Submitted to: Mam Irsha Qureshi

Submitted by: ALI HASSAN

Registration # : 2023-BS-AI-038

Department: BS-AI (A)

Semester: 3

LAB MANUAL

Table of Contents

Singly Linked List

Insertion (Front, Mid, Last)

Deletion (Front, Mid, Last)

Searching

Edit

Update

Find Index

Traversing

Doubly Linked List

Insertion (Front, Mid, Last)

Deletion (Front, Mid, Last)

Searching

Edit

Update

Find Index

Traversing

Circular Linked List

Insertion (Front, Mid, Last)

Deletion (Front, Mid, Last)

Searching

Edit

Update

Find Index

Traversing

Binary Search Tree (BST)

Insertion

Deletion

Searching

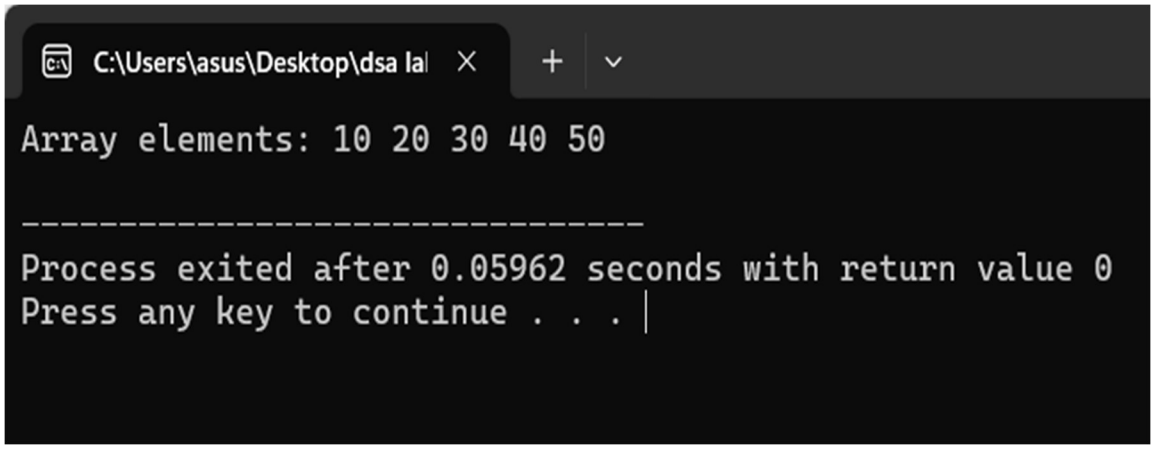
Traversals (In-Order, Pre-Order, Post-Order)

Lab 01 – Arrays

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;
}
```

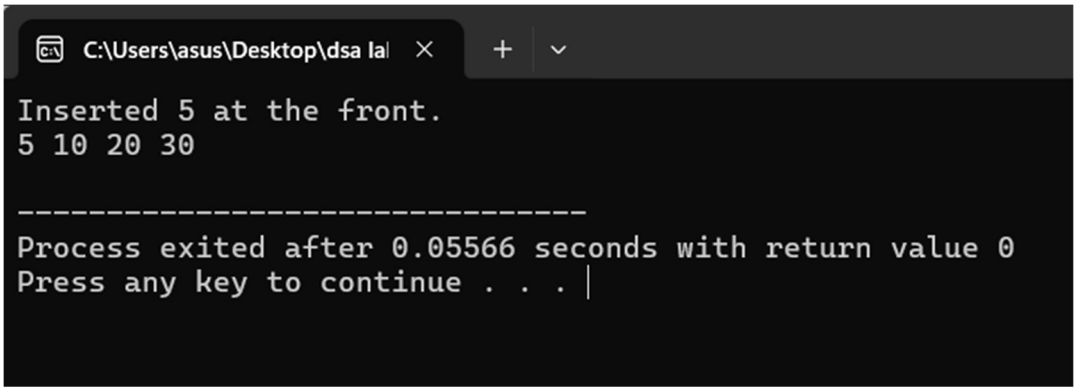


2. Insertion (Front, Mid, Last)

- **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};
```

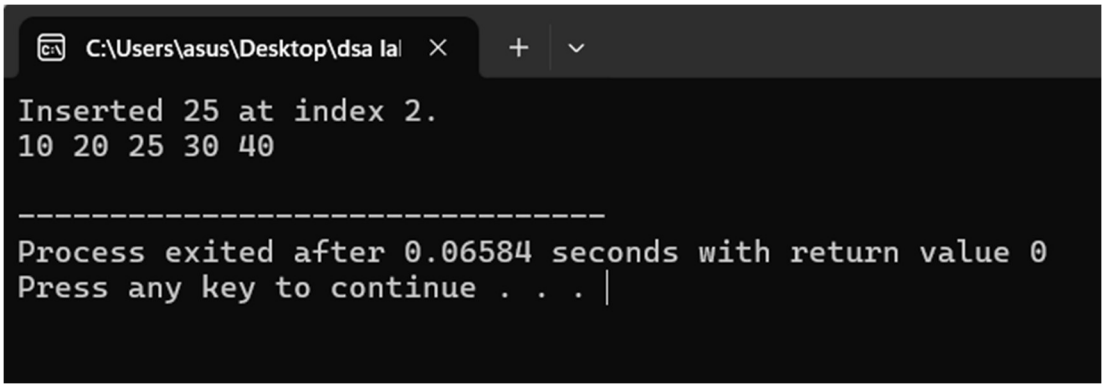
```
    int size = 3;

    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;
}
```

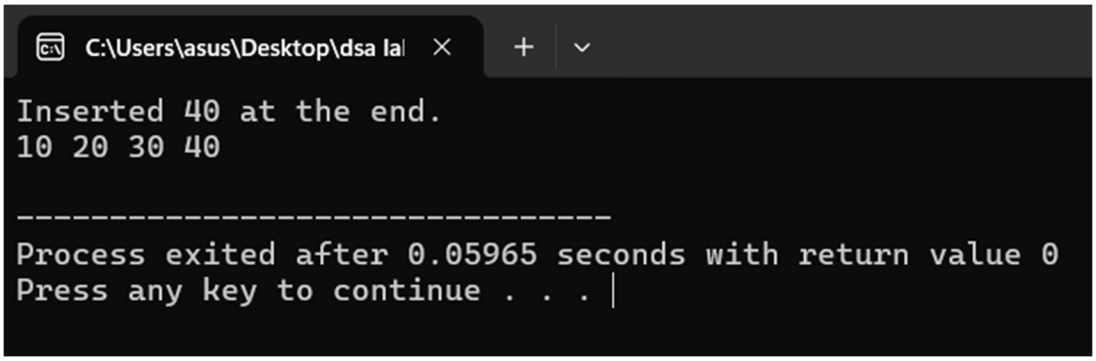
```
int main() {
    const int capacity = 10;
    int arr[capacity] = {10, 20, 30, 40};
    int size = 4;

    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;
}
```

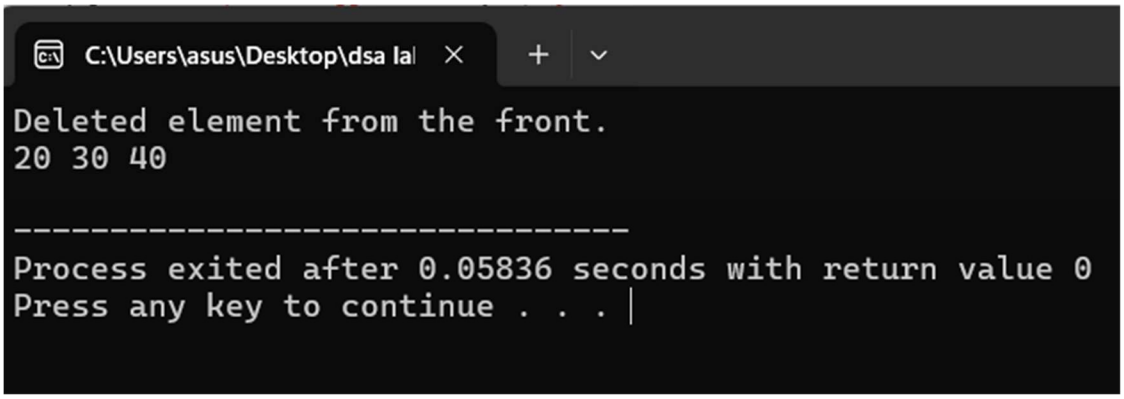


3. Deletion (Front, Mid, Last)

- **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];
    }
```

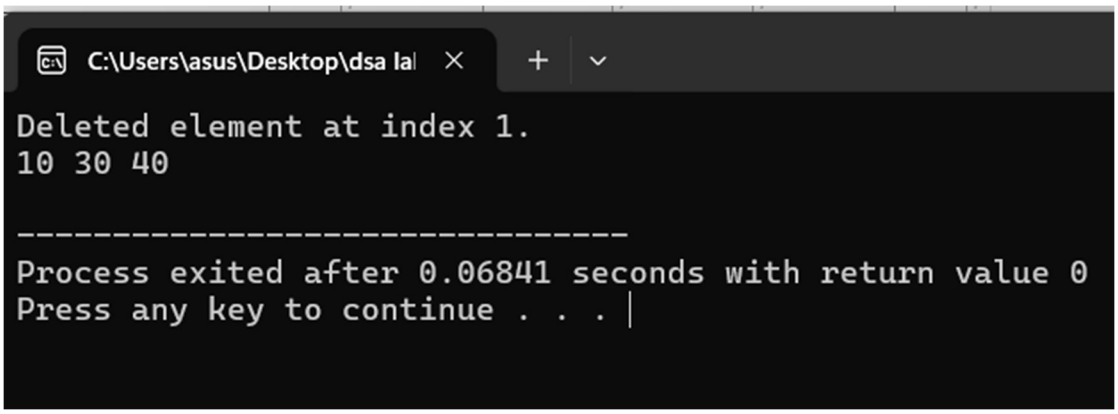
```
    size--;
    cout << "Deleted element from the front." << endl;
}

