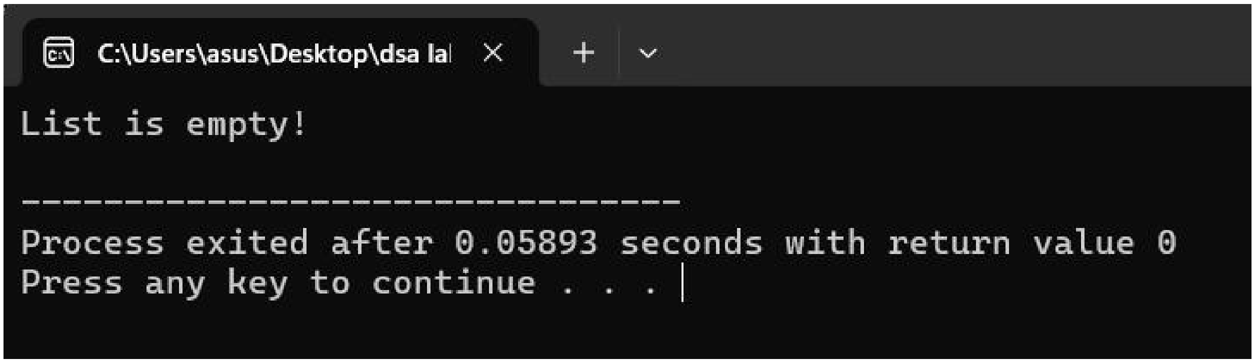
```
    Node* head = nullptr;
    cout << "Index of 10: " << findIndex(head, 10) << endl; // Testing on empty list
    return 0;
}
```




## 10. Traversing the List

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void traverse(Node* head) {
    Node* temp = head;
    if (temp == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    cout << "List elements: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
```

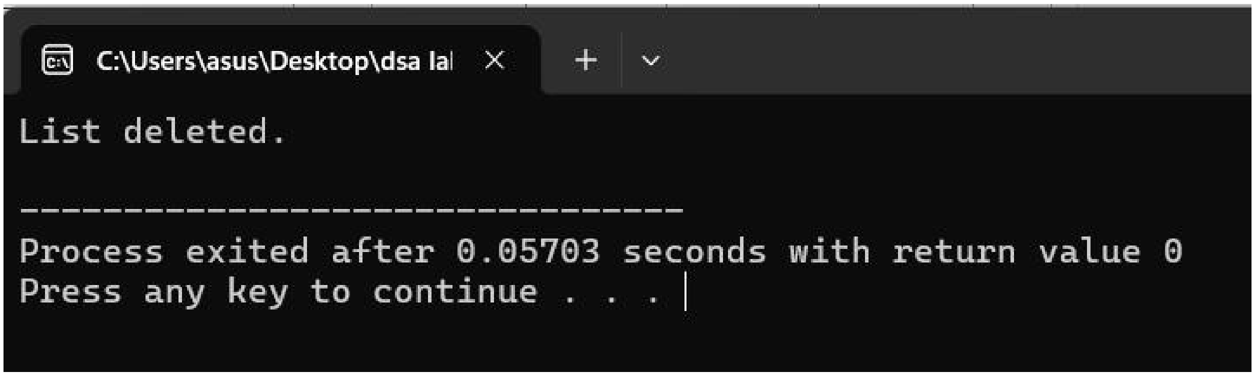
```
    Node* head = nullptr;
    traverse(head); // Testing on empty list
    return 0;
}
```




## 11. Deleting the List

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void deleteList(Node*& head) {
    Node* temp;
    while (head != nullptr) {
        temp = head;
        head = head->next;
        delete temp;
    }
    cout << "List deleted." << endl;
}

int main() {
    Node* head = nullptr;
    deleteList(head); // Testing on empty list
    return 0;
```

}




C:\Users\asus\Desktop\dsa lal
List deleted.
--------------------------------
Process exited after 0.05703 seconds with return value 0
Press any key to continue . . .

Lab 05 - Double Linked List (DLL)

## Implementation of Double Linked List (DLL)

## What is Double Linked List (DLL)?

A **Double Linked List** is a type of linked list where each node contains three parts:

1. **Data**
2. **Next** (points to the next node)
3. **Prev** (points to the previous node)

## Advantages:

1. **Bidirectional Traversal**: Can traverse both forwards and backwards.
2. **Easier Deletion**: Can delete a node easily with access to the previous node.
3. **Efficient Operations**: Insertion and deletion are more efficient at both ends.

## Disadvantages:

1. **Extra Memory**: Requires more memory for the prev pointer.
2. **Complexity**: More complex than singly linked lists.

## Types:

1. **Normal DLL**: Each node points to both the next and previous nodes.
2. **Circular DLL**: The last node points to the head, and the head points back to the last node.

## Real-life Uses:

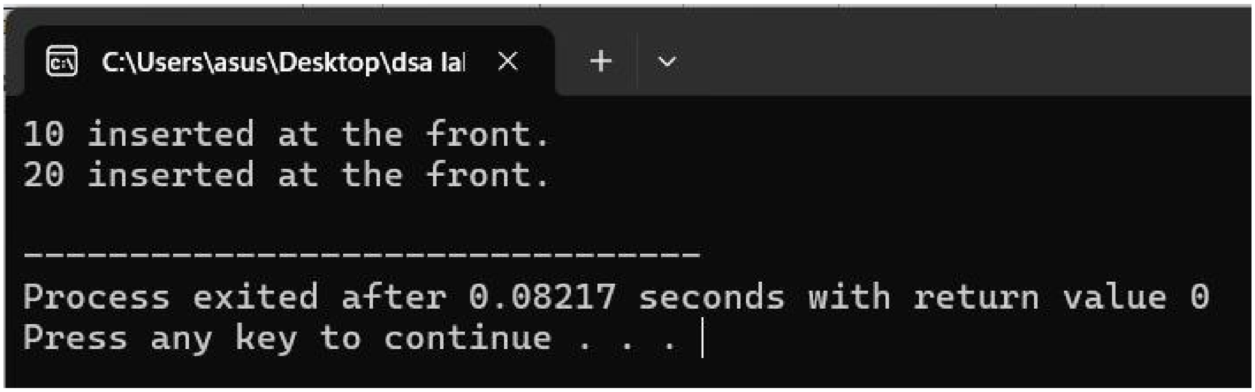
- **Browser History**
- **Undo/Redo operations**
- **Navigation systems** (e.g., media players)

## Example:

## 1. Insertion at the Front

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};

void insertFront(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    newNode->prev = nullptr;

    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
    cout << value << " inserted at the front." << endl;
}

int main() {
```

```
    Node* head = nullptr;
    insertFront(head, 10);
    insertFront(head, 20);
    return 0;
}
```




## 2. Insertion at the Last

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};

void insertLast(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
```

```
    if (head == nullptr) {
        newNode->prev = nullptr;
        head = newNode;
        cout << value << " inserted at the last." << endl;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
    cout << value << " inserted at the last." << endl;
}

int main() {
    Node* head = nullptr;
    insertLast(head, 10);
    insertLast(head, 20);
    return 0;
}
```

