National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

School of Electrical Engineering and Computer Science

CS-250: Data Structure and Algorithms

Class: BSEE13

Muhammad Anser Sohaib

367628

Lab 05: Doubly Linked Lists

Date: 26th February, 2025

Time: 02:00 pm – 4:50 pm

Instructor: Ms. Ayesha Sarwer

Lab Engineer: Areeba Rameen

Lab 05: Doubly Linked List

Introduction

This lab will introduce students with the practical implementation of the doubly linked list with its operations.

Objectives

The objective of this lab session is to acquire skills in working with Doubly linked lists.

Tools/Software Requirement

Visual Studio c++/ Code Spaces

Helping Material

Lecture slides, text book

Description

Your task is to implement the following operations of a doubly linked list:

- bool IsEmpty(); // checks whether the list is empty or not. Returns true if empty and false otherwise.
- InsertAtFront(value); // takes input from a user and inserts it at front of the list
- InsertAtEnd(value); //// takes input from a user and inserts it at tail end of the list
- InsertSorted(value); //If we want to maintain a sorted list, we should implement this function
- Search(value); This function shall search value in a list. If found, we will need to store two addresses:
    1. Address of the node in which the searched value is found in a pointer variable named Loc_; we will store NULL in Loc_ in case value is not found.
    2. Address of the node which is logical predecessor of value in a list.

  The Search() provides functionality for other operations such as insertion in a sorted list, deleting a value, modifying a value, printing it etc.
- int DeleteFront(value); Deletes front node of the list and also returns the value stored in it
- int DeleteLast(value); Deletes front node of the list and also returns the value stored in it
- Delete(value); // searches value and then deletes it, if found.

- DestroyList(); // Distroys all nodes of the list leaving the list in empty state.

Declare Node Class: The data structure that will hold the elements of the list is called **Node.** Declare it as follows:

```
class ListNode{
public:
    int data;
    ListNode    *next;
    ListNode    *prev;
};
```

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

// Define the ListNode structure
struct ListNode {
    int data;           // Data stored in the node
    ListNode* next;     // Pointer to the next node
    ListNode* previous; // Pointer to the previous node
};
```

Activity 1) Linked List

Now, declare your main class LinkedList:

```
class DoublyLinkedList{
public:
    ListNode *first; // special variable which stores address of head node.
    ListNode *last; // special variable which stores address of the last node.

    ListNode *PredLoc_; //to be used by Search(value) method to store address of logical predecessor of value in a list.
    ListNode *Loc_; //to be used by Search(value) method to store address of the node containing the searched value in a list. If it is not found it contains NULL.
```

}

Creating a LinkedList

In order to create an empty list, assign NULL value to start pointer variable.

DoublyLinkedList(){

first=NULL; *last=NULL;*

PredLoc_=NULL; *Loc_=NULL;*

}

```
class DoublyLinkedList {
    public:
        ListNode* first;  // Points to the head (first node) of the list
        ListNode* last;   // Points to the tail (last node) of the list
        ListNode* PredLoc_; // Stores the address of the logical predecessor during search
        ListNode* Loc_;    // Stores the address of the node containing the searched value

        // Constructor
        DoublyLinkedList() : first(nullptr), last(nullptr), PredLoc_(nullptr), Loc_(nullptr) {}

        // Destructor to clean up memory
        ~DoublyLinkedList() {
            ListNode* current = first;
            while (current != nullptr) {
                ListNode* temp = current;
                current = current->next;
                delete temp;
            }
        }
```

Activity 2) Inserting value at Front:

First, Reserve space for a new node to be inserted in the list by creating object of class ListNode and storing its address in a temporary pointer variable.

*ListNode *newnode = **new** ListNode();*

Now store value in the data part of the new node: *newnode->data=value;*

Insertion at front of a doubly linked list has two special cases:

1) Insertion into an empty list is a special case; you will have to update both the first and last pointer variables.
   first=newnode; *last=newnode;*

2) Insertion at front of a list is a general case; you will have to update only the pointer variable first.

newnode->next=first;

first->prev=newnode;

first=newnode;