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];
```

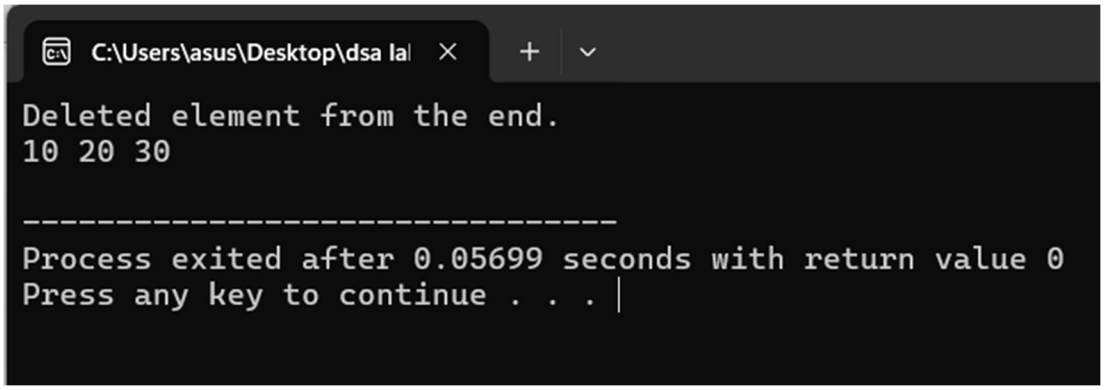
```
    }
    size--;
    cout << "Deleted element at index " << index << "." << endl;
}

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--;
```

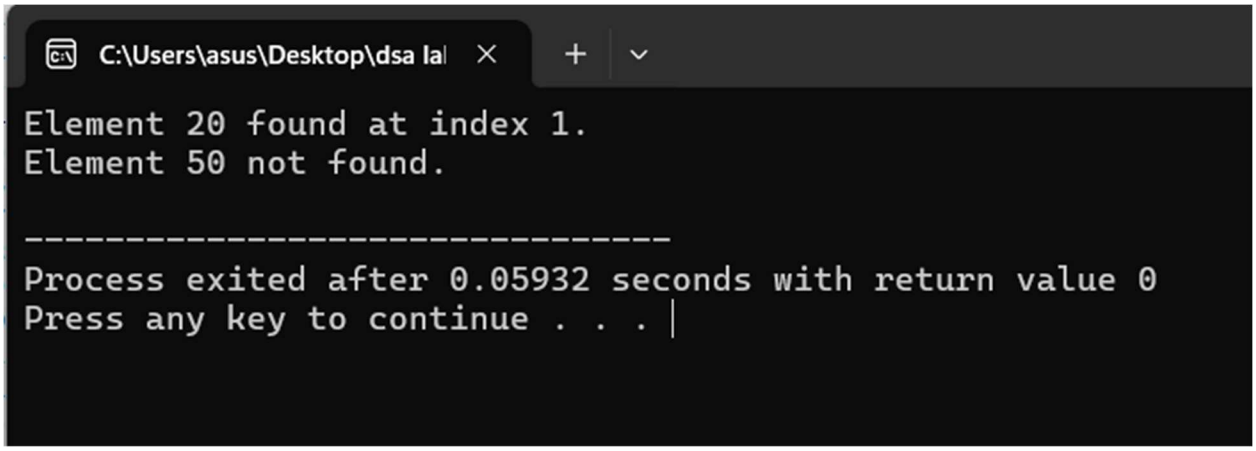
```
    cout << "Deleted element from the end." << endl;
}

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

```
#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;
        }
    }
```

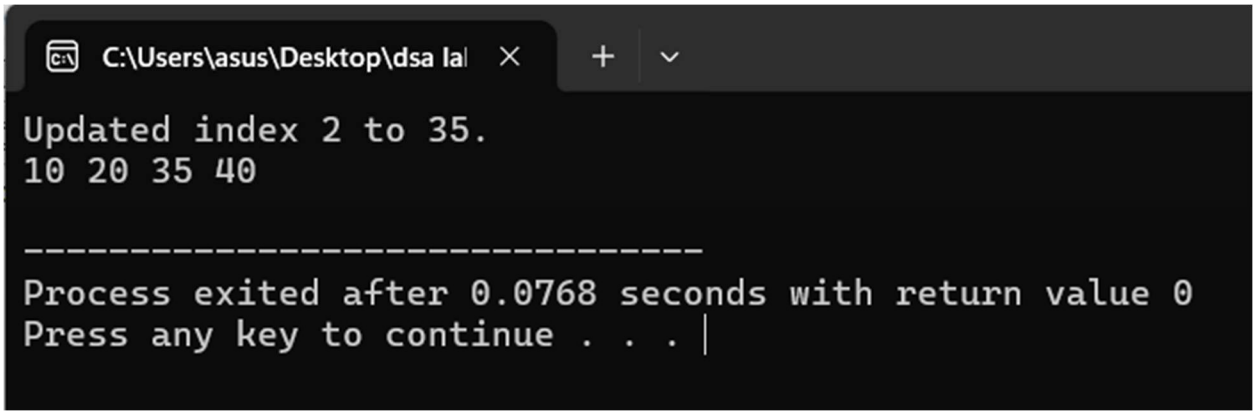
```
    cout << "Element " << element << " not found." << endl;
}

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

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

    return 0;
}
```



5. Update

```
#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;
}
```

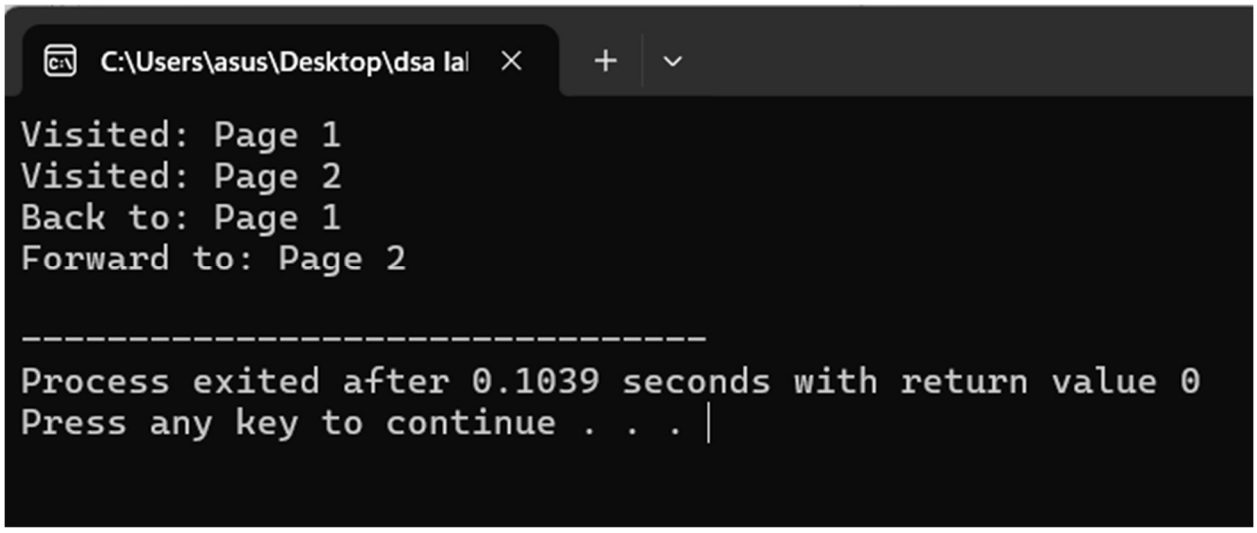


Lab 02- Stack

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);
```

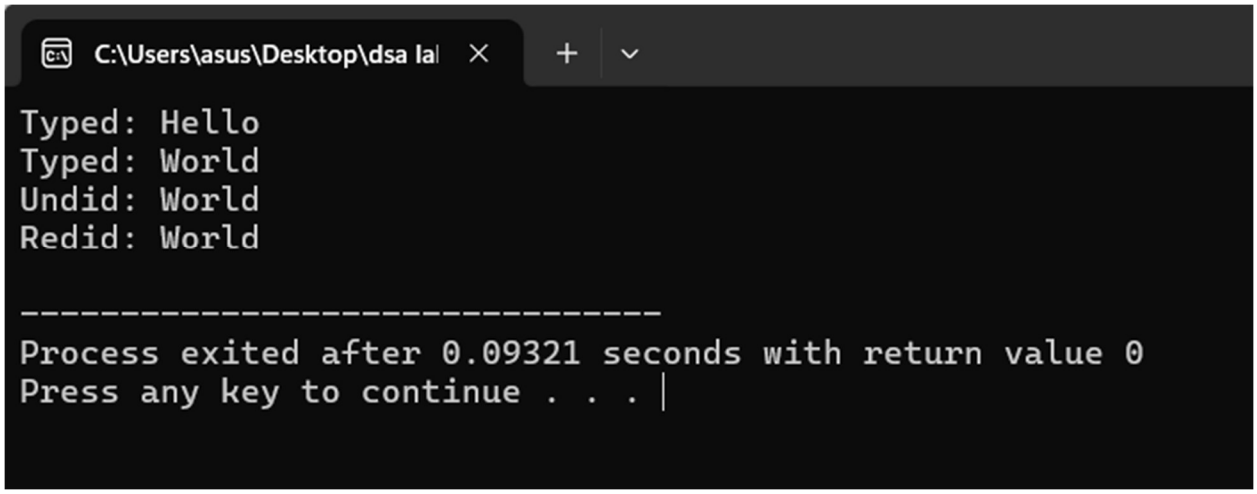
```
        while (!forwardStack.empty()) forwardStack.pop();  // Clear forward
stack
        cout << "Visited: " << page << endl;
    }