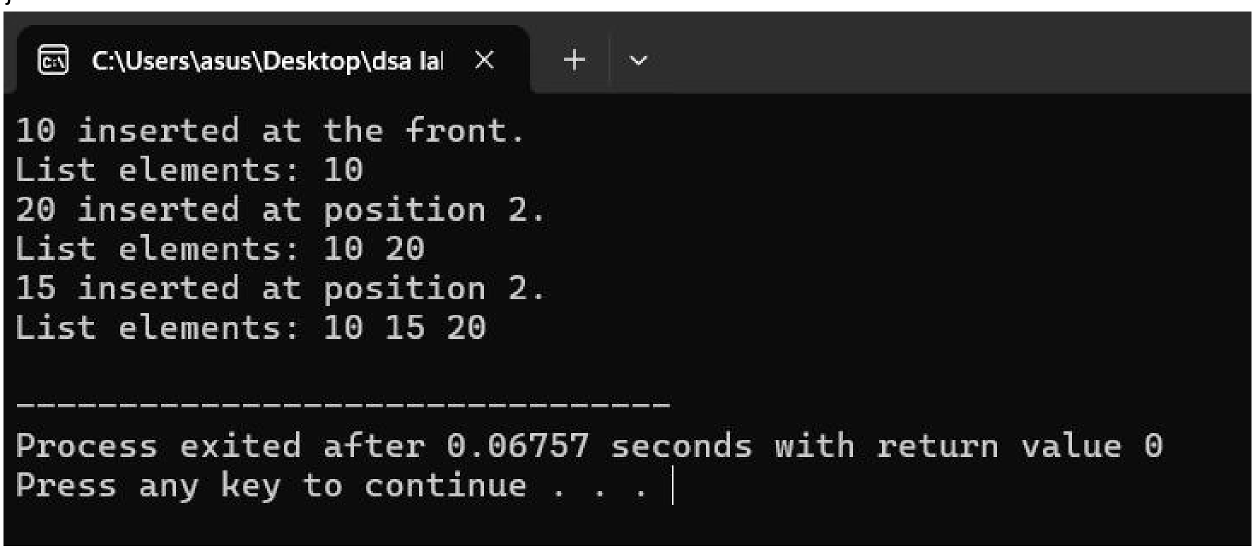
```
C:\Users\asus\Desktop\dsa la
10 inserted at the last.
20 inserted at the last.

--------------------------------
Process exited after 0.06575 seconds with return value 0
Press any key to continue . . .
```

## 3. Insertion at Middle (Specific Position)

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};

// Function to insert at the front
void insertFront(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    newNode->prev = nullptr;

    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
    cout << value << " inserted at the front." << endl;
}

// Function to insert at the middle (specific position)
void insertMid(Node*& head, int value, int position) {
    if (position == 1) {
        insertFront(head, value);
```

```
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }

    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = temp->next;
    newNode->prev = temp;

    if (temp->next != nullptr) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    cout << value << " inserted at position " << position << "." << endl;
}

// Function to traverse and display the list
void traverse(Node* head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    cout << "List elements: ";
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

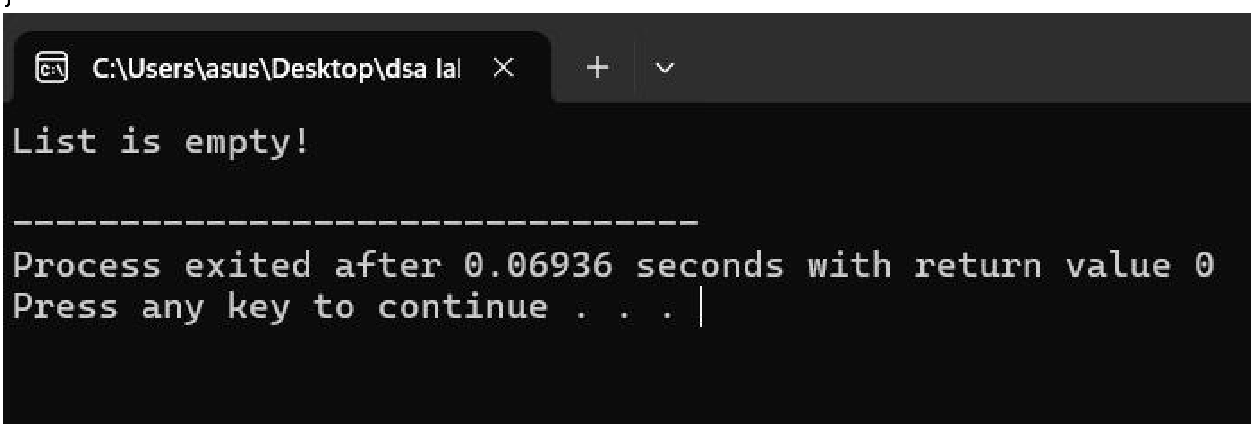
```
int main() {
    Node* head = nullptr;

    insertMid(head, 10, 1); // Insert at position 1
    traverse(head);

    insertMid(head, 20, 2); // Insert at position 2
    traverse(head);

    insertMid(head, 15, 2); // Insert at position 2
    traverse(head);

    return 0;
}
```




## 4. Deletion from the Front

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

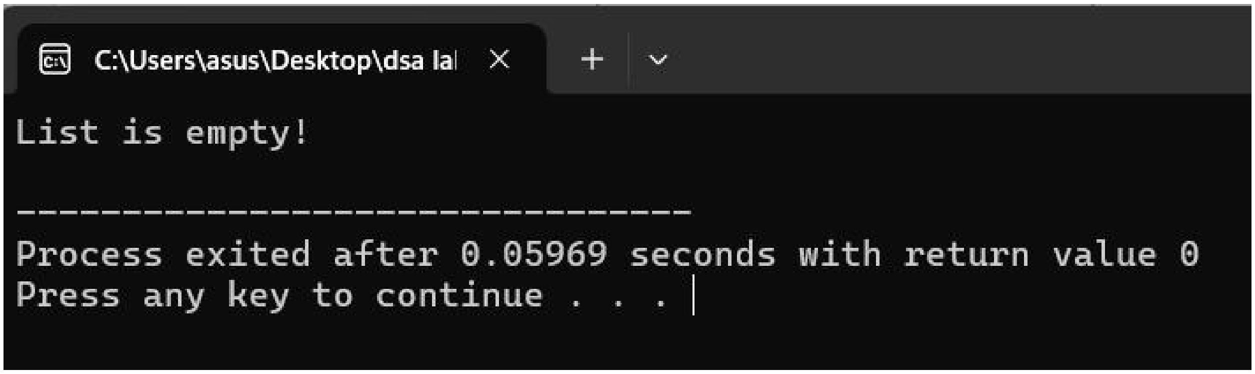
```
void deleteFront(Node*& head) {
   if (head == nullptr) {
      cout << "List is empty!" << endl;
      return;
   }

   Node* temp = head;
   head = head->next;

   if (head != nullptr) {
      head->prev = nullptr;
   }

   delete temp;
   cout << "Node deleted from the front." << endl;
}

int main() {
   Node* head = nullptr;
   deleteFront(head); // Test on an empty list
   return 0;
}
```