```
// Method to insert a value at the front of the list
void insertAtFront(int value) {
    // Reserve space for the new node
    ListNode* newnode = new ListNode();
    newnode->data = value;
    newnode->next = nullptr;
    newnode->prev = nullptr;

    if (first == nullptr) { // Special case: Insertion into an empty list
        first = newnode;
        last = newnode;
    } else { // General case: Insertion at the front of a non-empty list
        newnode->next = first;
        first->prev = newnode;
        first = newnode;
    }
}
```

Activity 3) Inserting value at Tail End:

After reserving space for the new node and inserting value in its data part, you will have to link it at the end of the list. Insertion at tail end of a doubly linked list has two special cases:

1) Insertion into an empty list is a special case; you will have to update both the first and last pointer variables.
   first=newnode; *last=newnode;*
2) Insertion at tail of a list is a general case; you will have to update only the pointer variable last.

last->next = newnode;

newnode->prev=last;

last=newnode;

```
// Method to insert a value at the tail of the list
void insertAtTail(int value) {
    // Reserve space for the new node
    ListNode* newnode = new ListNode();
    newnode->data = value;
    newnode->next = nullptr;
    newnode->prev = nullptr;

    if (first == nullptr) { // Special case: Insertion into an empty list
        first = newnode;
        last = newnode;
    } else { // General case: Insertion at the tail of a non-empty list
        last->next = newnode;
        newnode->prev = last;
        last = newnode;
    }
}
```

Activity 4) Search a Value Function

- This function shall search a value in a list. If found, we will need to store two addresses:
  1. The Address of the node in which the searched value is found. We shall store it in a pointer variable named Loc_. In case the s earched value is not found in the list, we will store NULL in Loc_.
  2. Moreover, we will store the address of the logical predecessor node of the value we are searching for. For this purpose, we shall use a pointer named PLoc.

Activity 5) Search value in a list

Void search(value){

Initialize loc & ploc

Loc= address of head node

Ploc = address of logical predecessor of head node. Note that the first node has no predecessor. Thus, always initialize Ploc with NULL value.

For the moment assume that we are maintaining a list sorted in ascending order. Search value until we reach end of the list or logical position of the value is passed.

While (loc!=NULL and loc.data < value){

Advance both ploc and loc

}

If(loc!null & loc.data!=value)

Loc=null; //as value is not found so set loc equal to null.

```
// Search method
void search(int value) {
    // Initialize Loc_ and PredLoc_
    Loc_ = first;       // Start from the head node
    PredLoc_ = nullptr; // The first node has no predecessor

    // Traverse the list until we reach the end or pass the logical position of the value
    while (Loc_ != nullptr && Loc_->data < value) {
        PredLoc_ = Loc_;       // Advance PredLoc_ to the current node
        Loc_ = Loc_->next;     // Advance Loc_ to the next node
    }

    // Check if the value is found
    if (Loc_ != nullptr && Loc_->data == value) {
        cout << "Value " << value << " found in the list." << endl;
    } else {
        Loc_ = nullptr; // Value not found, set Loc_ to nullptr
        cout << "Value " << value << " not found in the list." << endl;
    }
}
```

After execution of search(value) method, there are four possible combinations of loc and ploc

Table 1: Possible Values in Loc and Ploc after Call to Search() and their Interpretation

Ploc	**Loc**	**Interpretation**
Null	Null	Value not found, and its logical position is at the front of the list
Null	Non-null	Value found in the head node of the list
Non-null	Non-null	Value found but node in the head node. As ploc is non-null, it might be in any node other than the head node
Non-null & ploc=last	Null	Value not found. Its logical position is at the end of the list.
Non-null	Null	Value not found. Its logical position is somewhere after first and before last node.

Activity 6) Insertion in a Sorted List

For the moment, assume duplications are not allowed in the list. You have to insert value after call to search function by considering the above mentioned four possible combinations of loc and ploc pointer variable. This method should insert new node in the sorted list for all the three special cases:

1) Insertion at front of the list. This should be done when ploc=NULL after search
2) Insertion at tail end of the list. This should be done when ploc=last after search
3) General case insertion anywhere after first node and before last node.