    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;
    }
```

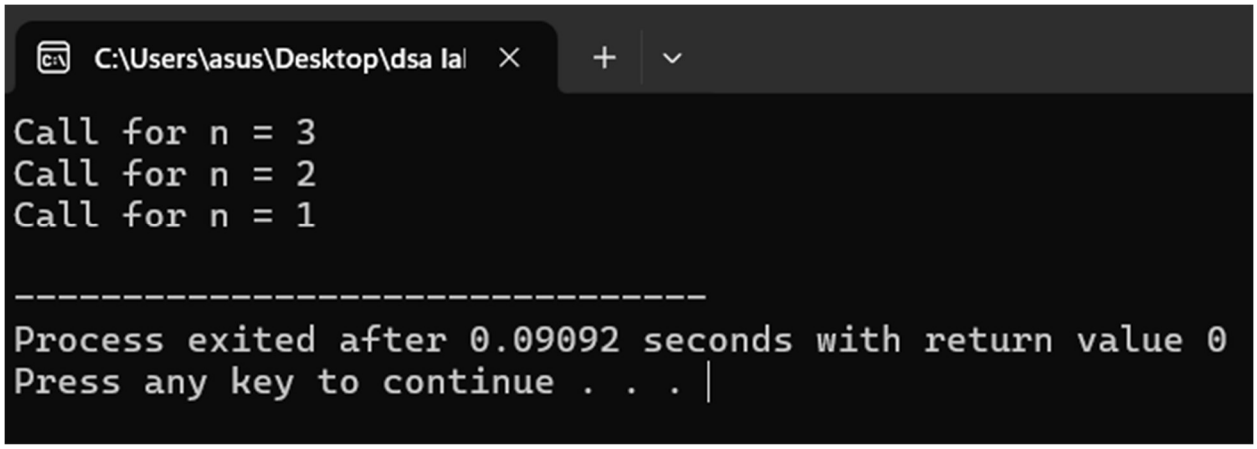
```
    void redo() {
        if (redoStack.empty()) {
            cout << "Nothing to redo!" << endl;
            return;
        }
        string lastText = redoStack.top();
        redoStack.pop();
        undoStack.push(lastText);
        cout << "Redid: " << lastText << endl;
    }
};

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



1. Function Call Stack (Recursion)

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

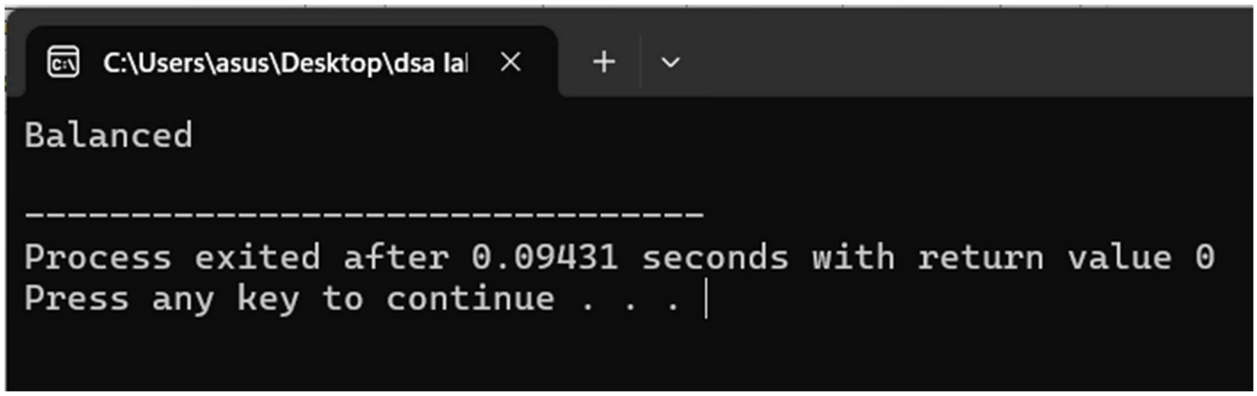
```
void recursiveFunction(int n) {
    stack<int> callStack;
    callStack.push(n);
    if (n > 0) {
        cout << "Call for n = " << n << endl;
        recursiveFunction(n - 1);
    }
    callStack.pop();
}

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



2. 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;
}
```

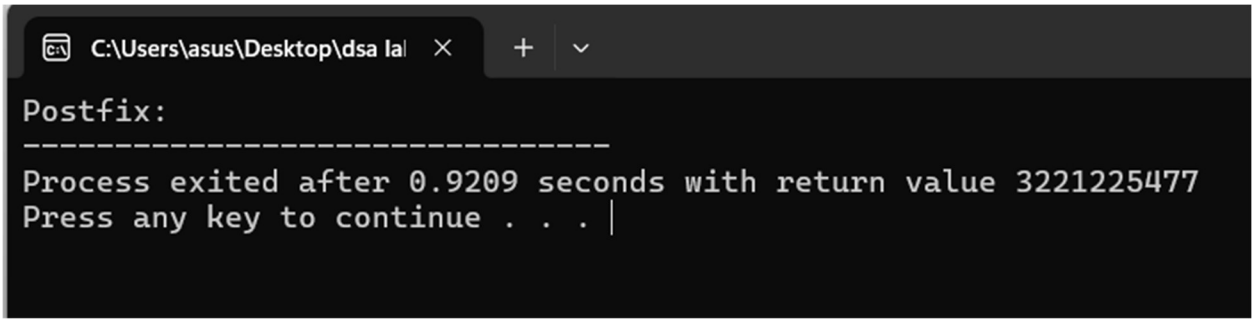


5. Infix to Postfix Conversion

```
#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;
```

}

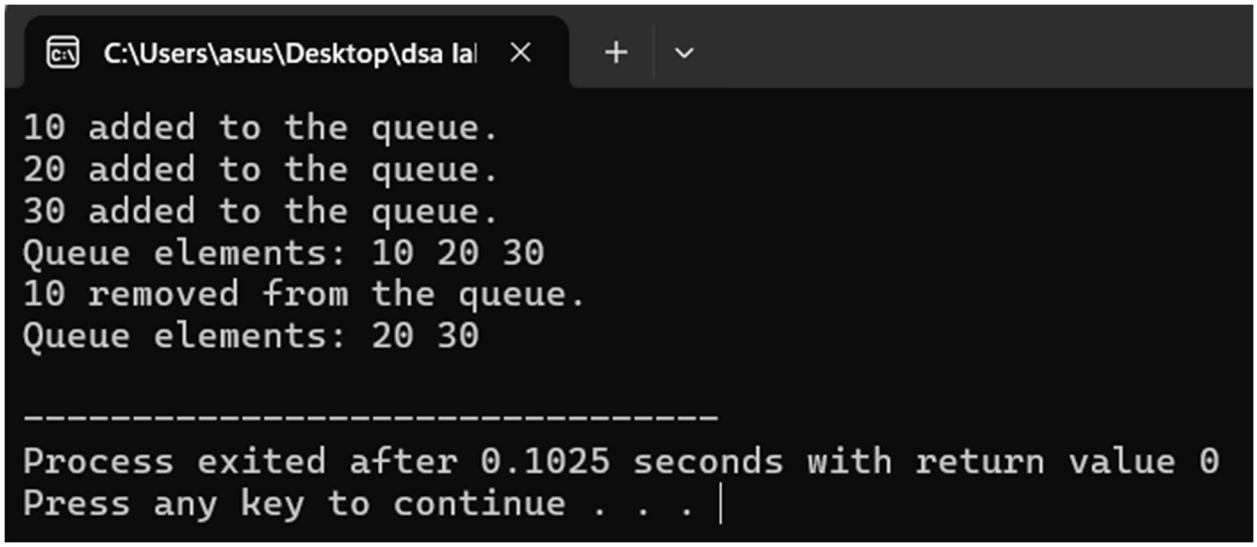


Lab 03 – Queue

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;
}
```

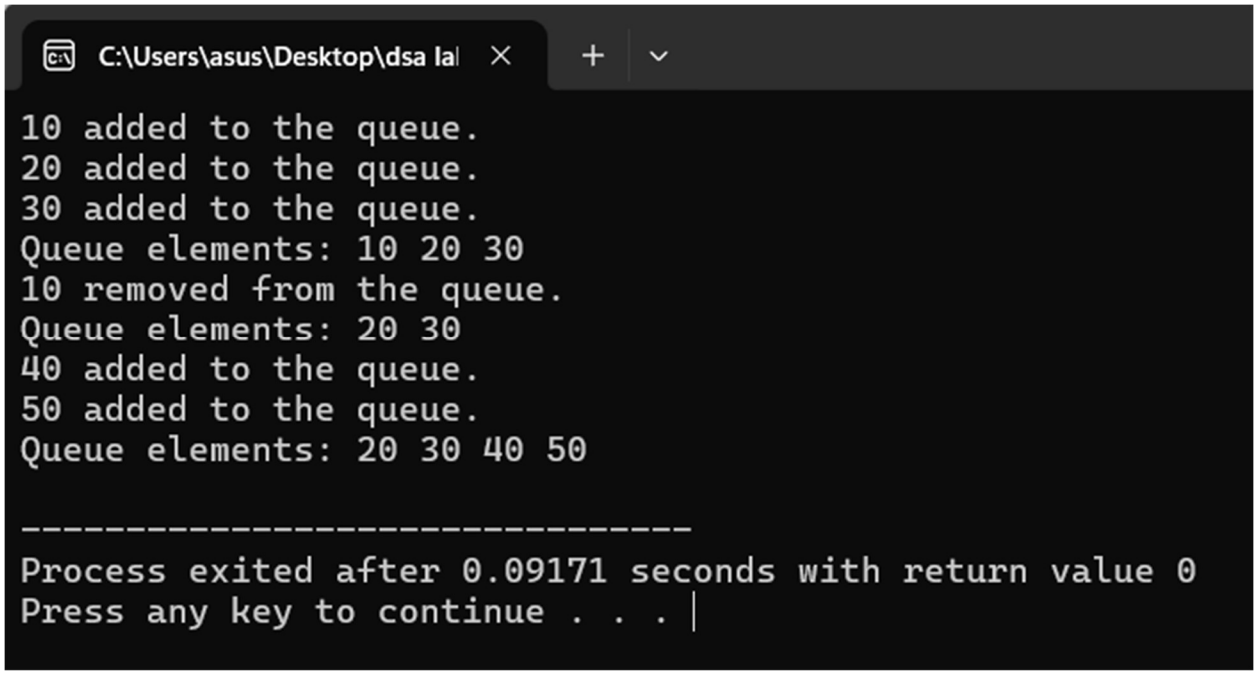


2. Circular Queue (Array-Based)

