

**CC-213L**

**Data Structures and Algorithms**

**Laboratory 08**

**Circular LinkedList**

**Version: 1.0.0**

**Release Date: 14-10-2023**

**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
    - Member access operators
  - Circular Singly LinkedList
    - Insert Node
    - Display
  - Circular Doubly LinkedList
    - Insert Node
    - Display
- Activities
  - Pre-Lab Activity
    - Task 01: Circular Singly LinkedList Implementation

**Learning Objectives:**

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Singly Circular LinkedList
- Doubly Circular 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 Instructor	Prof. Dr. Syed Waqar ul Qounain	<a href="mailto:swjaffry@pucit.edu.pk">swjaffry@pucit.edu.pk</a>
Lab Instructor	Madiha Khalid	<a href="mailto:madiha.khalid@pucit.edu.pk">madiha.khalid@pucit.edu.pk</a>
Teacher Assistants	Muhammad Nabeel	<a href="mailto:bitf20m009@pucit.edu.pk">bitf20m009@pucit.edu.pk</a>
	Abdul Rafay Zubairi	<a href="mailto:bcsf20a032@pucit.edu.pk">bcsf20a032@pucit.edu.pk</a>

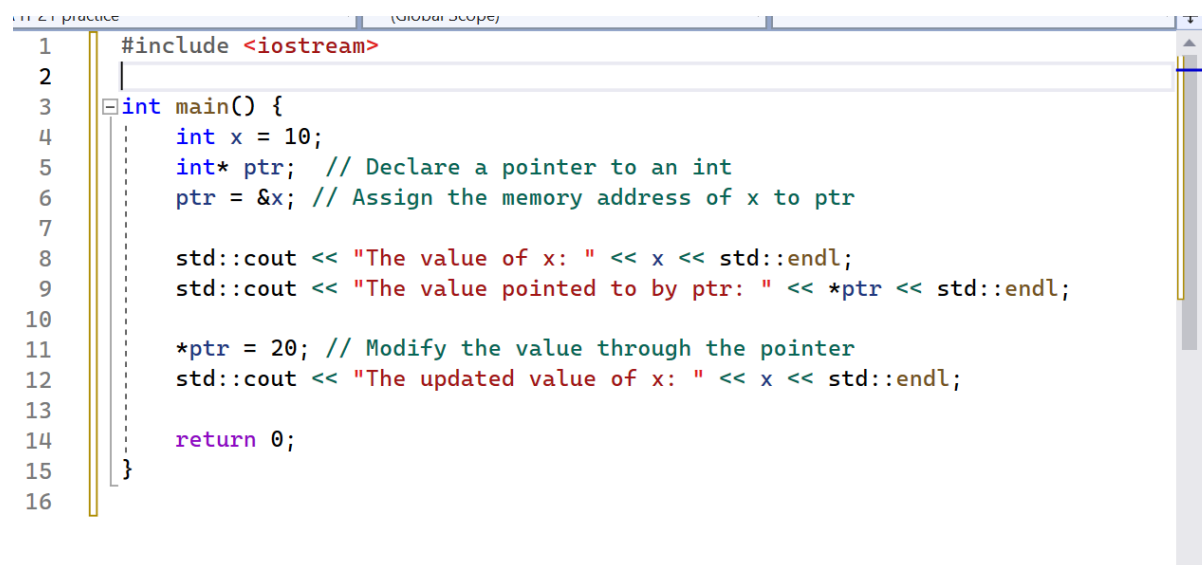
## Background and Overview

### 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.

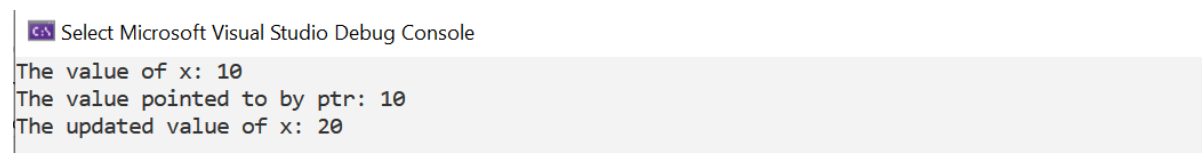


```
1  #include <iostream>
2
3  int main() {
4      int x = 10;
5      int* ptr; // Declare a pointer to an int
6      ptr = &x; // Assign the memory address of x to ptr