```
InsertSorted(value){
    Search(value)
    if (value already exists)
        Return without insertion and print a message
    else {
        if (position  of value is as head node)
            Insert value at front.
        else if (position  of value is as last node)
            Insert value at tail end.
        else {   //insert after ploc.
            Newnode->next= ploc->next;
            Newnode->prev= ploc;
            Ploc->next->prev= newnode;
            Ploc->next = newnode;
        }
    }

}
```

```
void InsertSorted(int value) {
    // Search for the value to determine insertion position
    search(value);

    // If the value already exists, do not insert
    if (Loc_ != nullptr) {
        cout << "Value " << value << " already exists in the list. No duplicates allowed." << endl;
        return;
    }
    // Create a new node
    ListNode* newNode = new ListNode{value, nullptr, nullptr};
    // Case 1: Insertion at the front of the list (Ploc_ == nullptr)
    if (PredLoc_ == nullptr) {
        newNode->next = first;
        if (first != nullptr) {
            first->prev = newNode;
        }
        first = newNode;
        // Update last pointer if the list was empty
        if (last == nullptr) {
            last = newNode;
        }
    }
    // Case 2: Insertion at the tail of the list (Ploc_ == last)
    else if (PredLoc_ == last) {
        PredLoc_->next = newNode;
        newNode->prev = PredLoc_;
        last = newNode;
    }
```

```
    // Case 3: General case (insert after Ploc_)
    else {
        newNode->next = PredLoc_->next;
        newNode->prev = PredLoc_;
        if (PredLoc_->next != nullptr) {
            PredLoc_->next->prev = newNode;
        }
        PredLoc_->next = newNode;
    }

    cout << "Value " << value << " inserted into the list." << endl;
}
```

Activity 7) Delete Front Node

If(list is not empty){

*ListNode *temp = first;*

first = first->next;

first->prev=NULL;

delete temp;

}

```
void DeleteFrontNode() {
    if (first == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    ListNode* temp = first;
    first = first->next;

    if (first != nullptr) {
        first->prev = nullptr;
    } else {
        last = nullptr; // List becomes empty
    }

    delete temp;
    cout << "Front node deleted." << endl;
}
```

Activity 8) Delete last Node:

If(list is not empty){

*ListNode *temp = last;*

last = last->prev;

last->next=NULL;

delete temp;

}

```
void DeleteLastNode() {
    if (last == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    ListNode* temp = last;
    last = last->prev;

    if (last != nullptr) {
        last->next = nullptr;
    } else {
        first = nullptr; // List becomes empty
    }

    delete temp;
    cout << "Last node deleted." << endl;
}
```

Activity 9) Delete a Value

Find value using search method and if a node containing the searched value is found, then delete that node from the linked list. Also, free the allocated memory.

Delete(value){

//if empty return

Search(value)

If(value is found){ //check loc

If(value is in the head node){

//delete head node and free memory

}

Else if (value is in the last node)

//delete last node and free memory

Else {

//delete the node using ploc

ploc->next = loc->next;

loc->next->prev=ploc;

delete loc;

}

}

```
void Delete(int value) {
    if (first == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }

    // Search for the value to locate the node
    search(value);

    // If the value is not found, return
    if (Loc_ == nullptr) {
        cout << "Value " << value << " not found in the list." << endl;
        return;
    }

    // Case 1: Delete the head node
    if (Loc_ == first) {
        DeleteFrontNode();
    }
    // Case 2: Delete the last node
    else if (Loc_ == last) {
        DeleteLastNode();
    }
    // Case 3: Delete a node in the middle
    else {
        Loc_->prev->next = Loc_->next;
        Loc_->next->prev = Loc_->prev;
        delete Loc_;
        cout << "Value " << value << " deleted from the list." << endl;
```

Activity 10) Destroy a Linked List:

This method should destroy all nodes of a linked list making it empty. It should also free space allocated for all the nodes.

```
void DestroyList() {
    ListNode* current = first;
    while (current != nullptr) {
        ListNode* temp = current;
        current = current->next;
        delete temp;
    }

    first = nullptr;
    last = nullptr;
    cout << "Linked list destroyed. All nodes deleted." << endl;
}
```

Hint: Save address of current head node in a temporary pointer variable. Advance start variable to second node so that it becomes new head node. Then, delete current head node using temporary pointer variable.

Lab Activities

1. Write a function which rearranges order of the entire doubly linked list by reversing it.