```
#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;
}
```



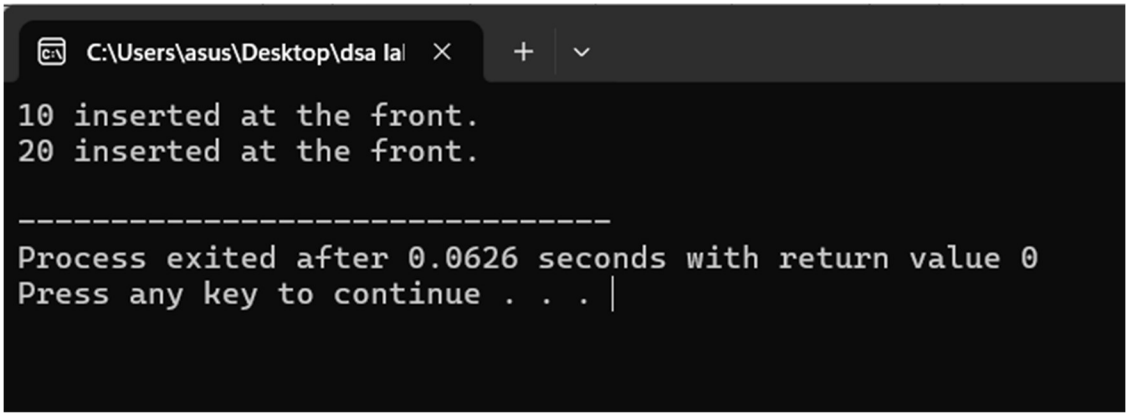
Lab 04 - Single Linked List (SLL)

1. **Playlist Management**: Music players use it to manage playlists.

    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;
}
```

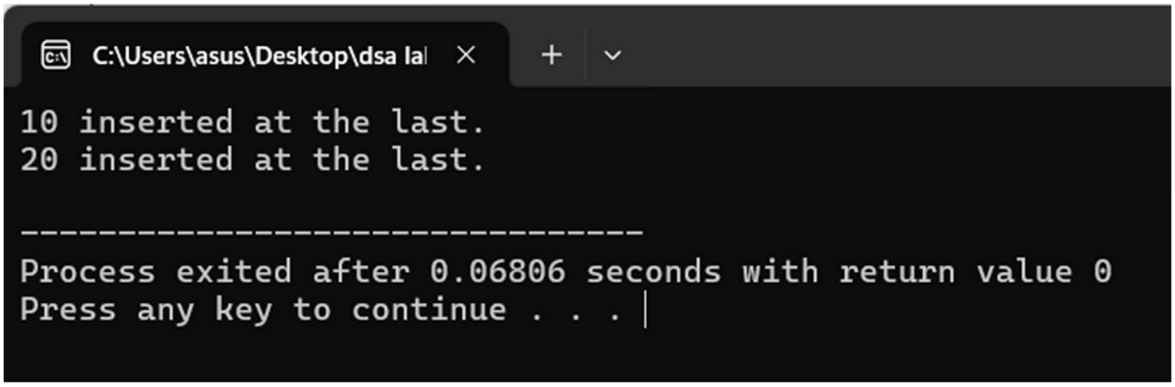


2. Insertion at End:

```
#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;
}
```

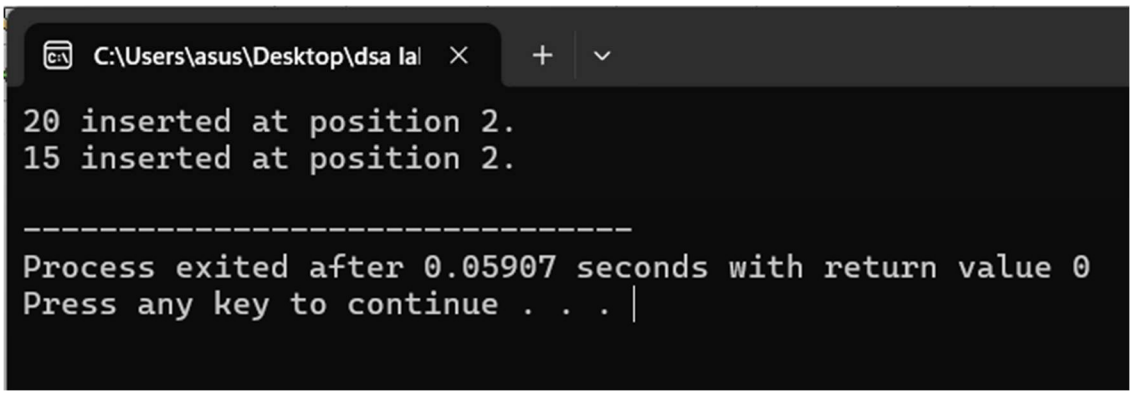
```
int main() {
    Node* head = nullptr;
    insertLast(head, 10);
    insertLast(head, 20);
    return 0;
}
```



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;
```

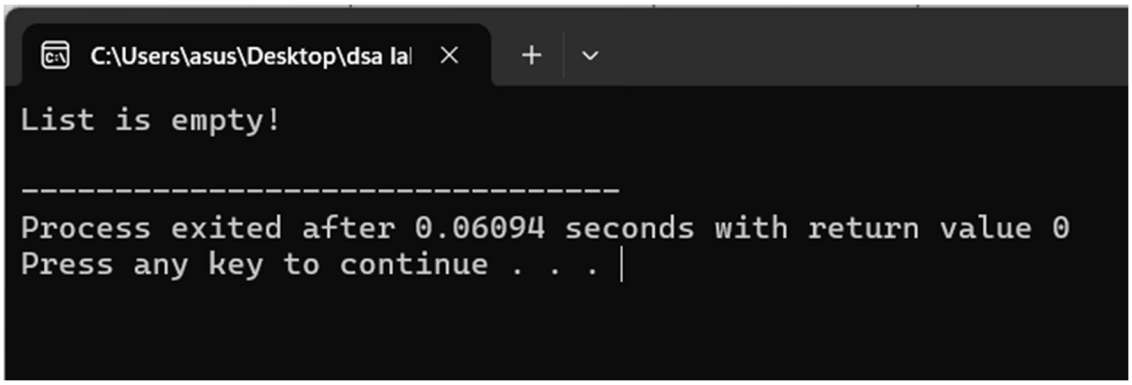
```
    Node* temp = head;

    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;
```

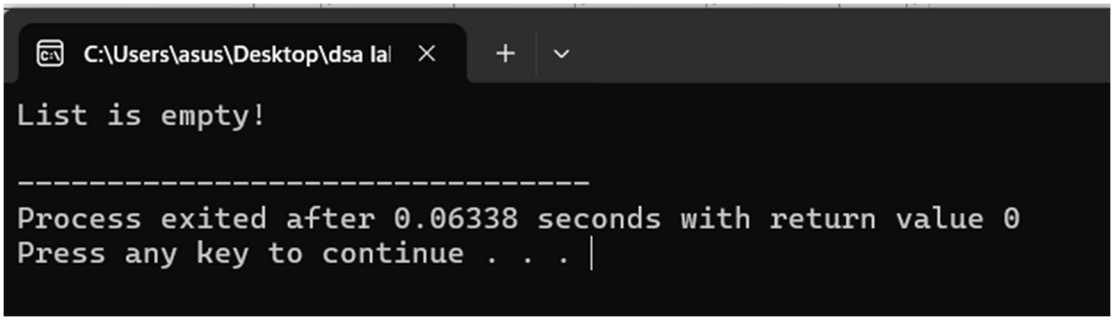
```
struct Node {
    int data;
    Node* next;
};

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;
```

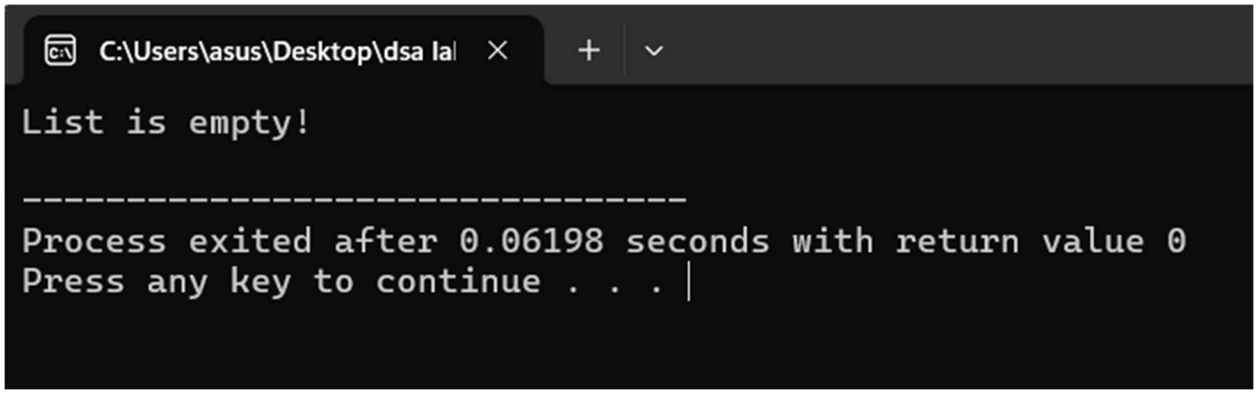
```
    Node* next;
};

