The University of Azad Jammu and Kashmir

Name: Zarnab Fatima

Roll NO: 2024-SE-14

Course Title: DSA

Course Code: CS-2101

Submitted To: Engr. Sidra Rafique

Department of Software Engineering

LAB 04

Linked List:

A Linked List is a linear data structure where elements (called nodes) are connected using pointers. Unlike arrays, the elements are not stored at contiguous memory locations.

Structure of a Node

Each node typically contains:

1. Data → Stores the value of the element.

2. Pointer (next) → Points to the next node in the list.

```
+------+------+
| data | next |
+------+------+
```

- Head → Points to the first node.
- Tail / NULL → Last node points to NULL indicating the end.

Types of Linked Lists:

1. Singly Linked List

- Each node points to the next node only.
- Traversal is forward only.

2. Doubly Linked List

- Each node points to both previous and next nodes.
- Traversal is forward and backward.

3. Circular Linked List

- Last node points back to the first node.
- Can be singly or doubly linked.

Advantages

- Dynamic size (can grow/shrink at runtime)
- Easy insertion and deletion (no need to shift elements like arrays)

Disadvantages

- Extra memory for pointer(s)
- Slower access compared to arrays (no direct indexing)

Doubly Linked List (DLL)

A Doubly Linked List (DLL) is a type of linked list in which each node contains a data part and two pointers:

1. *prev* → Points to the previous node

2. *next* → Points to the next node

This allows traversal in both directions (forward and backward), unlike a singly linked list which can only be traversed in one direction.

Structure of a Node

```
+------+-------+------+
| prev | data  | next |
+------+-------+------+
```

- prev → memory address of the previous node (or NULL for the first node)
- data → stores the value
- next → memory address of the next node (or NULL for the last node)

Advantages of Doubly Linked List

Can traverse forward and backward.

Easier to delete a node when a pointer to it is given (no need to traverse from head).

Can efficiently insert at both ends.

Basic Operations

1. Insert at beginning → Update head and prev pointers.
2. Insert at end → Update tail and next pointers.
3. Delete a node → Update prev and next pointers of adjacent nodes.
4. Display forward → Start from head and follow next pointers.
5. Display backward → Start from tail and follow prev pointers.

Code:

INKED.cpp

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

// Node structure
class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

// Doubly Linked List class
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() {
        head = nullptr;
```

Figure 1

LINKED.cpp

```cpp
    DoublyLinkedList() {
        head = nullptr;
        tail = nullptr;
    }

    // 1. Insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        if (!head) { // Empty list
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        cout << val << " inserted at the beginning.\n";
    }

    // 2. Insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!tail) { // Empty list
            head = tail = newNode;
        } else {
            tail->next = newNode;
```

Figure 2

LINKED.cpp

```cpp
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        cout << val << " inserted at the end.\n";
    }

    // 3. Delete from the beginning
    void deleteFromBeginning() {
        if (!head) {
            cout << "List is empty. Nothing to delete.\n";
            return;
        }
        Node* temp = head;
        cout << "Deleting " << temp->data << " from the beginning.\n";
        head = head->next;
        if (head)
            head->prev = nullptr;
        else
            tail = nullptr; // List becomes empty
        delete temp;
    }
```

Figure 3

LINKED.cpp

```cpp
    }

    // 4. Display forward
    void displayForward() {
        if (!head) {
            cout << "List is empty.\n";
            return;
        }
        Node* temp = head;
        cout << "Forward: ";
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Display backward
    void displayBackward() {
        if (!tail) {
            cout << "List is empty.\n";
            return;
        }
        Node* temp = tail;
        cout << "Backward: ";
        while (temp) {
```

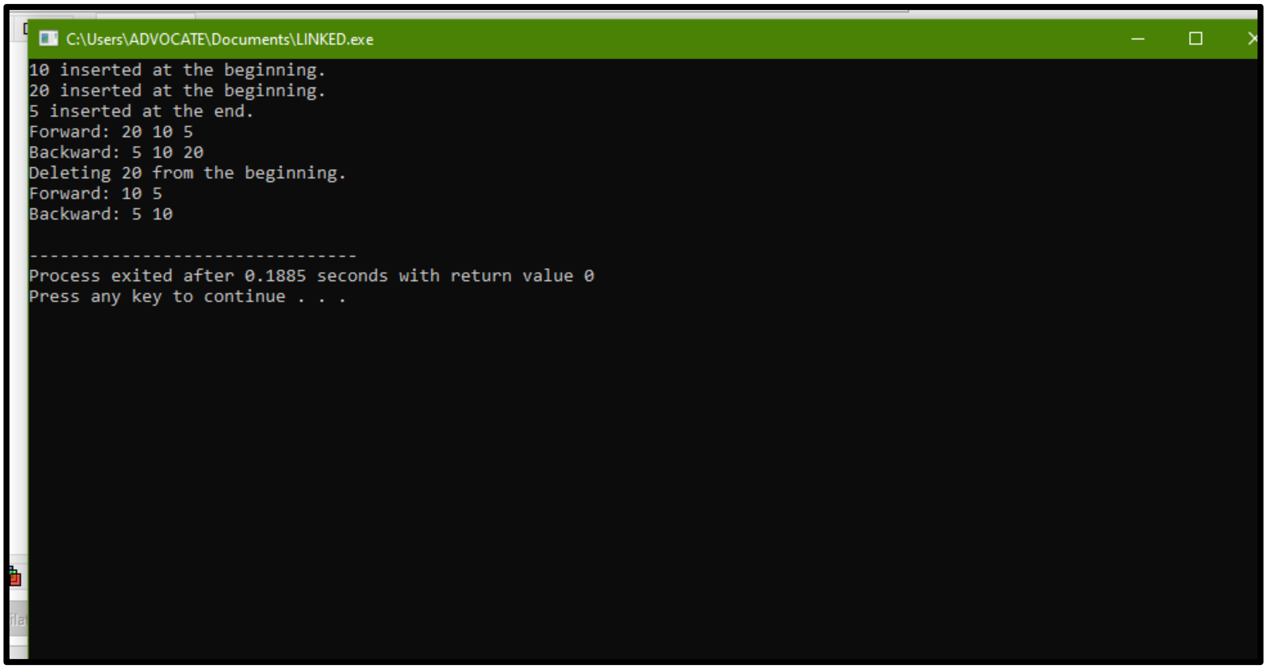
Figure 4

LINKED.cpp

```cpp
        cout << "Backward: ";
        while (temp) {
            cout << temp->data << " ";
            temp = temp->prev;
        }
        cout << endl;
    }
};

// Main function to test the doubly linked list
int main() {
    DoublyLinkedList list;

    list.insertAtBeginning(10);
    list.insertAtBeginning(20);
    list.insertAtEnd(5);
    list.displayForward();
  list.displayBackward();

    list.deleteFromBeginning();
    list.displayForward();
    list.displayBackward();

    return 0;
}
```

Figure 5

Output:



Explanation of Operations

1. Insert at Beginning: Adds a new node at the start and updates head.

2. Insert at End: Adds a new node at the tail and updates tail.

3. Delete from Beginning: Removes the first node and updates head.

4. Display Forward & Backward: Traverses the list using next and prev pointers.

*****THANK YOU*****