```
// Function to reverse a doubly linked list
ListNode* reversingDoublyLinkedList(ListNode* head) {
    if (head == nullptr) {
        return nullptr; // Empty list, nothing to reverse
    }
    ListNode* current = head;
    ListNode* temp = nullptr;
    // Traverse the list and swap next and previous pointers for each node
    while (current != nullptr) {
        temp = current->previous;       // Store the previous pointer in a temporary variable
        current->previous = current->next; // Swap previous and next pointers
        current->next = temp;           // Assign the stored previous pointer to next
        current = current->previous;    // Move to the next node (which is now in previous)
    }
    // After the loop, temp points to the new head (last node of the original list)
    if (temp != nullptr) {
        head = temp->previous; // Update the head to point to the new first node
    }
    return head;
}
```

```
@Anser2 →/workspaces/DSA/DoublyLinkedList (main) $ g++ DoublyLinkedList.cpp -o dll
@Anser2 →/workspaces/DSA/DoublyLinkedList (main) $ ./dll
Original list: 1 2 3 4
Reversed list: 4 3 2 1
```

2. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

```
// Function to search and replace nodes
ListNode* searchAndReplace(ListNode* head) {
    if (head == nullptr) {
        cout << "The list is empty!" << endl;
        return head;
    }
    int a, b;
    cout << "Enter the first value to search: ";
    cin >> a;
    cout << "Enter the second value to search: ";
    cin >> b;
    ListNode* current = head;
    ListNode* nodeA = nullptr; // Node containing value 'a'
    ListNode* nodeB = nullptr; // Node containing value 'b'

    // Step 1: Find the node with value 'a'
    while (current != nullptr && current->data != a) {
        current = current->next;
    }
    if (current != nullptr) {
        nodeA = current;
    }
// Step 2: Find the node with value 'b'
current = head; // Reset current to start from the head again
while (current != nullptr && current->data != b) {
    current = current->next;
}
if (current != nullptr) {
    nodeB = current;
}

// Step 3: Check if both nodes were found
if (nodeA == nullptr || nodeB == nullptr) {
    cout << "One or both values not found in the list." << endl;
    return head;
}

// Step 4: Handle edge case where nodeA and nodeB are the same
if (nodeA == nodeB) {
    cout << "Both values refer to the same node. No swap needed." << endl;
    return head;
}
```

```
// Step 5: Swap the nodes
// Update previous pointers
if (nodeA->previous != nullptr) {
    nodeA->previous->next = nodeB;
} else {
    head = nodeB; // If nodeA was the head, update head
}
if (nodeB->previous != nullptr) {
    nodeB->previous->next = nodeA;
} else {
    head = nodeA; // If nodeB was the head, update head
}

// Update next pointers
if (nodeA->next != nullptr) {
    nodeA->next->previous = nodeB;
}
if (nodeB->next != nullptr) {
    nodeB->next->previous = nodeA;
}

// Swap the previous and next pointers of nodeA and nodeB
ListNode* tempNext = nodeA->next;
ListNode* tempPrev = nodeA->previous;
nodeA->next = nodeB->next;
nodeA->previous = nodeB->previous;
nodeB->next = tempNext;
nodeB->previous = tempPrev;

return head;
}
```

OUTPUT:

```
@Anser2 →/workspaces/DSA/DoublyLinkedList (main) $ g++ DoublyLinkedList.cpp -o dll
@Anser2 →/workspaces/DSA/DoublyLinkedList (main) $ ./dll
Original list: 1 2 6 7 4
Enter the first value to search: 2
Enter the second value to search: 7
List after swapping: 1 7 6 2 4
Reversed list: 4 2 6 7 1
```

Deliverables

Compile a single word document by filling in the solution part and submit this Word file on LMS. The name of word document should follow this format. i.e. **YourFullName(reg)_Lab#.** This lab grading policy is as follows: The lab is graded between 0 to 10 marks. The submitted solution can get a maximum of 5 marks. At the end of each lab or in the next lab, there will be a viva related to the tasks. The viva has a weightage of 5 marks. Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your complete Word document to get your work graded. You must also submit this Word document on the LMS.

Note: Students are required to upload the lab on LMS before deadline.

Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks.