CC-213L

Data Structures and Algorithms

Laboratory 07

Doubly Linear Linked List

Version: 1.0.0

Release Date: 30-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
    - Pointers and Dynamic Memory Allocation
    - Self-Referential Objects
        - Representation
        - Implementation
    - Doubly LinkedList
        - Insert Node
        - Delete Node
- Activities
    - Pre-Lab Activity
        - Task 01: Doubly LinkedList Implementation

Learning Objectives:

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Doubly LinkedList

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Tahir Mustafvi	bcsf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

Background and Overview

Pointers and Dynamic Memory Allocation

Pointers and dynamic memory allocation are important concepts in programming, particularly in languages like C and C++. Pointers allow you to work with memory addresses, while dynamic memory allocation allows you to manage memory at runtime.

Pointers:

A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access the value of the variable stored at that address. Pointers are often used for various purposes, such as dynamically allocated memory, working with arrays, and passing functions as arguments.

```cpp
#include <iostream>

int main() {
    int x = 10;
    int* ptr;  // Declare a pointer to an int
    ptr = &x; // Assign the memory address of x to ptr

    std::cout << "The value of x: " << x << std::endl;
    std::cout << "The value pointed to by ptr: " << *ptr << std::endl;

    *ptr = 20; // Modify the value through the pointer
    std::cout << "The updated value of x: " << x << std::endl;

    return 0;
}
```

Figure 1(Pointers)

Explanation:

In this example, ptr is a pointer to an integer, and it is assigned the memory address of the variable x. You can access and modify the value of x through the pointer using the dereference operator (*ptr).

Select Microsoft Visual Studio Debug Console

```
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

Figure 2(Output)

Dynamic Memory Allocation

Dynamic memory allocation allows you to allocate memory for variables at runtime. In C++, you can use new and delete operators to allocate and deallocate memory for objects on the heap.

```cpp
#include <iostream>

int main() {
    int* dynamicArray = new int[5]; // Allocate an array of 5 integers

    for (int i = 0; i < 5; i++) {
        dynamicArray[i] = i * 10;
    }

    for (int i = 0; i < 5; i++) {
        std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
    }

    delete[] dynamicArray; // Deallocate the memory

    return 0;
}
```

Figure 3(Dynamic Memory Allocation)

Explanation:

In this example, dynamicArray is allocated on the heap with space for 5 integers. After using it, it is essential to deallocate the memory using delete[] to prevent memory leaks.

Note: In modern C++ (C++11 and later), it is recommended to use smart pointers like std::unique_ptr and std::shared_ptr for better memory management, as they automatically handle memory deallocation.

```
dynamicArray[0] = 0
dynamicArray[1] = 10
dynamicArray[2] = 20
dynamicArray[3] = 30
dynamicArray[4] = 40
```

Figure 4(Output)

Deallocation of Memory:

Memory should be properly deallocated. In case of not properly deallocating memory, every time your program run causes memory leak that is very critical problem.

```cpp
int main()
{
    int**** ptr = new int***;
    *ptr = new int**;
    **ptr = new int*;
    ***ptr = new int;
    ****ptr = 10;
    cout << ****ptr << endl;
}
```

Figure 5(Multiple indirection)

Explanation:

In above example a **ptr** is pointer and its type is pointer to pointer to pointer to integer. Output of above code is below

```
10
```

Figure 6(Output)

Deallocation of memory should be done properly.

```
    delete*** ptr;
    delete** ptr;
    delete* ptr;
    delete ptr;
    ptr = nullptr; // avoid dangling pointer
}
```

Figure 7(Deallocation)

Self-Referential Objects:

Classes that have capability to refer to their own types of objects are called **Self Referential Classes/Structs.** Objects of such classes are called self-referential Objects.

Self-referential structure in C++ are those structure that contains one or more than one pointer as their member which will be pointing to the structure of the same type. In simple words, a structure that is pointing to the structure of the same type is known as a self-referential structure.

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

Figure 8(Self Referential Objects)

Explanation:

In Figure 8 we have declared a struct Node. It has three data members info, next and prev;

Info Represents the information data part. Enables the object to store relevant information in it. There can be more than one identifier of same/different datatypes depending upon the application /situation.

next and prev Represent the link part. Enables the object to a self-referential object. There can be more than one such references used for different purposes in different applications /situations.

```
9   int main()
10  {
11
12      Node a, b,c, d, e;  // allocated on stack memory portion;
13      a.info = 1;
14      b.info = 2;
15      c.info = 3;
16      d.info = 4;
17      e.info = 5;
18      cout << a.info << " " << b.info << " " << c.info << " " << d.info << " " << e.info << endl;
19  }
```

Figure 9(Node objects)

Explanation:

At line 12 we have declared five Node objects and next lines we have initialized their info data members with proper values. At line 18 we have displayed them.

```
Select Microsoft Visual Studio Debug Console
1 2 3 4 5
```

Figure 10(Output)

```
a.next = &b;
b.next = &c;
c.next = &d;
d.next = &e;
b.prev = &a;
c.prev = &b;
d.prev = &c;
e.prev = &d;
e.next = a.prev = nullptr;
```

Figure 11(Double Link)

Explanation:

In Figure 11 we have initialized each Node next and previous pointer with the addresses of other nodes such that each node can refers a same node in sequence to a next node as well as previous Node.