7
8      std::cout << "The value of x: " << x << std::endl;
9      std::cout << "The value pointed to by ptr: " << *ptr << std::endl;
10
11     *ptr = 20; // Modify the value through the pointer
12     std::cout << "The updated value of x: " << x << std::endl;
13
14     return 0;
15 }
16
```

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.

```

1  #include <iostream>
2
3  int main() {
4      int* dynamicArray = new int[5]; // Allocate an array of 5 integers
5
6      for (int i = 0; i < 5; i++) {
7          dynamicArray[i] = i * 10;
8      }
9
10     for (int i = 0; i < 5; i++) {
11         std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
12     }
13
14     delete[] dynamicArray; // Deallocate the memory
15
16     return 0;
17 }

```

Figure 3(Dynamic Memory)

**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.

Select Microsoft Visual Studio Debug Console

```

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

```

Figure 4(Output)

**Self-Referential Objects (Single Self Reference):**

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.

Example in C++

```

1  struct Node
2  {
3      int info;
4      Node* ptr;
5  };

```

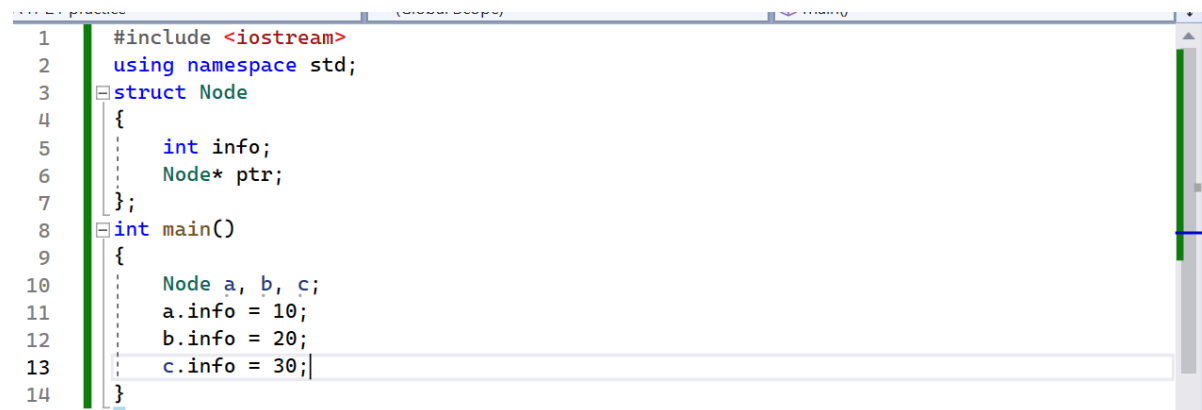
Figure 5(Self Referencing)

**Explanation:**

In Figure 5 we have declared a struct Node. It has two data members info and ptr.

