LAB SESSION 03

Doubly Linked List Operations

A doubly linked list (DLL) is a type of linked data structure where each node stores two pointers, one pointing to the previous node and the other pointing to the next node. This allows movement in both forward and backward directions, offering more flexibility than singly linked lists.

Structure of a Node:

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

1. **Data** – the value stored in the node.
2. **Next pointer** – Address of the next node (NULL for the last node).
3. **Previous pointer** – Address of the previous node (NULL for the first node).

Key Operations

1. **Traversal ((Forward and Backward)**

2. **Insertion**

- **At Beginning** – Link the new node’s next to the current head and set head’s prev to the new node, then update head.
- **At End** – Link the new node’s prev to the last node and last node’s next to the new node.
- **At Position** – Adjust prev and next pointers of surrounding nodes to fit the new node in between.

3. **Deletion**

- **Deletion at Beginning –** Update head to the second node, set its prev to NULL, and free the removed node’s memory.
- **Deletion at End –** Move to the last node, update the second-last node’s next to NULL, and free the last node.
- **Deletion at a Specific Position –** Adjust the next of the previous node and the prev of the next

Table 3.1 Comparison with Singly Linked List

Feature	**Singly Linked List**	**Doubly Linked List**
Memory per node	2 fields	3 fields
Backward traversal	No	Yes
Insertion/deletion	Requires extra steps	Easier (if previous pointer available)
Performance	Less flexible	More flexible

Advantages:

- Traversal possible in both directions.
- Faster insertion/deletion at both ends.

Disadvantages:

- Requires more memory per node.
- Slightly more complex pointer management.

1. Traversal

Forward Traversal

Algorithm:

```
1. Start with a  pointer ptr = head.
2. Repeat until ptr == NULL:
    a. Print ptr → data.
    b. Move ptr = ptr → next.
3. End.
```

Backward Traversal

Algorithm:

```
1. Start with a pointer ptr = tail.
2. Print "Backward: ".
3. Repeat until ptr == NULL:
    a. Print ptr → data followed by " <-> ".
    b. Move ptr = ptr → prev.
4. Print "NULL".
```

2. Insertion

Insertion at Beginning

Algorithm:

```
1. Create a new node newNode.
2. Set newNode → data = value.
3. Set newNode → prev = NULL.
4. Set newNode → next = head.
5.  If head != NULL, set head → prev = newNode.
6. Set head = newNode.
7. End.
```

Insertion at End

Algorithm:

```
1. Create a new node newNode.
2. Set newNode → data = value.
3. Set newNode → next = NULL.
4. If head == NULL:
     a. Set newNode → prev = NULL.
     b. Set head = newNode.
     c. End.
5. Else:
     a. Start ptr = head.
     b. Traverse until ptr → next == NULL.
     c. Set ptr → next = newNode.
     d. Set newNode → prev = ptr.
6. End.
```

Insertion at Position (pos)

Algorithm:

```
1. If pos == 0, call insertAtBeginning(value).

2. Else:
     a. Start ptr = head.
     b. Traverse (pos – 1) times (or until end).
     c. If ptr == NULL, print "Position out of range".
     d. Create a newNode.
     e. Set newNode → data = value.
     f. Set newNode → next = ptr → next.
     g. Set newNode → prev = ptr.
     h. If ptr → next != NULL, set (ptr → next) → prev = newNode.
     i. Set ptr → next = newNode.

3. End.
```

3. Deletion

Deletion at Beginning

Algorithm:

```
1. If head == NULL:
    Print "List Empty" and End.
2. Set temp = head.
3. Set head = head → next.
4. If head != NULL, set head → prev = NULL.
5. Free/Delete temp.
6. End.
```

Deletion at End

Algorithm:

```
7. If head == NULL:
    Print "List Empty" and End.
8. Set ptr = head.
9. Traverse until ptr → next == NULL.
10. If ptr → prev != NULL, set (ptr → prev) → next = NULL.
11. Else set head = NULL.
12. Free/Delete ptr.
13. End.
```

Deletion at Position (pos)

Algorithm:

```
1. If head == NULL:
    Print "List Empty" and End.
2. Set ptr = head.
3. Traverse pos times (or until end).
4. If ptr == NULL:
    Print "Position out of range" and End.
5. If ptr → prev != NULL, set (ptr → prev) → next = ptr → next.
    Else set head = ptr → next.
6. If ptr → next != NULL, set (ptr → next) → prev = ptr → prev.
7. Free/Delete ptr.
8. End.
```

EXERCISES:

1. Implement the above algorithms in C++.
2. Add a search function to find a given value.
3. Write a function to count the total number of nodes in the DLL.
4. Modify insertAtPosition() to handle inserting at the last position without using insertAtEnd().
5. Modify deleteAtPosition() to handle deletion from the last position without calling deleteAtEnd().
6. Implement a function to reverse a doubly linked list.
7. Implement a function to merge two DLLs.
8. Create a menu-driven program to test all operations dynamically.