EC-200 Data Structures

LAB MANUAL # 05

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

# Lab Manual # 05

# Circular Linked List Implementation

# Lab Objective:

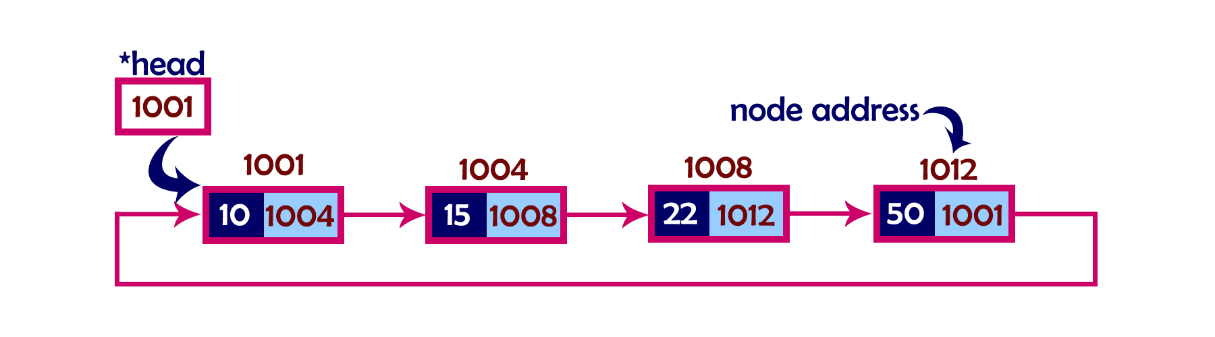
To implement Circular Linked List ADT functions.

# Lab Description:

In single linked list, every node points to its next node in the sequence and the last node points NULL. However, a **circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.**

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

## Circular Linked List:



In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

## Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

1. Create a **newNode** with given value.
2. Check whether list is **Empty** (**head** == **NULL**)
3. If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
4. If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
5. Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
6. Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

## Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

1. Create a **newNode** with given value.
2. Check whether list is **Empty** (**head** == **NULL**).
3. If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
4. If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
5. Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
6. Set **temp → next** = **newNode** and **newNode → next** = **head**.

## Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

1. Create a **newNode** with given value.
2. Check whether list is **Empty** (**head** == **NULL**)
3. If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
4. If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
5. Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
6. Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
7. If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
8. If **temp** is last node, then set **temp → next** = **newNode** and **newNode → next** = **head**.
9. If **temp** is not last node, then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

## Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
4. Check whether list is having only one node (**temp1 → next** == **head**)
5. If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
6. If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head** )
7. Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

## Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
4. Check whether list has only one Node (**temp1 → next** == **head**)
5. If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
6. If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
7. Set **temp2 → next**= **head** and delete **temp1**.

## Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
4. Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
5. If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
6. If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
7. If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
8. If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
9. If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
10. If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
11. If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
12. If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

## Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
3. If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
4. Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
5. Finally display **temp → data** with arrow pointing to **head → data**.

# LAB TASK:

1. **Create templated Circular Linked List ADT and provide** 
   1. Constructor / **copy constructor**
   2. Destructor
   3. InsertAtStart(intval)
   4. **InsertAtAnyposition(intposition,intval)**
   5. insertAtEnd(intval)
   6. deleteAtstart()
   7. **DeleteAtAnyposition(node)**
   8. deleteAtEnd()
   9. Traverse()
   10. isEmpty()

**Note: Your program should have no memory leakage & dangling pointers. Your program should be validated properly and display warnings and errors to user as well. Reuse functions if needed to increase program efficiency.**

**TEST PLAN:**

1. **Execute your test plan. If you discover mistakes in your implementation of the Circular List ADT, correct them and execute your test plan again.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Expected Results** | **Results/Status** |
| **1.** | Create an integer type list LinkedList<int> **myList** with default constructor | head=NULL |  |
| **2.** | Copy constructor | Head = NULL |  |
| **3.** | Check if the list isEmpty? | yes |  |
| **4.** | Insert values 1-5 in list |  |  |
| **5.** | Check if the list isEmpty? | No |  |
| **6.** | Traverse the list | 1,2,3,4,5 |  |
| **7.** | Delete from start |  |  |
| **8.** | Traverse the list | 2,3,4,5 |  |
| **9.** | Insert at Start (6) |  |  |
| **10.** | Traverse the list | 6,2,3,4,5 |  |
| **11.** | Insert at End (9) |  |  |
| **12.** | Traverse the list | 6,2,3,4,5,9 |  |
| **13.** | Delete from end |  |  |
| **14.** | Traverse the list | 6,2,3,4,5 |  |
| **15.** | Delete from position 3 |  |  |
| **16.** | Traverse the list | 6,2,4,5 |  |
| **17.** | Insert value 7 at position 4 |  |  |
| **18.** | Traverse the list | 6,2,4,7,5 |  |
| **19.** | Create an integer type list LinkedList<float> **myList2** with default constructor | head = NULL |  |
| **20.** | Insert 5 values. | 1.5, 2.5, 3.5, 4.5, 5.5 |  |
| **21.** | Delete from position 2 |  |  |
| **22.** | Traverse the list | 1.5,3.5,4.5,5.5 |  |
| **23.** | Insert at start (5.5) |  |  |
| **24.** | Insert at end (6.5) |  |  |
| **25.** | Insert at position (5,7.7) |  |  |
| **26.** | Traverse the list | 5.5,1.5,3.5,4.5,7.7,5.5 |  |

**THINK:**

|  |
| --- |
| 1. If you do not provide copy constructor in the above ADT, program will generate a run time error at the time of destructor calling. Why? 2. If an object of circular linked list is made using copy constructor and the copy done is shallow copy instead of deep copy then the newly created object will also point to same memory locations of object from which copy is being done. When destructor is called, it delete that memory locations for one object. When second object will try to delete same memory locations, there would be nothing in memory and run time error will be thrown. So reason why run time error is generated is shallow copy (which is done by compiler as a default behavior )instead of deep copy. 3. In above test plan no object is being copied, therefore copy constructor is not being called, and therefore, providing a copy constructor or not providing it, will not make any difference. |
| 1. Consider a small circular linked list. How to detect the presence of cycles in this list effectively?   **The most efficient way to determine cycle in a circular linked list is by using Floyd’s Cycle-Finding Algorithm. According to this algo:**  **Use two pointer one will be slow which will move one step and other pointer fast which will move two steps.**  **If these two pointer, slow and fast meet at a point, then there is a loop else there is no loop.** |
| 1. What will happen if you do not provide destructor to your Circular linked list ADT?   If there is no destructor then every time a new node is created, it will remain and memory and will not be deleted. Destructor deletes the memory. Therefore, without destructor, there would be memory leak in program. |
| 1. In which sequence the destructor of circular linked lists will be called? Which list will be destroyed first?   The sequence in which destructor is called for a circular linked list is last object will be deleted first and first object will be deleted at last, just like popping of stack.  The float list, myList2, will be destroyed first and then myList of integers will be destroyed. |