**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.

**ptr** Represents 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.

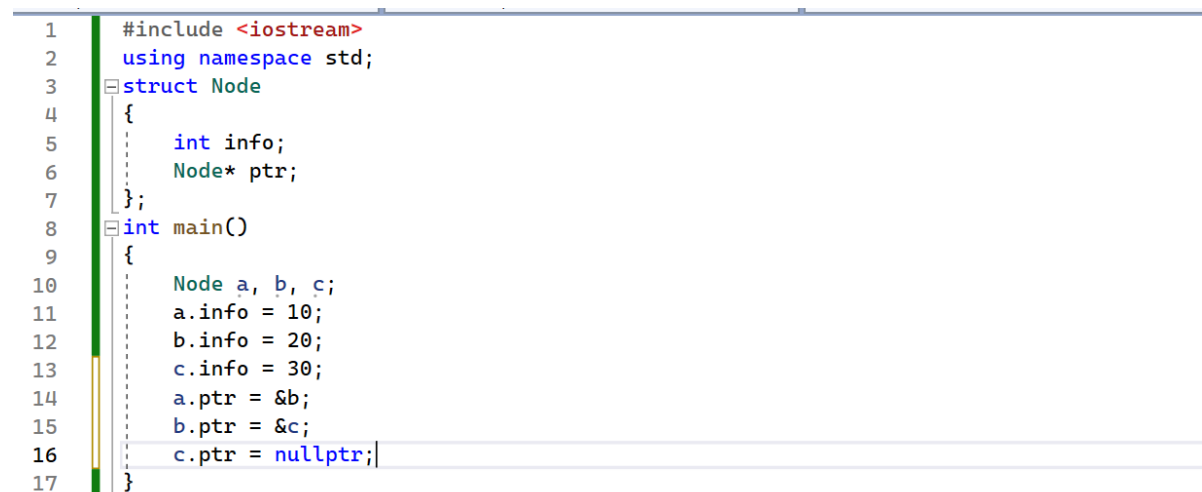


```
1  #include <iostream>
2  using namespace std;
3  struct Node
4  {
5      int info;
6      Node* ptr;
7  };
8  int main()
9  {
10     Node a, b, c;
11     a.info = 10;
12     b.info = 20;
13     c.info = 30;
14 }
```

Figure 6(Self Referential Objects)

**Explanation:**

We have declared three Node type variables a, b and stored proper values in their info data member.



```
1  #include <iostream>
2  using namespace std;
3  struct Node
4  {
5      int info;
6      Node* ptr;
7  };
8  int main()
9  {
10     Node a, b, c;
11     a.info = 10;
12     b.info = 20;
13     c.info = 30;
14     a.ptr = &b;
15     b.ptr = &c;
16     c.ptr = nullptr;
17 }
```

Figure 7(Self Referencing)

**Explanation:**

We have declared three Node type variables a, b and stored proper values in their info data member.

At line number 14 the ptr variable of type Node\* (pointer to Node) is assigned the address of Node b and similarly at line 15 ptr variable of Node b is assigned address of Node c. ptr of Node c is pointing to null.

### Some Important Operators:

- **Member Access operators** arrow operator ( $\rightarrow$ ) and dot operator ( $\cdot$ ) have same precedence with associativity from left to right.
- **Dereference /indirection operator (\*) address operator (&)** have same precedence but associativity from right to left.
- Member access operators have high priority than indirection and address operators.

```

8  int main()
9  {
10     Node a, b, c;
11     a.info = 10;
12     b.info = 20;
13     c.info = 30;
14     a.ptr = &b;
15     b.ptr = &c;
16     c.ptr = nullptr;
17     Node* p = &a;
18     cout << p->info << endl;
19     cout << p->ptr->info << endl;
20     cout << p->ptr->ptr->info << endl;
21
22 }
```

Figure 8(Member Access)

### Explanation:

On line 17, a pointer of type Node named **p** is declared. This pointer will be used to reference a node. With this **p** pointer, it becomes straightforward to traverse through the linked list, as each node contains a **ptr** member that points to the next node in the sequence. We have accessed all the next node as well as information through the  $\rightarrow$  (pointer member access operator).

```

Select Microsoft Visual Studio Debug Console

10
20
30
```

Figure 9(Output)

```

8  int main()
9  {
10     Node a, b, c;
11     a.info = 10;
12     b.info = 20;
13     c.info = 30;
14     a.ptr = &b;
15     b.ptr = &c;
16     c.ptr = nullptr;
17     Node* p = &a;
18     cout << (*p).info << endl;
19     cout << (*(p).ptr).info << endl;
20     cout << (*(p).ptr).ptr.info << endl;
21 }
22