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;

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;
}
```

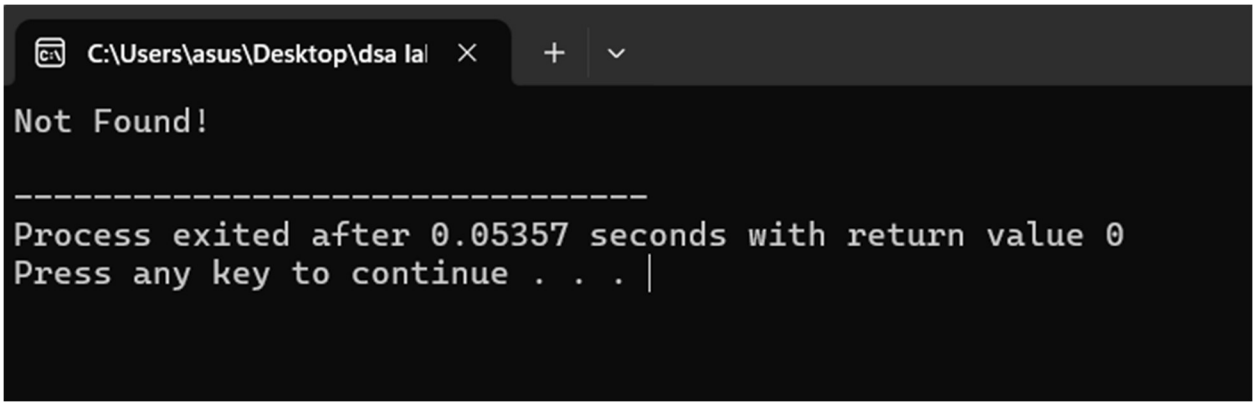
```
int main() {
    Node* head = nullptr;
    deleteMid(head, 2); // Testing on empty list
    return 0;
}
```



2. 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;
```

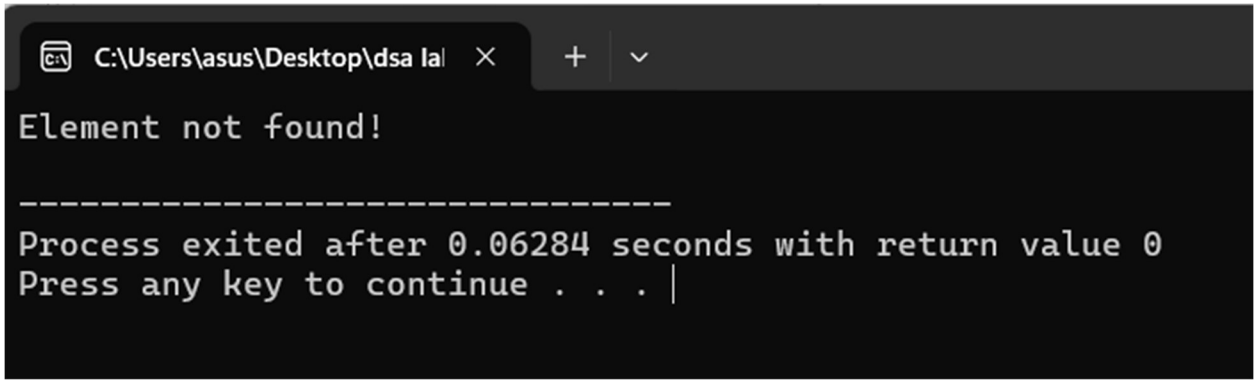
```
    if (search(head, 10)) {
        cout << "Found!" << endl;
    } else {
        cout << "Not Found!" << endl;
    }
    return 0;
}
```



3. 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;
```

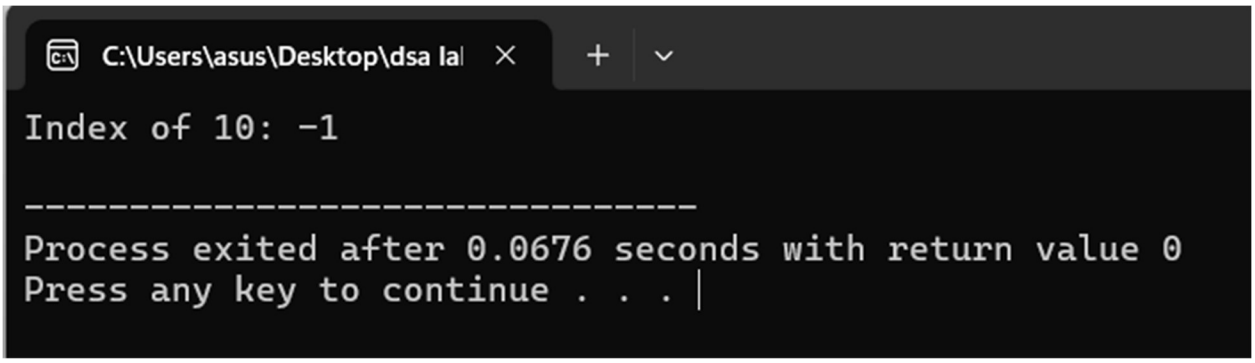
```
}

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



4. 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++;
    }
```

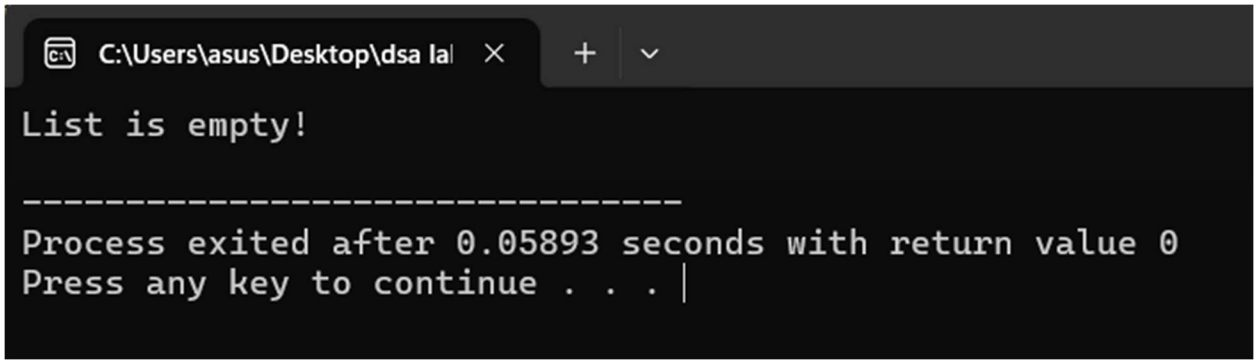
```
    return -1; // Not found
}

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



5. 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;
```

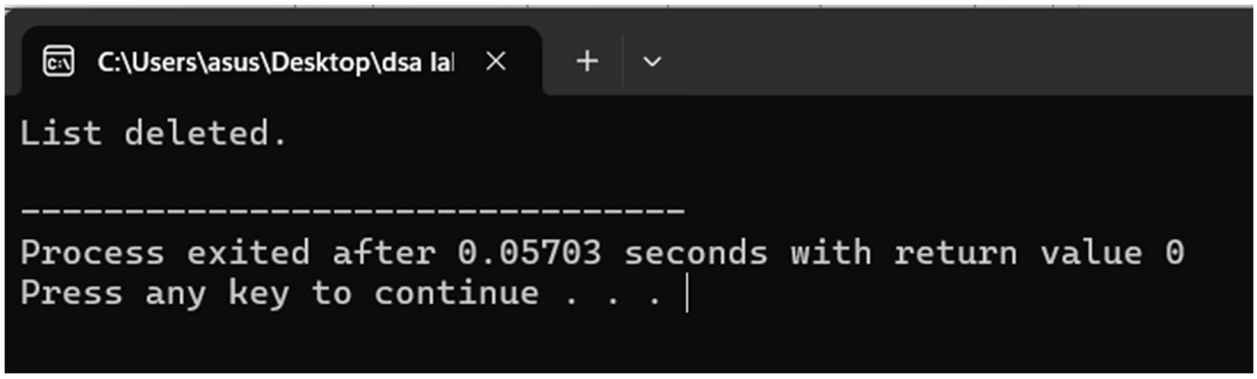
```
    }
    cout << endl;
}

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;
}
```

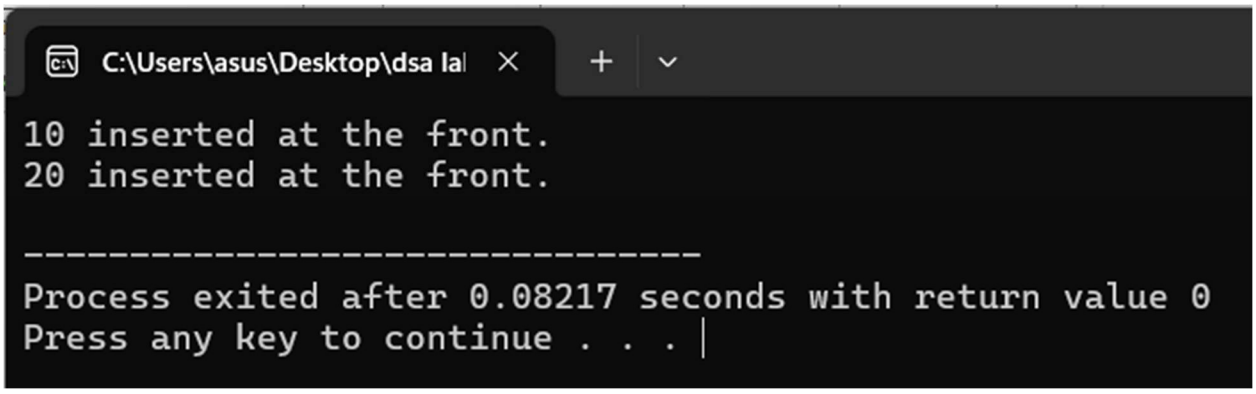


Lab 05 - Double Linked List (DLL)

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;
}
```