## 5. Deletion from the Last

```
#include <iostream>
using namespace std;

struct Node {
```

```cpp
    int data;
    Node* next;
    Node* prev;
};

void deleteLast(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        cout << "Node deleted from the last." << endl;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->prev->next = nullptr;
    delete temp;
    cout << "Node deleted from the last." << endl;
}

int main() {
    Node* head = nullptr;
    deleteLast(head); // Test on an empty list
    return 0;
}
```



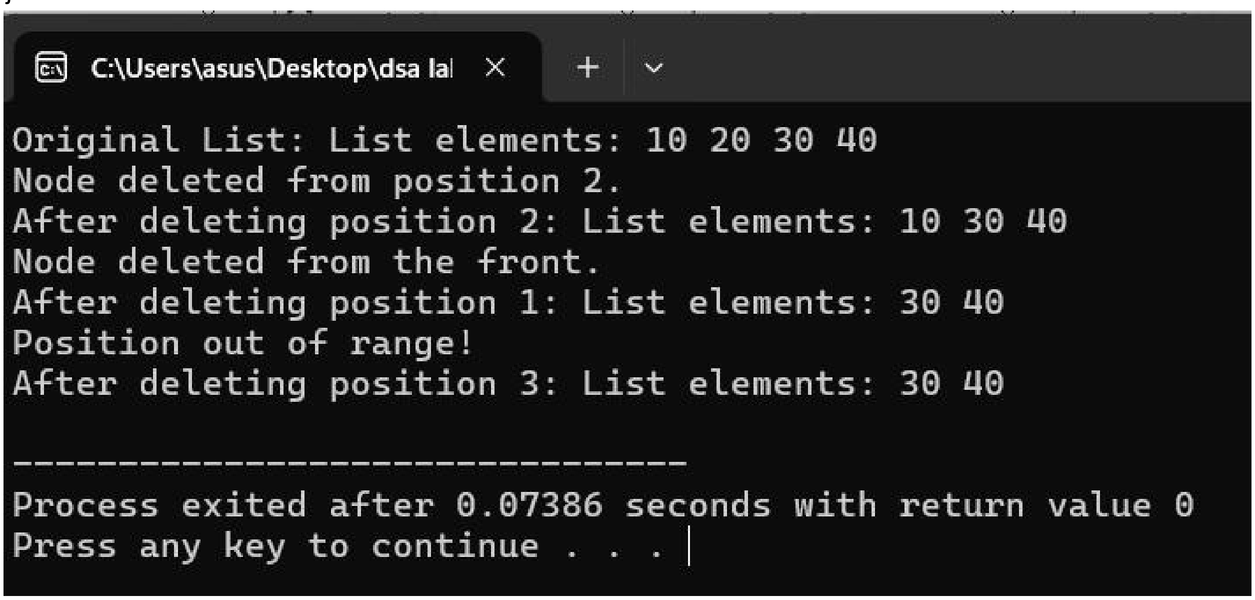

## 6. Deletion from the Middle (Specific Position)

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};

// Function to delete a node from the front
void deleteFront(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    head = head->next;

    if (head != nullptr) {
        head->prev = nullptr;
    }

    delete temp;
    cout << "Node deleted from the front." << endl;
}
```

```
// Function to delete a node from a specific position
void deleteMid(Node*& head, int position) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (position == 1) {
        deleteFront(head); // Reuse deleteFront
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {
        cout << "Position out of range!" << endl;
        return;
    }

    if (temp->next != nullptr) {
        temp->next->prev = temp->prev;
    }

    if (temp->prev != nullptr) {
        temp->prev->next = temp->next;
    }

    delete temp;
    cout << "Node deleted from position " << position << "." << endl;
}

// Function to traverse and display the list
void traverse(Node* head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
```

```
    cout << "List elements: ";
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

// Function to insert a node at the end for testing
void insertLast(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        newNode->prev = nullptr;
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}

int main() {
    Node* head = nullptr;

    // Insert nodes for testing
    insertLast(head, 10);
    insertLast(head, 20);
    insertLast(head, 30);
    insertLast(head, 40);

    cout << "Original List: ";
    traverse(head);
```

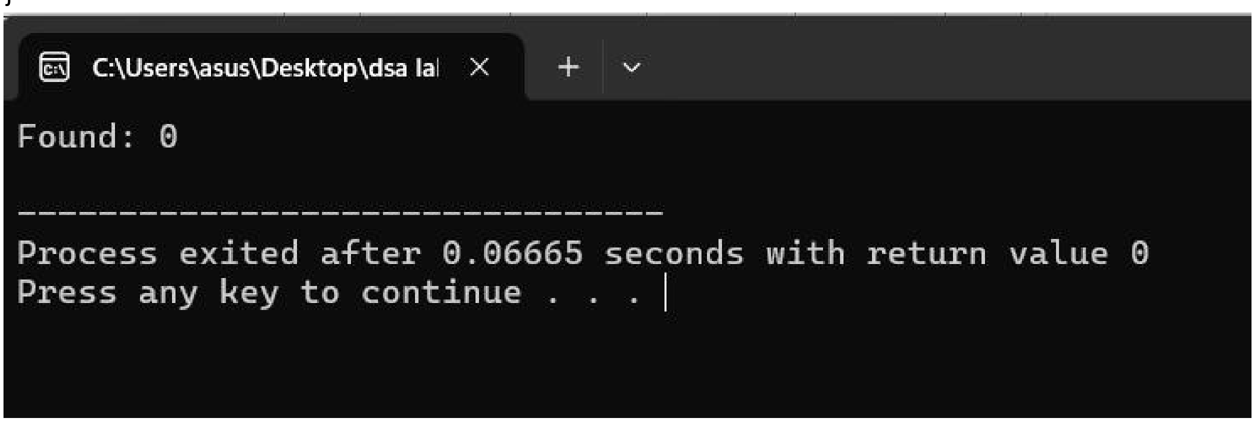
```
  deleteMid(head, 2); // Delete the node at position 2
  cout << "After deleting position 2: ";
  traverse(head);

  deleteMid(head, 1); // Delete the node at position 1
  cout << "After deleting position 1: ";
  traverse(head);

  deleteMid(head, 3); // Delete the node at position 3
  cout << "After deleting position 3: ";
  traverse(head);