```
    Node* head = &a;
    cout << "a : " << a.info << endl;
    cout << "b : " << a.next->info << endl;
    cout << "c : " << a.next->next->info << endl;
    cout << "d : " << a.next->next->next->info << endl;
    cout << "e : " << a.next->next->next->next->info << endl;
}
```

Figure 12(Double Links)

Output:

```
Select Microsoft Visual Studio Debug Console
a : 1
b : 2
c : 3
d : 4
e : 5
```

Figure 13(Output)

```
    cout << "a : " << e.prev->prev->prev->prev->info << endl;
    cout << "b : " << e.prev->prev->prev->info << endl;
    cout << "c : " << e.prev->prev->info << endl;
    cout << "d : " << e.prev->info << endl;
    cout << "e : " << e.info << endl;
}
```

Figure 14(Previous Link)

Explanation:

In Figure 14 we have access info of a, b, c, d and e nodes through previous links of each node. This is the facility of Double links.

Some Interesting Scenarios:

```
    Node* head = &a;
    cout << head->next->next->prev->next->next->prev->info << endl;
    head->next->next->prev->next->info = head->next->next->prev->next->info;
    cout << head->next->next->prev->next->info << endl;
    cout << (*(*((*head).next)).next->prev->next).info << endl;
```

Figure 15(Doubly Link)

Doubly Linear LinkedList

A doubly linked list is a data structure used in computer science and programming for organizing and storing a collection of elements. It is similar to a singly linked list, but with a key difference: each node in a doubly linked list contains not only a reference to the next node (like in a singly linked list) but also a reference to the previous node. This bidirectional linkage allows for more versatile operations compared to a singly linked list.

Here are the key characteristics of a doubly linked list:

1. **Nodes**: Each element in a doubly linked list is stored in a node. Each node contains data and two references or pointers: one pointing to the next node in the list (often called "next" or "forward" pointer), and the other pointing to the previous node (often called "prev" or "backward" pointer).

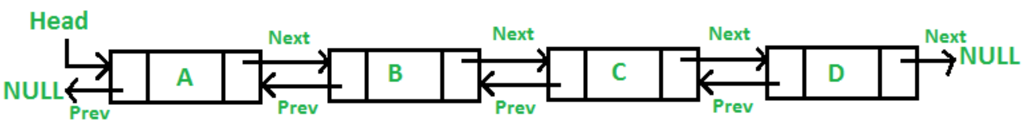

Figure 16(Double LinkedList)

2. **Traversal:** You can traverse a doubly linked list in both directions, forward and backward, using the next and prev pointers. This makes it more flexible for certain operations that require moving in both directions, such as inserting or deleting elements.
3. **Insertion and Deletion:** Inserting and deleting nodes in a doubly linked list is generally more efficient than in a singly linked list because you can access the previous node directly. In a singly linked list, to delete a node, you often need to traverse the list from the beginning to find the previous node, which takes O(n) time in the worst case. In a doubly linked list, this can be done in O (1) time.

Doubly linked lists are commonly used in scenarios where efficient insertions and deletions are required, and you need bidirectional traversal, such as in certain types of data structures like double-ended queues (deque) or when implementing certain algorithms like LRU (Least Recently Used) caches. However, they require more memory than singly linked lists due to the additional backward pointers for each node.

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

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

Figure 17(Doubly LinkedList Node)

Explanation:

At Line 2 we have declared a class Node. It has data, prev and next node.

```
11 int main()
12 {
13     Node* head = new Node(10);
14     head->next = new Node(20);
15     head->prev = nullptr;
16     head->next->prev = head;
17     cout << head->data << endl; // 10
18     cout << head->next->data << endl; //20;
19     cout << head->next->prev->next->data << endl;  //20
20     cout << head->next->prev->data << endl; //10
21
22     delete head->next;
23     delete head;
24     head = nullptr;
25
26 }
```

Figure 18(Insertion in LinkedList)

Activities

Pre-Lab Activities:

Task 01: Doubly Linear LinkedList implementation

Declare these two classes and add public function definition and implementations for a working doubly linear LinkedList

```cpp
// Forward declaration of template class List
template<class T>
class DList;

template<class T>
class DNode {
    friend DList<T>;
    T info;
    DNode<T>* next;
    DNode<T>* prev;
    // Additional methods as required
};

template<class T>
class DList {
    DNode<T>* head; // Pointer to head node
    DNode<T>* tail; // Pointer to tail node

public:
    // Constructor - initializes an empty list
    DList();

    // Destructor - releases all nodes
    ~DList();

    // Copy Constructor - creates deep copy of list
    DList(const DList<T>& other);

    // Insert at the head of the list
    void insertAtHead(T value);

    // Insert at the tail of the list
    void insertAtTail(T value);

    // Delete node at head, returns true if successful
    bool deleteAtHead();

    // Delete node at tail, returns true if successful
    bool deleteAtTail();
```

```
// Print all nodes in list
    void printList();

    // Returns pointer to nth node or last node if n exceeds list length
    DNode<T>* getNode(int n);

    // Insert new node after node with info == key, returns true if inserted
    bool insertAfter(T value, T key);

    // Insert new node before node with info == key, returns true if inserted
    bool insertBefore(T value, T key);

    // Delete node before node with info == key, returns true if deleted
    bool deleteBefore(T key);

    // Delete node after node with info == key, returns true if deleted
    bool deleteAfter(T key);

    // Returns the total number of nodes in the list
    int getLength();

    // Searches for node with value x, returns pointer to first occurrence
    DNode<T>* search(T x);
};
```