```

Figure 10(Indirection Operator)

**Explanation:**

At line number 18,19 and 20 members have been accessed through the indirection and dot operator that is object member access operator.

```

8  int main()
9  {
10     Node a, b, c;
11     a.ptr = &b;
12     b.ptr = &c;
13     c.ptr = nullptr;
14     Node* p = &a;
15     p->info = 100;
16     p->ptr->info = 200;
17     p->ptr->ptr->info = 300;
18     while (p != nullptr)
19     {
20         cout << p->info << endl;
21         p = p->ptr;
22     }
23
24     return 0;
25 }

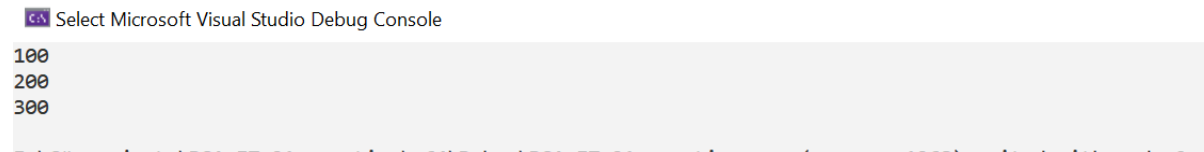
```

Figure 11(Traverse Nodes)

**Explanation:**

Rather than individually accessing information from each node, a more efficient approach is to employ a loop for traversing the nodes. This way, you can access the information in each node as long as you haven't reached a node with its **ptr** member pointing to **nullptr**.





```

100
200
300

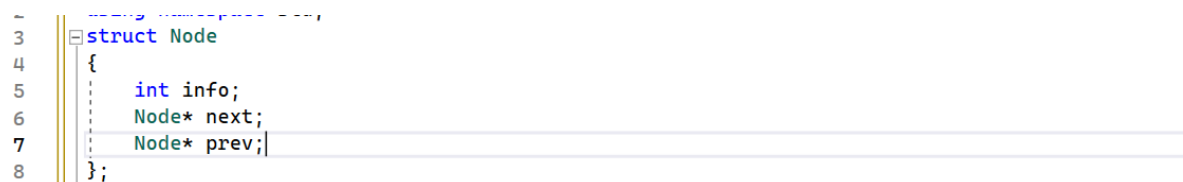
```

Figure 12(Output)

### Self-Referential Objects (Double Self Reference):

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.



```

3 struct Node
4 {
5     int info;
6     Node* next;
7     Node* prev;
8 };

```

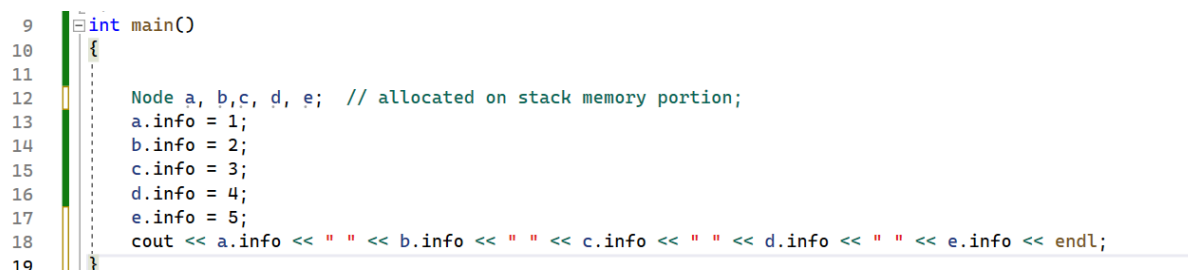
Figure 13(Self Reference)