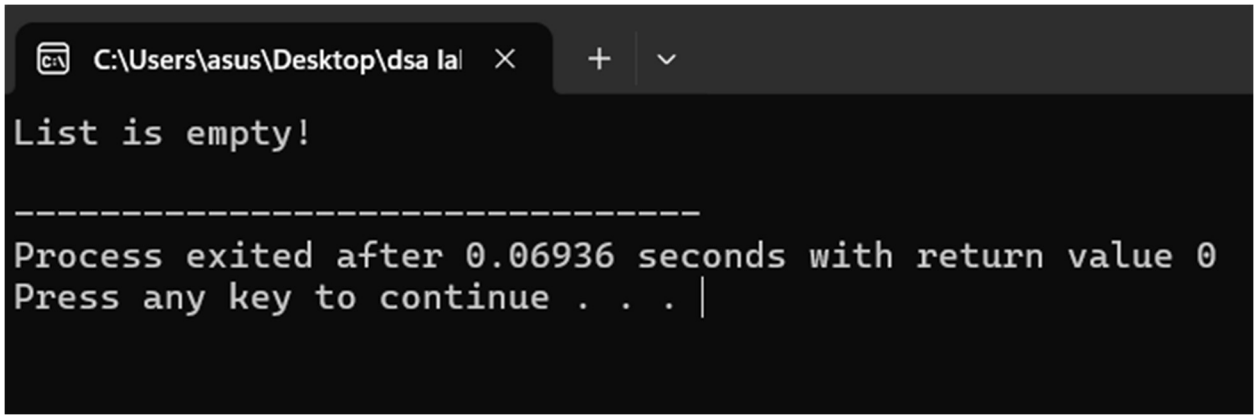


2. Insertion at the Last

```
#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;
    }
```

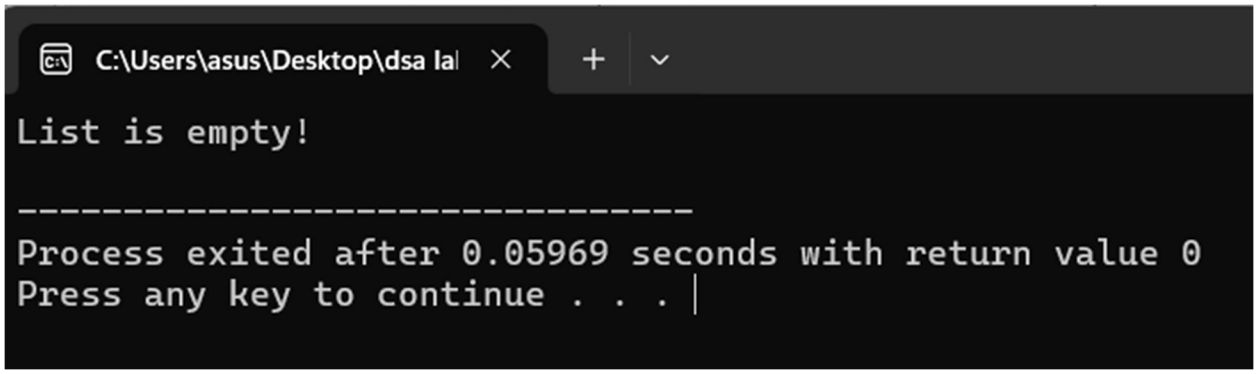
```
    temp->next = newNode;
    newNode->prev = temp;
    cout << value << " inserted at the last." << endl;
}

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



1. **Deletion from the Last**

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

struct Node {
    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;
}
```



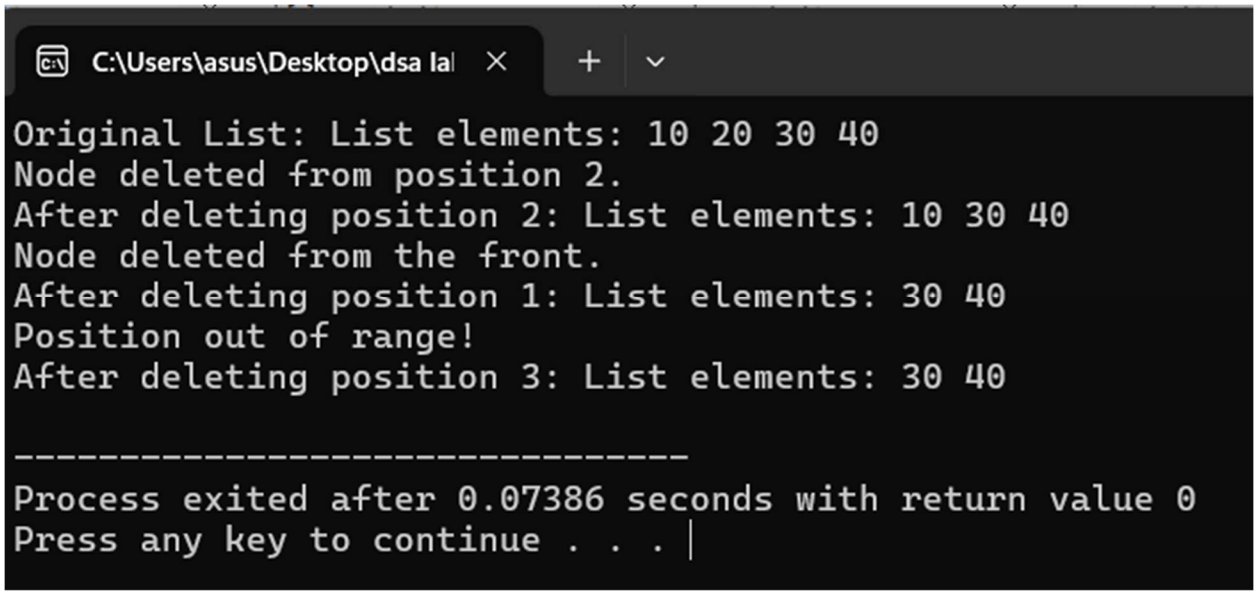
2. Deletion from the Middle (Specific Position)

```
#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);
```

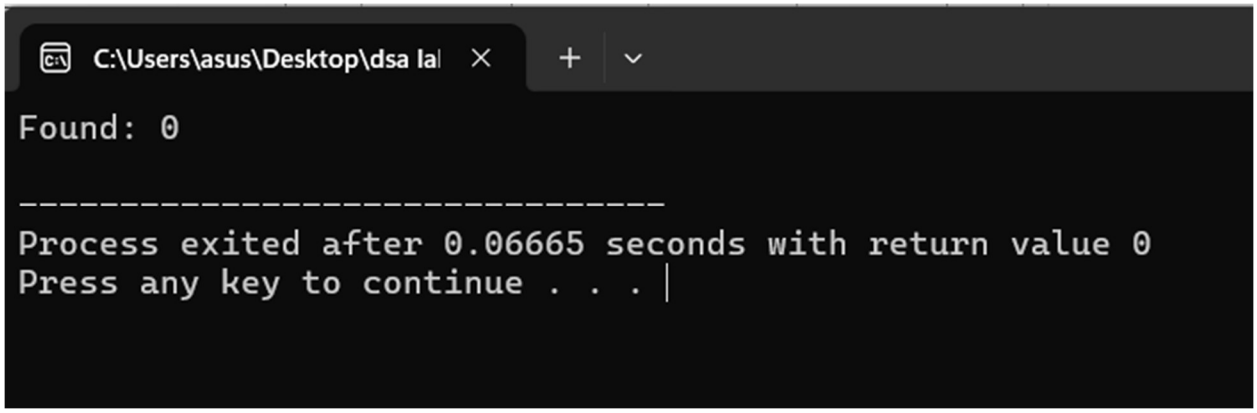
```
    deleteMid(head, 1); // Delete the node at position 1
    cout << "After deleting position 1: ";
    traverse(head);

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

    return 0;
}
```



3. 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) {
```

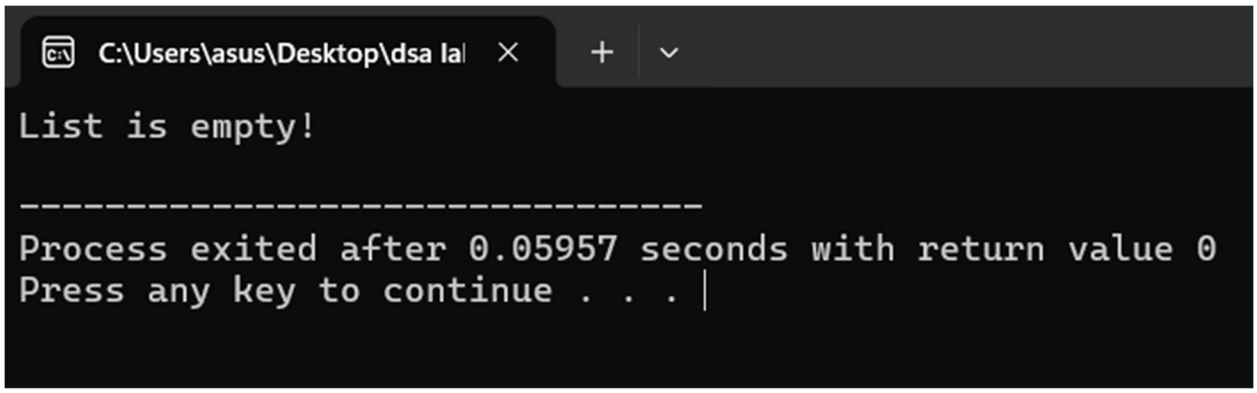
```
        if (temp->data == value) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