  return 0;
}
```




## 7. Searching

```
#include <iostream>
using namespace std;

struct Node {
  int data;
  Node* next;
  Node* prev;
```

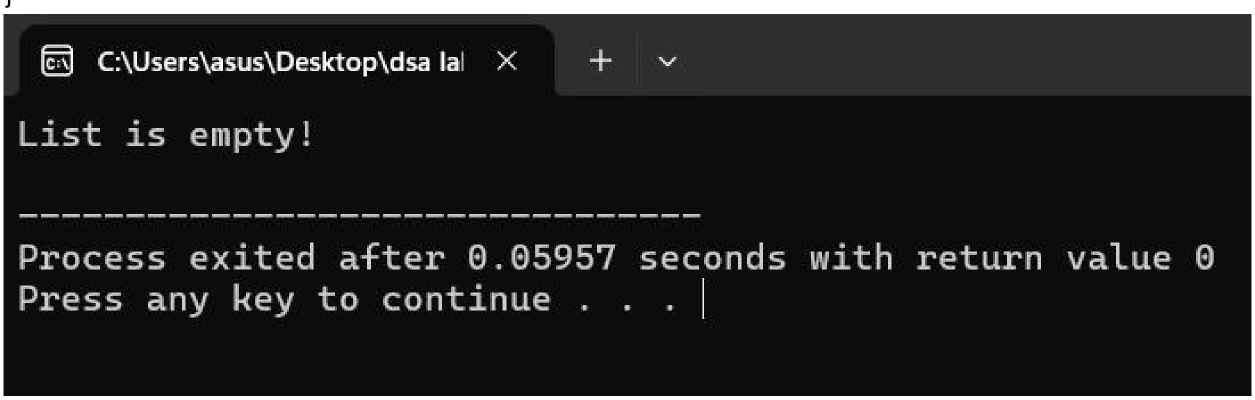
```
};

bool search(Node* head, int value) {
    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == value) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

int main() {
    Node* head = nullptr;
    cout << "Found: " << search(head, 10) << endl; // Test on an empty list
    return 0;
}
```




## 8. Traversing the List

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

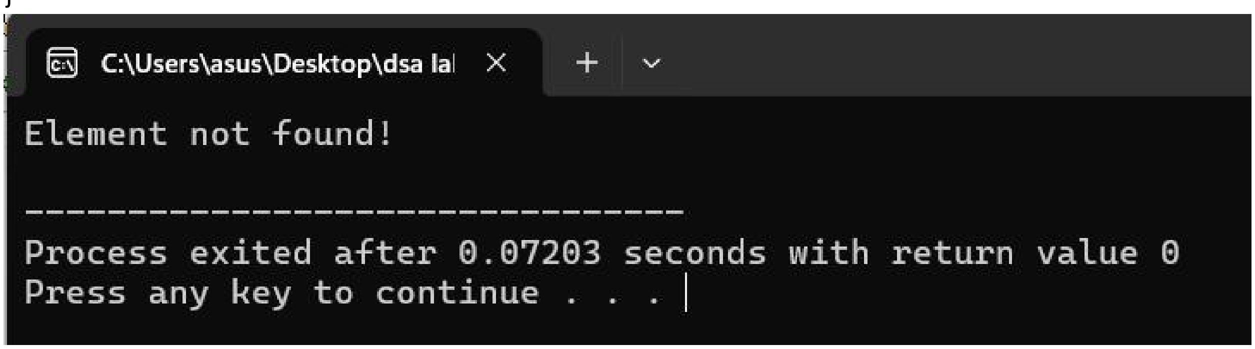
```
void traverse(Node* head) {
   if (head == nullptr) {
      cout << "List is empty!" << endl;
      return;
   }

   Node* temp = head;
   cout << "List elements: ";
   while (temp != nullptr) {
      cout << temp->data << " ";
      temp = temp->next;
   }
   cout << endl;
}

int main() {
   Node* head = nullptr;
   traverse(head); // Test on an empty list
   return 0;
}
```




## 9. Update a Node

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* next;
```

```
    Node* prev;
};

void update(Node* head, int oldValue, int newValue) {
    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == oldValue) {
            temp->data = newValue;
            cout << "Updated value " << oldValue << " to " << newValue << endl;
            return;
        }
        temp = temp->next;
    }
    cout << "Element not found!" << endl;
}

int main() {
    Node* head = nullptr;
    update(head, 10, 20); // Test on an empty list
    return 0;
}
```



Lab 06 - Circular Linked List

# Implementation of Circular Linked List

## What is Circular Linked List?

A Circular Linked List is a type of linked list where the last node points back to the first node, forming a loop.

## Advantages:

1. **Continuous Traversal**: No need to reset the pointer; you can keep looping through the list.
2. **Efficient Memory Use**: Useful for tasks that require circular or repeated processing.

## Applications:

1. **Round Robin Scheduling** in OS.
2. **Circular Buffers** for data handling.
3. **Music Playlists** that loop indefinitely.
4. **Circular Queues** for continuous processing.

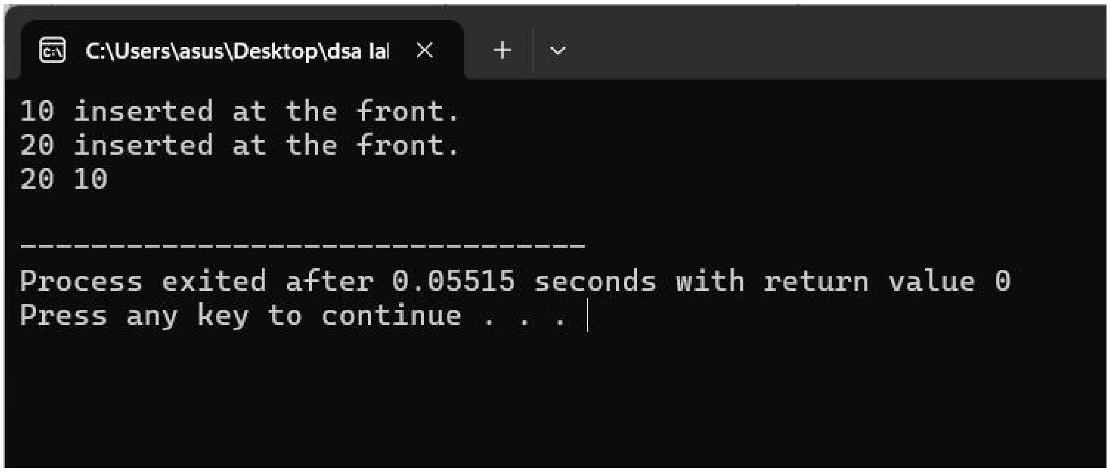
## Example:

### 1. Insertion at Front

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* next;
};
```

```
void insertFront(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;

    if (head == nullptr) {
        newNode->next = newNode;
        head = newNode;
    } 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;
}

void traverse(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;
}

int main() {
    Node* head = nullptr;

    insertFront(head, 10);
    insertFront(head, 20);
    traverse(head);

    return 0;
```

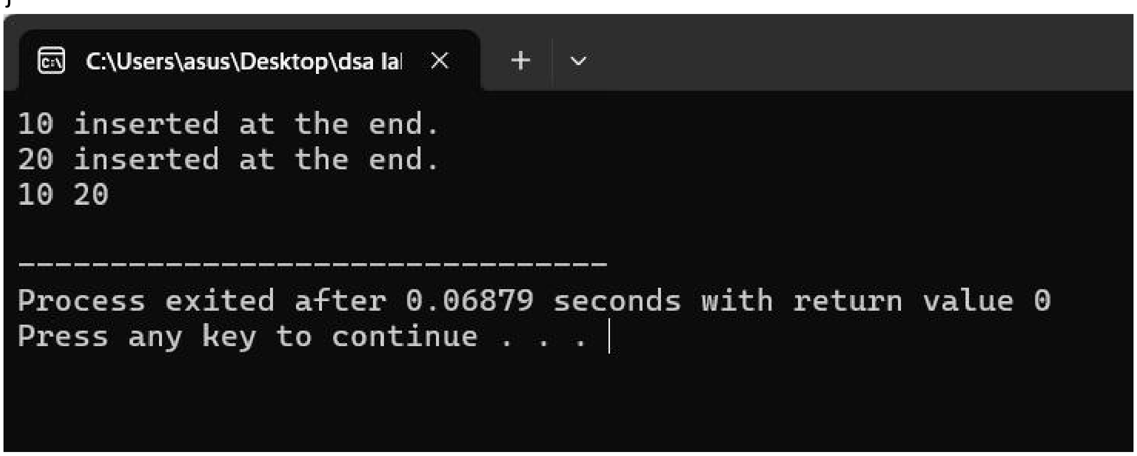
```
}
```




## 2. Insertion at Last

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* next;
};

void insertLast(Node*& head, int value) {
   Node* newNode = new Node();
   newNode->data = value;

   if (head == nullptr) {
      newNode->next = newNode;
      head = newNode;
   } else {
      Node* temp = head;
      while (temp->next != head) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->next = head;
   }
   cout << value << " inserted at the end." << endl;
```

```
}

void traverse(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;
}

int main() {
    Node* head = nullptr;

    insertLast(head, 10);
    insertLast(head, 20);
    traverse(head);

    return 0;
}
```




## 3.Insertion at a Specific Position

```
#include <iostream>
using namespace std;
```

```
struct Node {
    int data;
    Node* next;
};

void insertMid(Node*& head, int value, int position) {
    if (position == 1) {
        Node* newNode = new Node();
        newNode->data = value;

        if (head == nullptr) {
            newNode->next = newNode;
            head = newNode;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            newNode->next = head;
            temp->next = newNode;
            head = newNode;
        }
        cout << value << " inserted at position 1." << endl;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next != head; i++) {
        temp = temp->next;
    }

    if (temp->next == head && position > 2) {
        cout << "Position out of range!" << endl;
        return;
    }

    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = temp->next;
    temp->next = newNode;
```

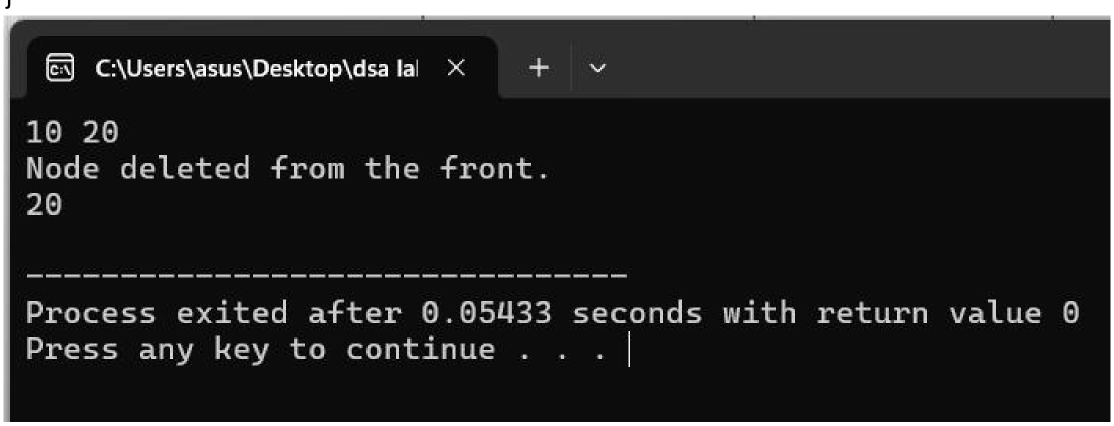
```
    cout << value << " inserted at position " << position << "." << endl;
}

void traverse(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;
}

int main() {
    Node* head = nullptr;

    insertMid(head, 10, 1);
    insertMid(head, 20, 2);
    insertMid(head, 15, 2);
    traverse(head);

    return 0;
}
```

```
C:\Users\asus\Desktop\dsa lal
10 inserted at position 1.
20 inserted at position 2.
15 inserted at position 2.
10 15 20

--------------------------------
Process exited after 0.06308 seconds with return value 0
Press any key to continue . . .
```

## 4. Deletion from Front

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void deleteFront(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (head->next == head) {
        delete head;
        head = nullptr;
    } else {
        Node* temp = head;
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        head = head->next;
        last->next = head;
        delete temp;
    }
    cout << "Node deleted from the front." << endl;
}

void traverse(Node* head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    do {
```

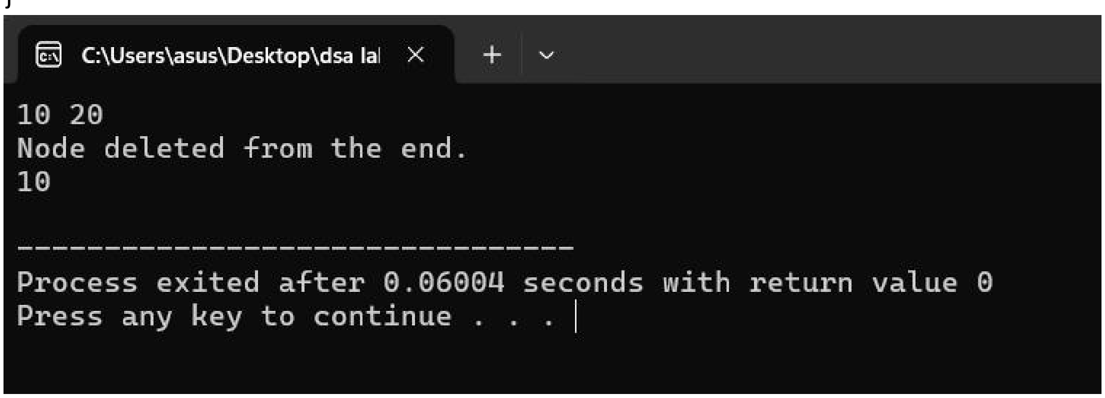
```
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Example: Inserting some nodes
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

    Node* second = new Node();
    second->data = 20;
    second->next = head;
    head->next = second;

    traverse(head);
    deleteFront(head);
    traverse(head);

    return 0;
}
```



## 5.Deletion from Last

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void deleteLast(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (head->next == head) {
        delete head;
        head = nullptr;
    } else {
        Node* temp = head;
        Node* prev = nullptr;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head;
        delete temp;
    }
    cout << "Node deleted from the end." << endl;
}

void traverse(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;
}

int main() {
    Node* head = nullptr;

    // Example: Inserting some nodes
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

    Node* second = new Node();
    second->data = 20;
    second->next = head;
    head->next = second;

    traverse(head);
    deleteLast(head);
    traverse(head);

    return 0;
}
```

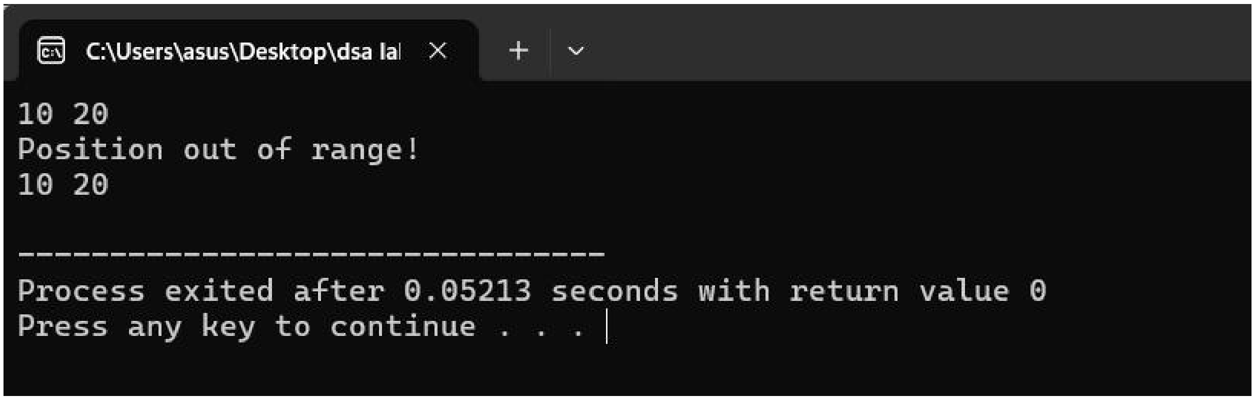



## 6.Deletion from a Specific Position

```
#include <iostream>
using namespace std;
```

```
struct Node {
    int data;
    Node* next;
};

void deleteMid(Node*& head, int position) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    if (position == 1) {
        if (head->next == head) {
            delete head;
            head = nullptr;
        } else {
            Node* temp = head;
            Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            head = head->next;
            last->next = head;
            delete temp;
        }
        cout << "Node deleted from position 1." << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;
    for (int i = 1; i < position && temp->next != head; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp->next == head && position > 1) {
        cout << "Position out of range!" << endl;
        return;
    }
```

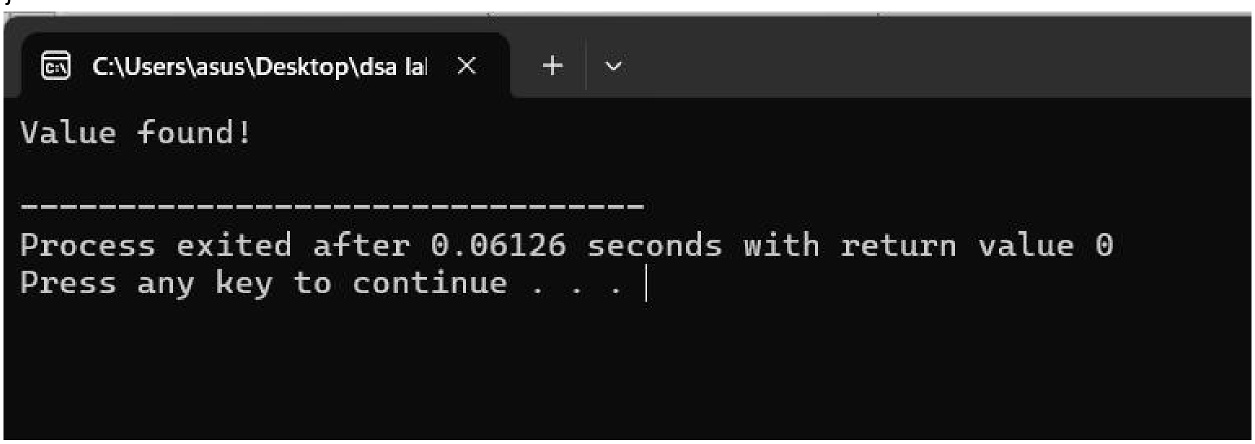
```
    prev->next = temp->next;
    delete temp;
    cout << "Node deleted from position " << position << "." << endl;
}

void traverse(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;
}

int main() {
    Node* head = nullptr;

    // Example: Inserting some nodes
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

    Node* second = new Node();
    second->data = 20;
    second->next = head;
    head->next = second;

    traverse(head);
    deleteMid(head, 2);
    traverse(head);

    return 0;
}
```




## 7.Searching

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* next;
};

bool search(Node* head, int value) {
   if (head == nullptr) {
      return false;
   }

   Node* temp = head;
   do {
      if (temp->data == value) {
         return true;
      }
      temp = temp->next;
   } while (temp != head);

   return false;
}

int main() {
   Node* head = nullptr;
```

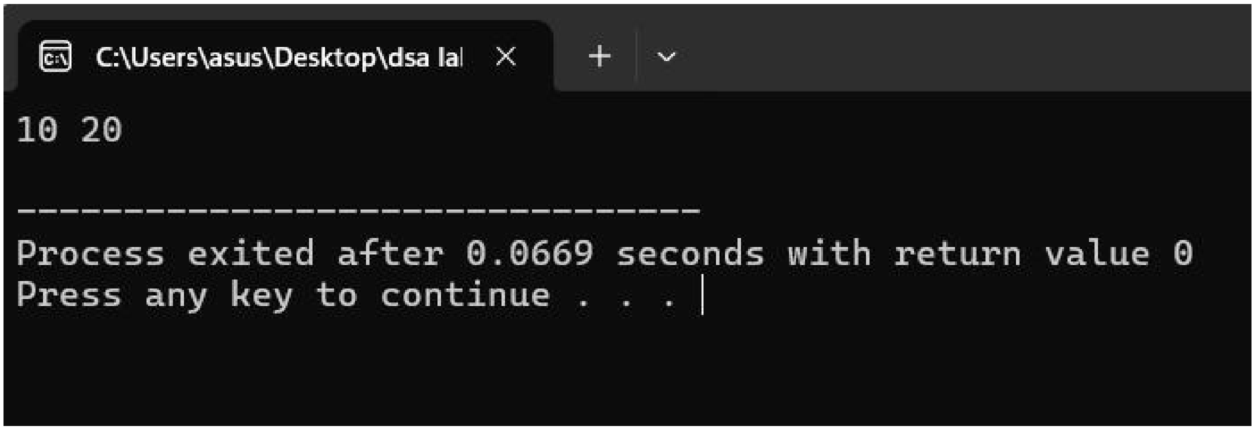
```
    // Example: Creating a circular linked list
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