### 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.



```

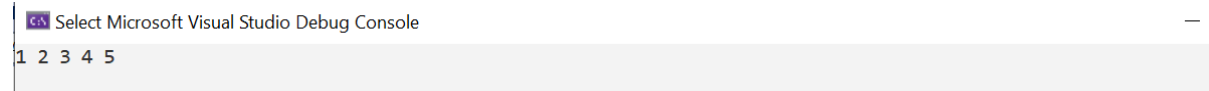
9 int main()
10 {
11
12     Node a, b, c, d, e; // allocated on stack memory portion;
13     a.info = 1;
14     b.info = 2;
15     c.info = 3;
16     d.info = 4;
17     e.info = 5;
18     cout << a.info << " " << b.info << " " << c.info << " " << d.info << " " << e.info << endl;
19 }

```

Figure 14(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 15(Output)

```

20     a.next = &b;
21     b.next = &c;
22     c.next = &d;
23     d.next = &e;
24     b.prev = &a;
25     c.prev = &b;
26     d.prev = &c;
27     e.prev = &d;
28     e.next = a.prev = nullptr;
```

Figure 16(Double Link)

### Explanation:

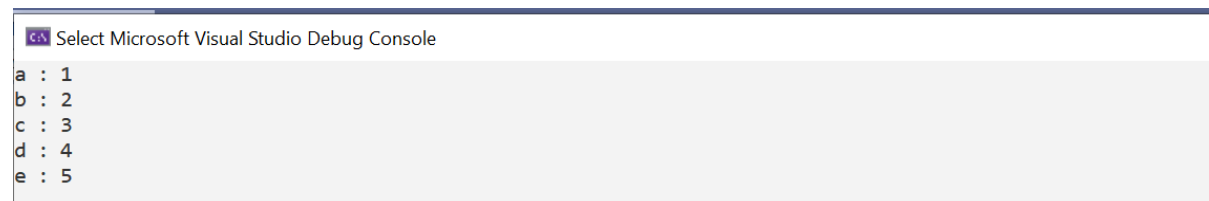
In Figure 11 we have initialized each Node next and previous pointer with the addresses of other nodes such that each node can refer a same node in sequence to a next node as well as previous Node.

```

29     Node* head = &a;
30     cout << "a : " << a.info << endl;
31     cout << "b : " << a.next->info << endl;
32     cout << "c : " << a.next->next->info << endl;
33     cout << "d : " << a.next->next->next->info << endl;
34     cout << "e : " << a.next->next->next->next->info << endl;
35 }
```

Figure 17(Double Links)

### Output:



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

Figure 18(Output)

```
30     cout << "a : " << e.prev->prev->prev->prev->info << endl;
31     cout << "b : " << e.prev->prev->prev->info << endl;
32     cout << "c : " << e.prev->prev->info << endl;
33     cout << "d : " << e.prev->info << endl;
34     cout << "e : " << e.info << endl;
35 }
```

Figure 19(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:

```
36 Node* head = &a;|
37 cout << head->next->next->prev->next->next->prev->info << endl;
38 head->next->next->prev->next->info = head->next->next->prev->next->info;
39 cout << head->next->next->prev->next->info << endl;
40 cout << (*( (*head).next)).next->prev->next).info << endl;
41
42
```

Figure 20(Doubly Link)

## Circular Singly LinkedList

A circular singly linked list is a variation of a singly linked list in which the last node of the list points back to the first node, forming a circle. In a regular singly linked list, the last node points to `null` to indicate the end of the list. However, in a circular singly linked list, the last node points to the first node, creating a circular structure.

In a circular singly linked list:

1. The last node's "next" pointer points to the first node in the list.
2. Each node in the list has a "next" pointer pointing to the next node in the sequence.
3. Traversal of the list starts from any node, and you can keep moving to the next node until you reach the starting node again.

