Data Structures Lab

Lab # 05

Doubly Linked List

Lab Engineer: Hilal Ahmad

Department of Computer Science,
National University of Computer and Emerging Sciences FAST
Peshawar Campus

Table of Contents

1. Introduction to Doubly Linked List

A **doubly linked list** is a dynamic data structure consisting of nodes. Each node contains:

- **Data**: Stores the value of the node.
- **Previous Pointer**: Points to the previous node in the list.
- **Next Pointer**: Points to the next node in the list.

Differences from Singly Linked List:

- Doubly linked lists allow traversal in both directions (forward and backward).
- Each node has an extra pointer to the previous node, adding some overhead in terms of memory.

2. Doubly Linked List Class Structure

Node Class:

```
class Node {
public:
    int data;
    Node* prev;
    Node* next;

    // Constructor
    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};
```

DoublyLinkedList Class:

```
class DoublyLinkedList {
private:
    Node* head;     // Pointer to the first node
    Node* tail;     // Pointer to the last node

public:
    // Constructor
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    // Destructor
    ~DoublyLinkedList() {
        while (head) {
            deleteHead();
        }
    }

    // Function prototypes
    void insertAtHead(int value);
    void insertAtTail(int value);
    void insertAtPosition(int value, int position);
    void deleteHead();
    void deleteTail();
    void deleteByValue(int value);
    void printListForward();
    void printListBackward();
};
```

3. Operations on Doubly Linked List

3.1 Insertion

We will implement insertion at two positions:

- **At the Head**: Insert a new node at the beginning.
- **At the Tail**: Insert a new node at the end

Insertion at the Head:

```
void DoublyLinkedList::insertAtHead(int value){
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = tail = newNode; // If the list is empty, set newNode as head and tail
    }
    else{
        newNode->next = head; // Point newNode's next to the current head
        head->prev = newNode; // Set current head's prev to newNode
        head = newNode; // Update head to the newNode
    }
}
```

Insertion at the Tail:

```
void DoublyLinkedList::insertAtTail(int value) {
    Node* newNode = new Node(value);
    if (tail == nullptr) {
        head = tail = newNode;          // If list is empty, set newNode as head and tail
    } else {
        newNode->prev = tail;           // Point newNode's prev to the current tail
        tail->next = newNode;           // Set current tail's next to newNode
        tail = newNode;                 // Update tail to the newNode
    }
}
```

3.2 Deletion

We will implement deletion of:

- **By Value**: Remove a node with a specific value.

Deletion by value:

```cpp
void DoublyLinkedList::deleteByValue(int value) {
    if (head == nullptr) return;      // If list is empty, return

    if (head->data == value) {
        deleteHead();
        return;
    }

    if (tail->data == value) {
        deleteTail();
        return;
    }

    Node* temp = head;
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;            // Traverse to the node with the value
    }

    if (temp == nullptr) {
        cout << "Value not found!" << endl;
        return;
    }

    temp->prev->next = temp->next;
    if (temp->next != nullptr) {
        temp->next->prev = temp->prev;
    }
    delete temp;                      // Free the memory of the deleted node
}
```

4. Tasks

Task 1: Deletion of a Node

Write a function to delete a node with a specific value from the doubly linked list. Handle all edge cases, such as deleting the head, tail, or a middle node.

Task 2: Reverse the Doubly Linked List

Write a function to reverse the doubly linked list. Reverse the list by only adjusting the *next* and *prev* pointers of each node.

Task 3: Count the Number of Nodes

Implement a function to count the total number of nodes in the doubly linked list.

Task 4: Find a Node by Value

Write a function to search for a node with a given value in the list. Return the position and value of the node if found, otherwise return NULL.