    Node* second = new Node();
    second->data = 20;
    second->next = head;
    head->next = second;

    if (search(head, 20)) {
        cout << "Value found!" << endl;
    } else {
        cout << "Value not found!" << endl;
    }

    return 0;
}
```




## 8. Traversing

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
```

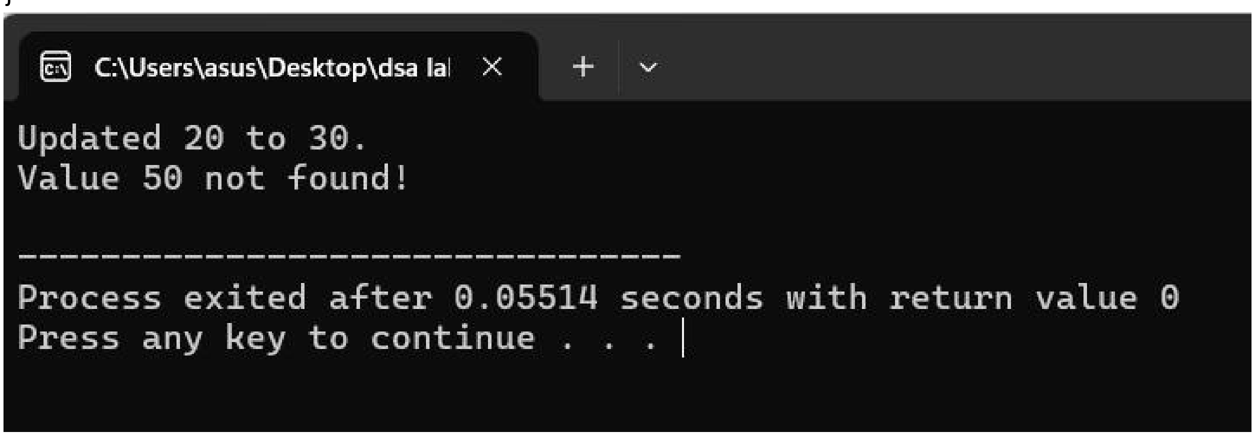
```
};

void traverse(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;
}

int main() {
    Node* head = nullptr;

    // Example: Creating a circular linked list
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

    Node* second = new Node();
    second->data = 20;
    second->next = head;
    head->next = second;

    traverse(head);

    return 0;
}
```




## 9.Update

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

void update(Node* head, int oldValue, int newValue) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    do {
        if (temp->data == oldValue) {
            temp->data = newValue;
            cout << "Updated " << oldValue << " to " << newValue << "." << endl;
            return;
        }
        temp = temp->next;
    } while (temp != head);

    cout << "Value " << oldValue << " not found!" << endl;
}
```

```
int main() {
   Node* head = nullptr;

   // Example: Creating a circular linked list
   Node* newNode = new Node();
   newNode->data = 10;
   head = newNode;
   head->next = head;

   Node* second = new Node();
   second->data = 20;
   second->next = head;
   head->next = second;

   update(head, 20, 30);
   update(head, 50, 60);

   return 0;
}
```



# Lab 07 - Binary Search Tree (BST)

## Implementation of Circular Linked List

## What is Binary Search Tree (BST)?

A **Binary Search Tree** is a type of binary tree where each node has at most two children. It follows these properties:

- Left subtree nodes are smaller than the root.
- Right subtree nodes are larger than the root.
- No duplicate values are allowed.

## Advantages:

1. **Efficient searching** with O(log n)O(\log n)O(logn) time complexity (in balanced trees).
2. **Dynamic structure**, grows and shrinks as needed.
3. **Sorted data** can be retrieved via in-order traversal.
4. **Efficient insertion and deletion** in balanced trees.

## Types of BST:

1. **Normal BST:** No balance enforced.
2. **Balanced BST:** Like AVL or Red-Black Trees, maintains O(log n)O(\log n)O(logn) operations.
3. **Unbalanced BST:** Can degrade to linked list form, causing O(n)O(n)O(n) operations.

**Applications:**

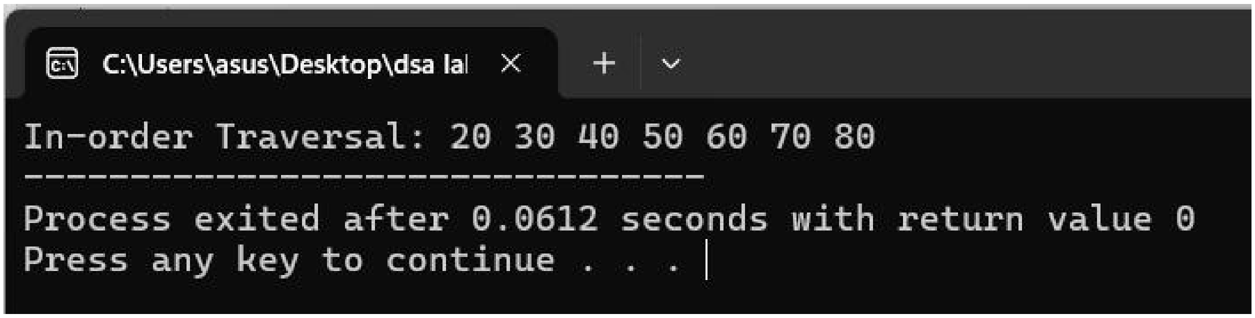
1. **Database indexing**
2. **File systems**
3. **Dictionary implementation**
4. **Routing tables** in networks

## Examples:

## 1. Insertion in Binary Search Tree

```
#include <iostream>
using namespace std;

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

Node* insert(Node* root, int value) {
   if (root == nullptr) {
      root = new Node();
      root->data = value;
      root->left = root->right = nullptr;
   } else if (value < root->data) {
      root->left = insert(root->left, value);
   } else {
      root->right = insert(root->right, value);
   }
   return root;
}

void inorderTraversal(Node* root) {
   if (root == nullptr) return;
   inorderTraversal(root->left);
   cout << root->data << " ";
   inorderTraversal(root->right);
}

int main() {
   Node* root = nullptr;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 20);
   insert(root, 40);
```