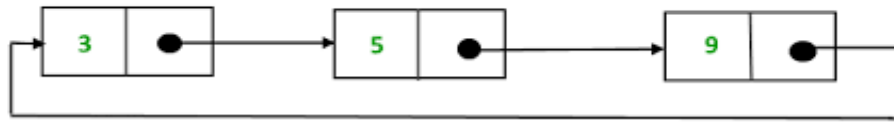


Figure 21(Circular Singly LinkedList)

```

3  class Node {
4  public:
5      int data;
6      Node* next;
7
8      Node(int value) : data(value), next(nullptr) {}
9  };
  
```

Figure 22(Node class)

```

58  int main()
59  {
60      Node* head = new Node(10);
61      head->next = new Node(20);
62      head->next->next = new Node(30);
63      head->next->next->next = head;
64
65      Node* temp = head;
66      while (temp->next != head)
67      {
68          cout << temp->data << endl;
69          temp = temp->next;
70      }
71      cout << temp->data << endl;
72
73      return 0;
74  }
  
```

Figure 23(Circular Singly LinkedList)

## Output

```

Select Microsoft Visual Studio Debug Console
10
20
30
E:\C++ projects\Circular Singly LinkedList\vs64\Debug\Circular Singly LinkedList.exe (process 4788) exited
  
```

Figure 24(Output)

## Circular Doubly LinkedList

A circular doubly linked list is a type of linked list in which each node in the list contains data, a pointer to the next node, and a pointer to the previous node. The circular doubly linked list is similar to a regular doubly linked list, but in this case, the last node points back to the first node, forming a circular structure.

In a circular doubly linked list:

Each node has three components: data, a "next" pointer pointing to the next node, and a "previous" pointer pointing to the previous node.

The "next" pointer of the last node in the list points to the first node, and the "previous" pointer of the first node points to the last node, creating a circular connection.

```

59  class Node
60  {
61  public:
62      int data;
63      Node* next;
64      Node* prev;
65
66      Node(int value) : data(value), next(nullptr), prev(nullptr) {}
67  };

```

Figure 25(Doubly LinkedList Node)

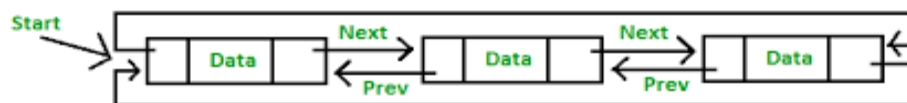


Figure 26(Circular Doubly LinkedList)

```

69  int main()
70  {
71      Node* head = new Node(10);
72      head->next = new Node(20);
73      head->next->prev = head;
74      head->next->next = head;
75
76      cout << head->next->prev->data << endl; // 10
77
78      cout << head->next->next->next->data << endl;
79      return 0;
80  }

```

Figure 27(Doubly LinkedList)



## Activities

### Pre-Lab Activities:

#### Task 01: Circular Singly LinkedList implementation

In previous Lab you have implemented Singly Linear LinkedList. Modify implementation of that LinkedList and make it **circular Singly LinkedList**.

```
template<class T>
class LinkedList;
template<class T>
class Node
{
public:
    T info;
    Node<T>* next;
    // Methods...
};
template<class T>
class LinkedList
{
    Node<T>* head;

    // Methods...
};
```

Implement following functions for List class.

**1. Constructor, destructor, Copy-constructor.**

**2. void insertAtHead( T value )**

**3. void insertAtTail( T value )**

**4. bool deleteAtHead()**

**5. bool deleteAtTail()**

**6. void printList()**

**7. Node\* getNode(int n)**

This function should return pointer to nth node in the list. Returns last node if n is greater than the number of nodes present in the list.

**8. bool insertAfter( T value, T key )**

Insert a node after some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise.

**9. bool insertBefore( T value, T key )**

Insert a node before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise.

**10. bool deleteBefore( T key )**

Delete a node that is before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or first node in the list.

**11. bool deleteAfter( T value )**

Delete a node that is after some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or first node in the list.

**12. int getLength( )** returns the total number of nodes in the list.

**13. Node\* search(T x)**

Search a node with value “x” from list and return its link. If multiple nodes of same value exist, then return pointer to first node having the value “x”.