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



4. 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;
    }
```

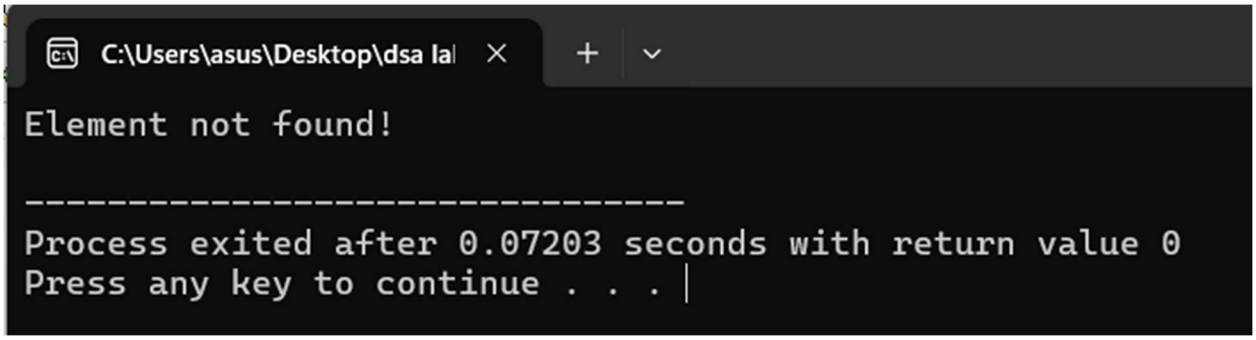
```
    Node* temp = head;
    cout << "List elements: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

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



5. 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

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:

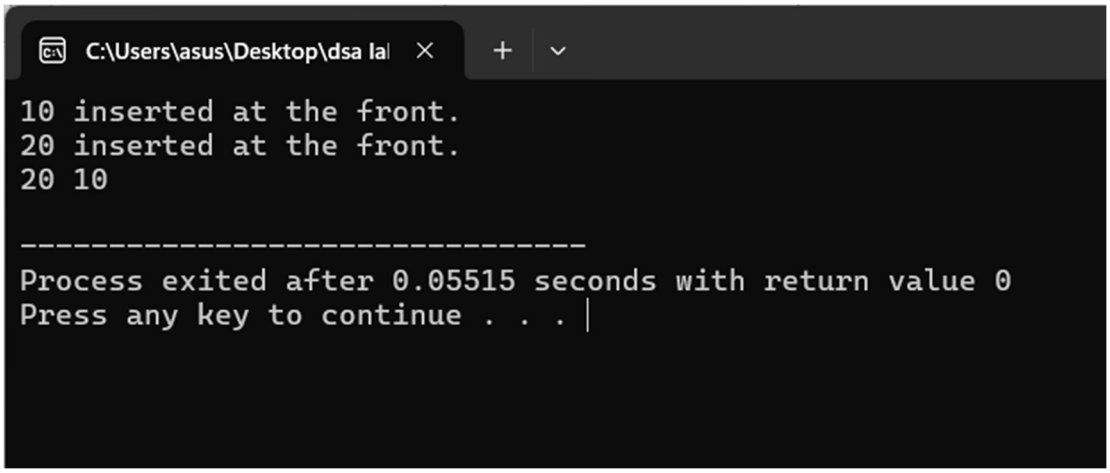
1. **Round Robin Scheduling** in OS.
2. **Circular Buffers** for data handling.
3. **Music Playlists** that loop indefinitely.
4. **Circular Queues** for continuous processing.

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;
    }
```

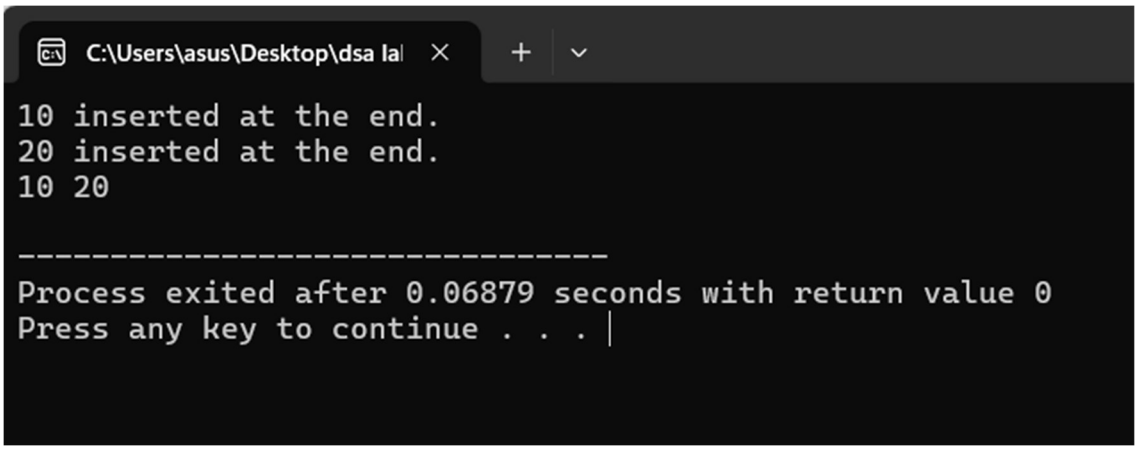
```
    cout << value << " inserted at the front." << endl;
}

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;
}
```

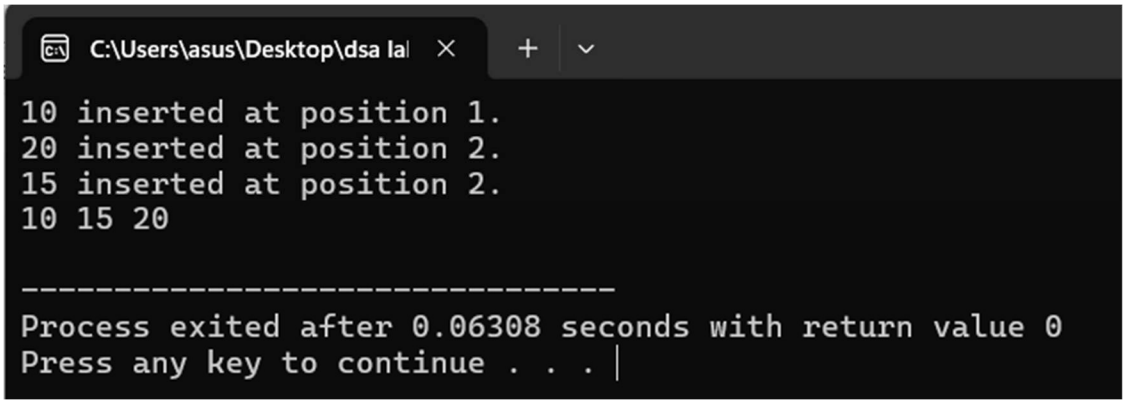


3. Insertion at a Specific Position

```
#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;
}
```

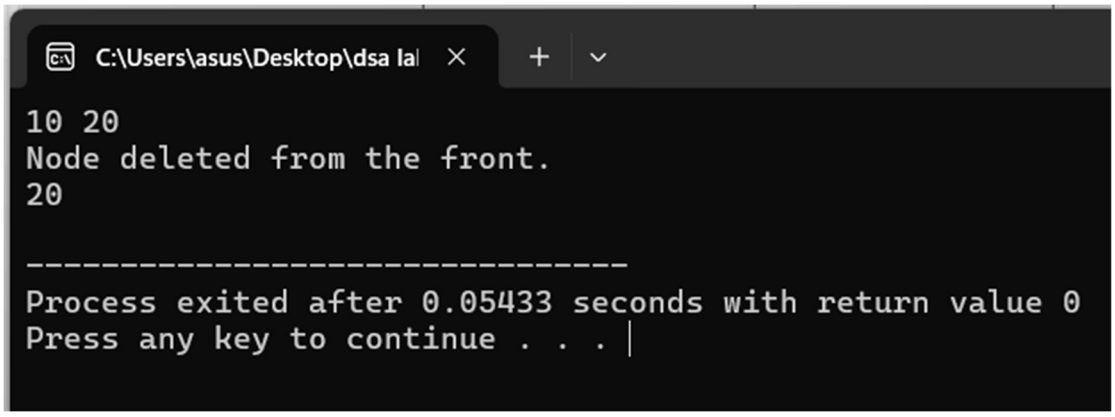


4. Deletion from Front

```
#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;
}
```

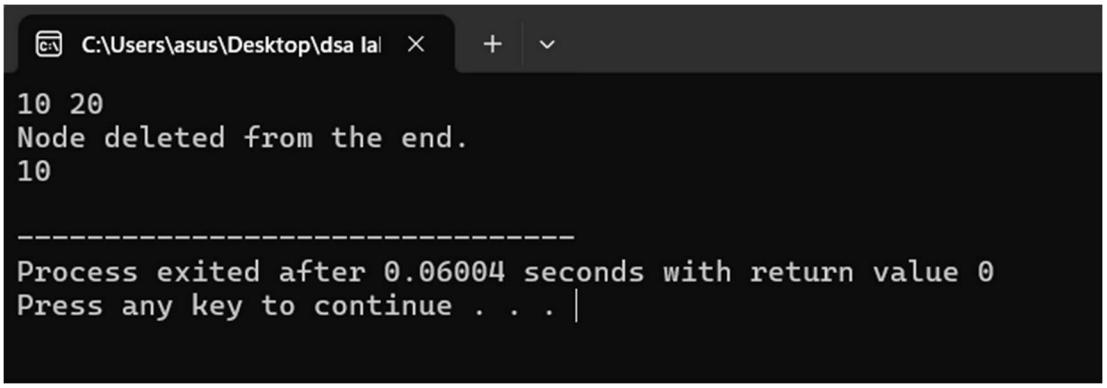


5.Deletion from Last