```
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    cout << "In-order Traversal: ";
    inorderTraversal(root);
    return 0;
}
```



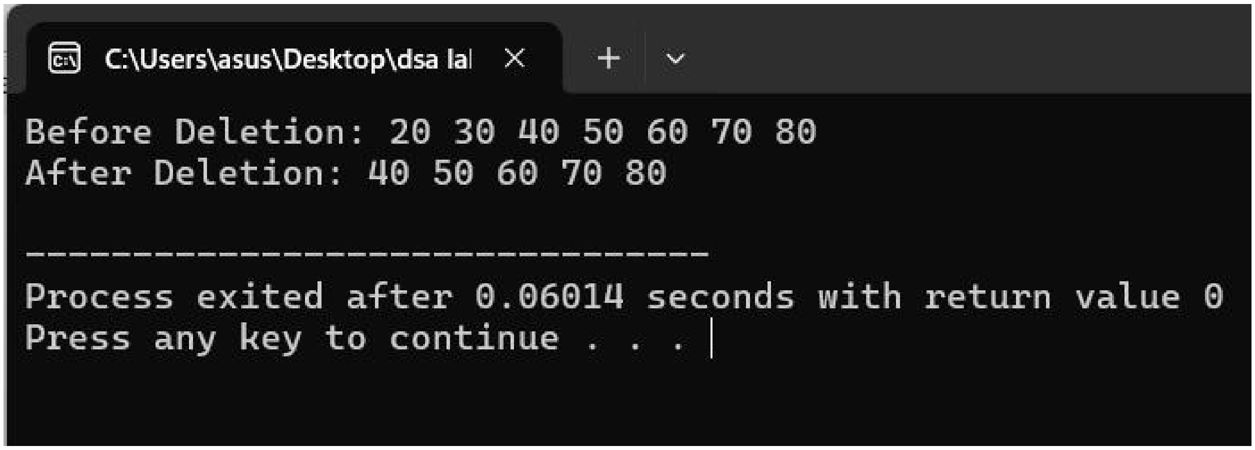

## 2. Deletion in Binary Search Tree

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to initialize node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Insert function for Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // Create and return a new node if root is null
    }
```

```
    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

// Find the minimum value node in a given tree
Node* findMin(Node* root) {
    while (root->left != nullptr) root = root->left;
    return root;
}

// Delete a node from the binary search tree
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) return root;

    if (value < root->data) {
        root->left = deleteNode(root->left, value);  // Traverse left subtree
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);  // Traverse right subtree
    } else {
        // Node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children: get the inorder successor
        Node* temp = findMin(root->right);

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

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

    return root;
}

// In-order traversal of the binary search tree
void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert values into the binary search tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Before Deletion
    cout << "Before Deletion: ";
    inorderTraversal(root);
    cout << endl;

    // Delete nodes
    root = deleteNode(root, 20);  // Delete node with value 20
    root = deleteNode(root, 30);  // Delete node with value 30

    // After Deletion
    cout << "After Deletion: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
```

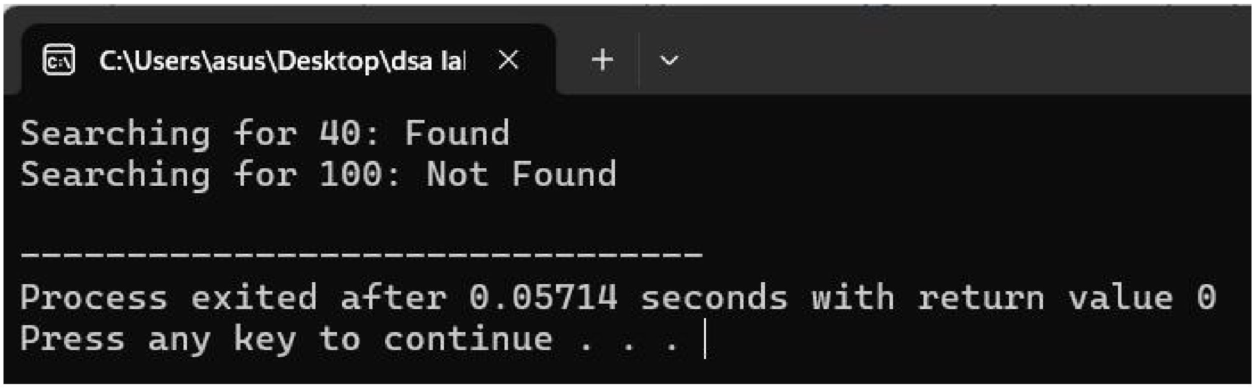
```
}
```




## 3. Searching in Binary Search Tree

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to initialize node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Insert function for Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // Create and return a new node if root is null
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
```

```
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

// Search function for Binary Search Tree
bool search(Node* root, int value) {
    if (root == nullptr) return false;  // Base case: if root is null, value is not found
    if (root->data == value) return true;  // If root's data matches the value, return true

    if (value < root->data) {
        return search(root->left, value);  // Search in the left subtree
    } else {
        return search(root->right, value);  // Search in the right subtree
    }
}

int main() {
    Node* root = nullptr;

    // Insert values into the binary search tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Perform search operations
    cout << "Searching for 40: " << (search(root, 40) ? "Found" : "Not Found") << endl;
    cout << "Searching for 100: " << (search(root, 100) ? "Found" : "Not Found") << endl;

    return 0;
}
```




## 4.Traversal (In-order, Pre-order, Post-order)

### In-order Traversal

```
#include <iostream>
using namespace std;

void inorderTraversal(Node* root) {
   if (root == nullptr) return;
   inorderTraversal(root->left);
   cout << root->data << " ";
   inorderTraversal(root->right);
}
```

### Pre-order Traversal

```
#include <iostream>
using namespace std;

void preorderTraversal(Node* root) {
   if (root == nullptr) return;
   cout << root->data << " ";
   preorderTraversal(root->left);
   preorderTraversal(root->right);
}
```

### Post-order Traversal

```
#include <iostream>
using namespace std;

void postorderTraversal(Node* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->data << " ";
}
```

## Main Program (Traversals)

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Insertion function for 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 {
        root->right = insert(root->right, value);
```

```
    }

    return root;
}

// In-order traversal: left, root, right
void inorderTraversal(Node* root) {
    if (root == nullptr) return;
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

// Pre-order traversal: root, left, right
void preorderTraversal(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

// Post-order traversal: left, right, root
void postorderTraversal(Node* root) {
    if (root == nullptr) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = nullptr;

    // Insert values into the binary search tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```

```
    // Traversals
    cout << "In-order Traversal: ";
    inorderTraversal(root);
    cout << endl;

    cout << "Pre-order Traversal: ";
    preorderTraversal(root);
    cout << endl;

    cout << "Post-order Traversal: ";
    postorderTraversal(root);
    cout << endl;

    return 0;
}
```