United International University

CSE 2216: Data Structure and Algorithm lab

Lab 5: Doubly Linked List

06/12/25

Contents

1 Introduction

In our earlier lesson, we studied singly linked lists. In this worksheet, we focus on:

- Doubly Linked Lists
- Basic Circular Linked Lists

These two structures extend the idea of linked lists using different pointer arrangements to improve navigation and build useful data structures.

2 Doubly Linked List (DLL)

A **Doubly Linked List** is a linked list in which each node contains:

1. Data
2. Pointer to the next node
3. Pointer to the previous node

Node Structure

```
struct Node {
    int data;
    Node *prev;
    Node *next;
};
Node *head = nullptr;
```

2.1 Why Doubly Linked Lists?

- Allows movement in both directions.
- Faster deletion of a given node (if pointer is known).
- Useful in navigation systems, undo-redo operations, and various algorithms.

2.2 Diagram

```
NULL <- [prev | data | next] <-> [prev | data | next] <-> [prev | data | next] -> NUL
```

3 Insertion Operations (DLL)

3.1 1. Insert at the Beginning

Steps:

- Create a new node
- Set newNode next = head
- Set head prev = newNode
- Make newNode the head

C++ Code

```cpp
void insertAtBeginning(int value) {
    Node *newNode = new Node();
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = head;

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

3.2 2. Insert at the End

```cpp
void insertAtEnd(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;
}
```

3.3 3. Insert at Any Position

```cpp
void insertAtPosition(int value, int pos) {
    if (pos == 0) {
        insertAtBeginning(value);
        return;
    }

    Node *temp = head;
    for (int i = 0; i < pos - 1 && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) 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;
}
```

4 Deletion Operations (DLL)

4.1 1. Delete from Beginning

```cpp
void deleteBeginning() {
    if (head == nullptr) return;

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

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

    delete temp;
}
```

4.2 2. Delete from End

```cpp
void deleteEnd() {
    if (head == nullptr) return;

    Node *temp = head;

    if (temp->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }

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

    temp->prev->next = nullptr;
    delete temp;
}
```

4.3 3. Delete at Any Position

```
void deleteAtPosition(int pos) {
    if (head == nullptr) return;

    if (pos == 0) {
        deleteBeginning();
        return;
    }

    Node *temp = head;
    for (int i = 0; i < pos && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) return;

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

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

    delete temp;
}
```

5 Traversing a DLL

Forward Traversal

```cpp
void displayForward() {
    Node *temp = head;
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
```

Backward Traversal

```cpp
void displayBackward() {
    if (head == nullptr) return;

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

    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->prev;
    }
    cout << "NULL\n";
}
```

6 Advantages of DLL

- Bidirectional traversal.
- Faster deletion of a known node.
- Useful for complex data structures.

7 Disadvantages

- Uses more memory (extra pointer).
- Slightly more complex to implement.

8 Circular Linked List (CLL)

A **Circular Linked List** is a linked list whose last node points back to the first node.

Diagram

```
[ data | next ] -> [ data | next ] -> [ data | next ]
        ^                                    |
        |____________________________________|
```

9 Basic Circular Linked List Structure

```
struct Node {
    int data;
    Node *next;
};
Node *head = nullptr;
```

10 Insertion at End (CLL)

```
void insertEndCLL(int value) {
    Node *newNode = new Node();
    newNode->data = value;

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

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

    temp->next = newNode;
    newNode->next = head;
}
```

11 Traversal of Circular List

```cpp
void displayCLL() {
    if (head == nullptr) return;

    Node *temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);

    cout << "(back to head)\n";
}
```

12 Applications

- Round-robin scheduling
- Multiplayer games (turn-based loops)
- Repeating playlists
- Handling buffers (e.g., circular buffers)

13 Tasks

Task 1: Singly Linked List (SLL)

Task: Insert at the End — but with a twist.

You are given the following singly linked list:

$$2 \rightarrow 4 \rightarrow 6 \rightarrow \text{NULL}$$

Your task is:

1. Insert the value **8** at the end of the list.
2. Before inserting, apply the following rule:
   - If the last element in the list is **even**, double the value to be inserted.
   - If the last element is **odd**, insert normally.

Hint: The last element is 6 (which is even), so think: what value will actually be inserted?

Output Requirement: Display the final linked list.

Task 2: Doubly Linked List (DLL)

Task: Insert at the Beginning — then swap values.

You are given the following doubly linked list:

$$10 \leftrightarrow 20 \leftrightarrow 30 \leftrightarrow \text{NULL}$$

Your task is:

1. Insert the value **5** at the beginning.
2. Immediately after inserting, **swap the data values** of:
   - the new head node
   - the second node

Important Note: Do **not** change any pointers. Only swap the **data fields**.

Output Requirement: Display both forward and backward traversals of the final list.