```
#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;
```

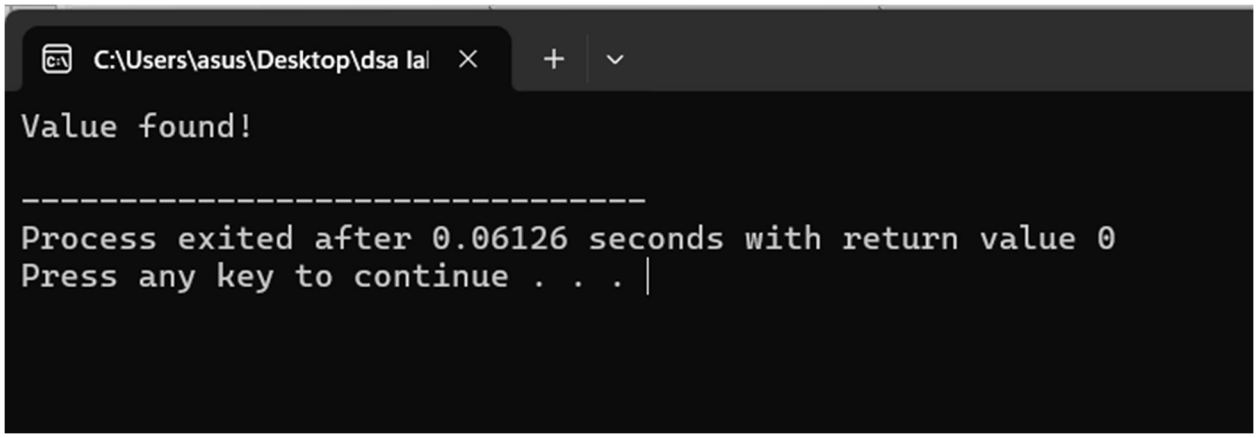
```
    // Example: Inserting some nodes
    Node* newNode = new Node();
    newNode->data = 10;
    head = newNode;
    head->next = head;

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

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

    return 0;
}
```



7.Searching

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

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

bool search(Node* head, int value) {
```

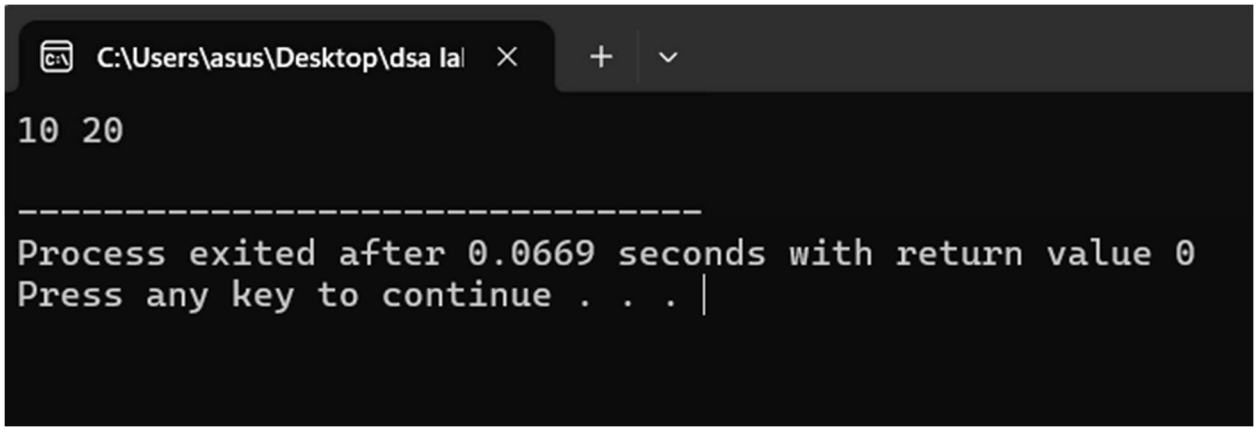
```
    if (head == nullptr) {
        return false;
    }

    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();
```

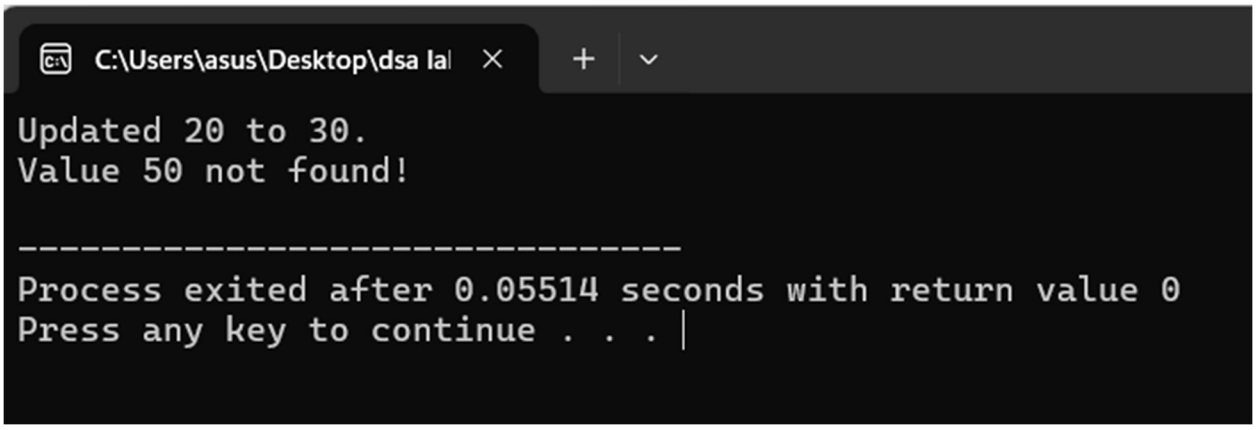
```
    newNode->data = 10;
    head = newNode;
    head->next = head;

    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;
}
```

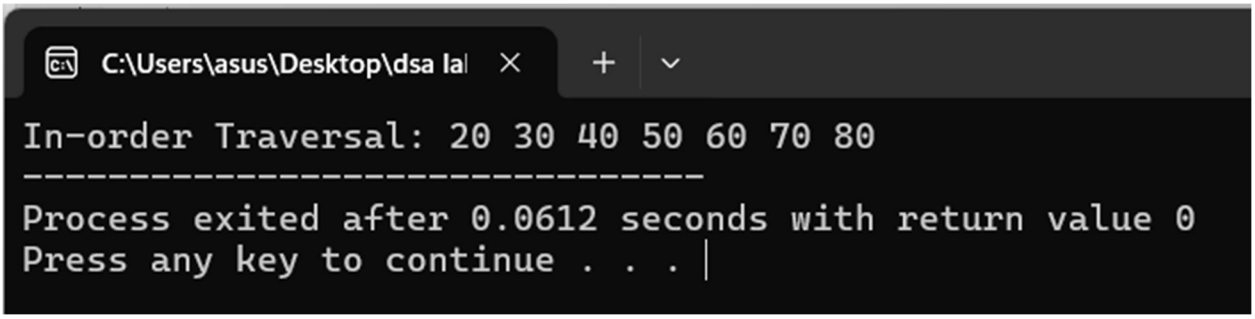


Lab 07 - Binary Search Tree (BST)

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;
}
```



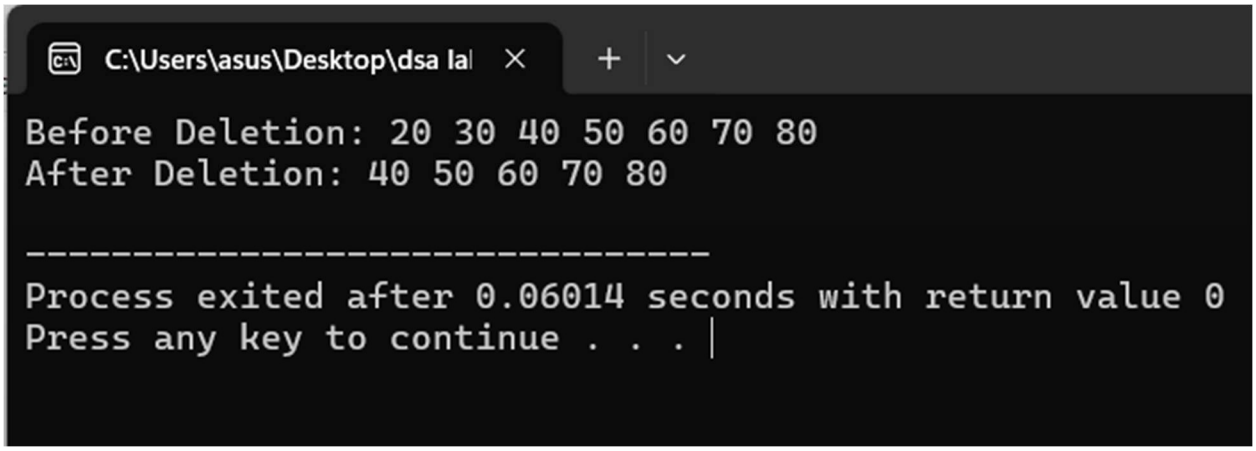
2. Deletion in Binary Search Tree

```
#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

        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;
}
```