Batch: B2	**Roll No.:16010124107**
Experiment No.	
Grade: AA / AB / BB / BC / CC / CD /DD	

Title: Implementation of various types of LL- doubly LL, circular LL

Objective: To understand the use of linked lists as data structures for various applications.

Expected Outcome of Experiment:

CO	Outcome
CO 2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:
https://www.geeksforgeeks.org/dsa/linked-list-data-structure/

Introduction:

A linked list is a linear data structure that stores a sequence of elements, called nodes, in a non-contiguous manner. Each node contains both data and a pointer (or reference) to the next node in the sequence, allowing them to form a dynamic chain. The list is accessed through a "head" pointer, which points to the first node, and the last node's pointer typically points to NULL, indicating the end of the list.

Types of linked list:

There are two types of linked lists:

1. Circular linked list:

   In this type of linked list, the last element is linked to the first element. Thus, you're still allowed to access the previous element while traversing the list by going all around in a revolution.

We have two types: Ordered and unordered circular linked lists. Ordered circular linked lists store elements in a sorted order and unordered circular linked lists store elements in the order of their insertion.

2. Doubly Linked Lists

   In this type, each node in a linked list is linked to its next and previous element. So each node has one value and two pointers. This is more memory heavy but it allows faster access to the previous and next element and allows for more flexible traversal and other operations.

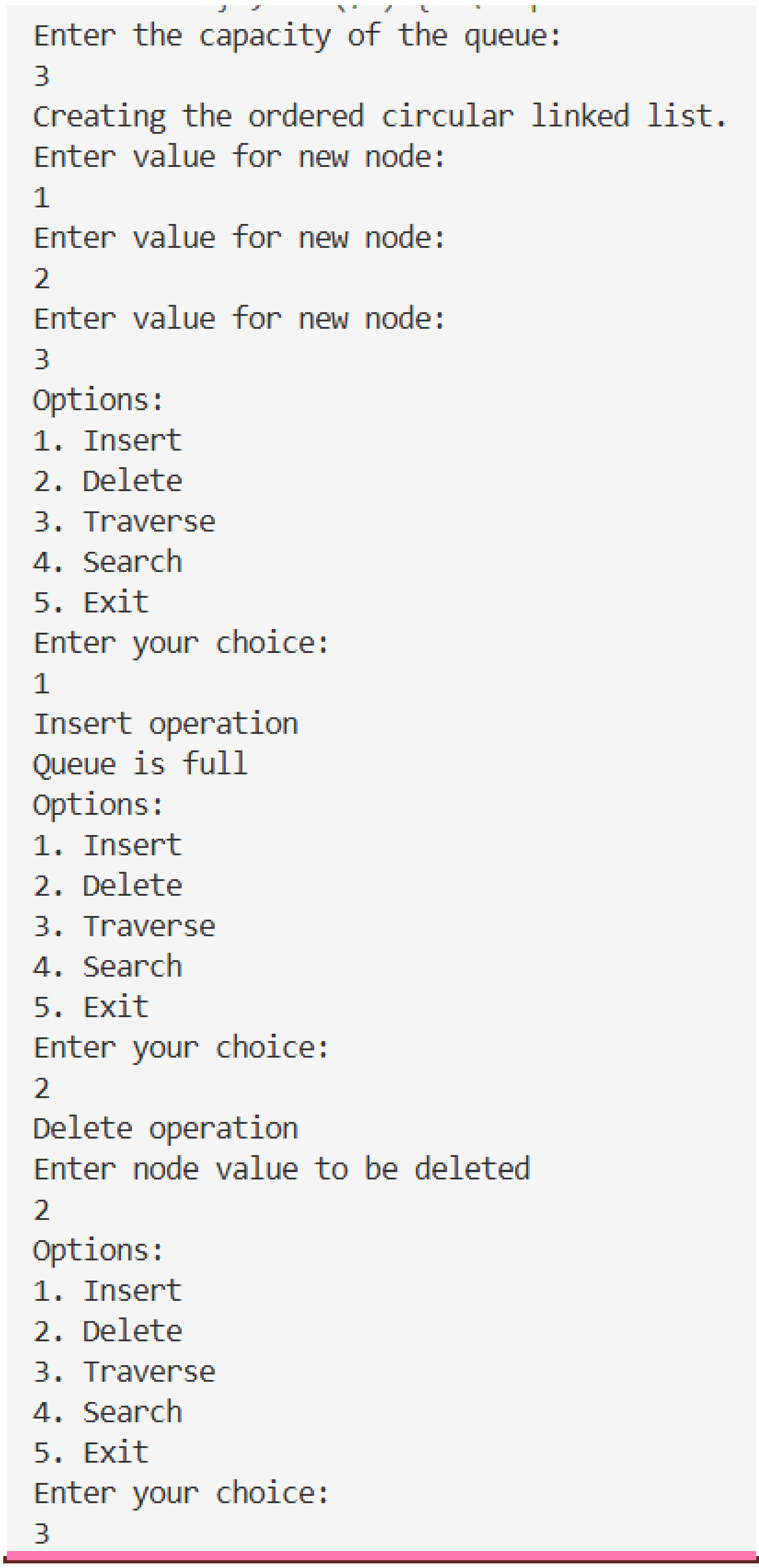
Algorithm for creation, insertion, deletion, traversal and searching an element in assigned linked list type:

Program

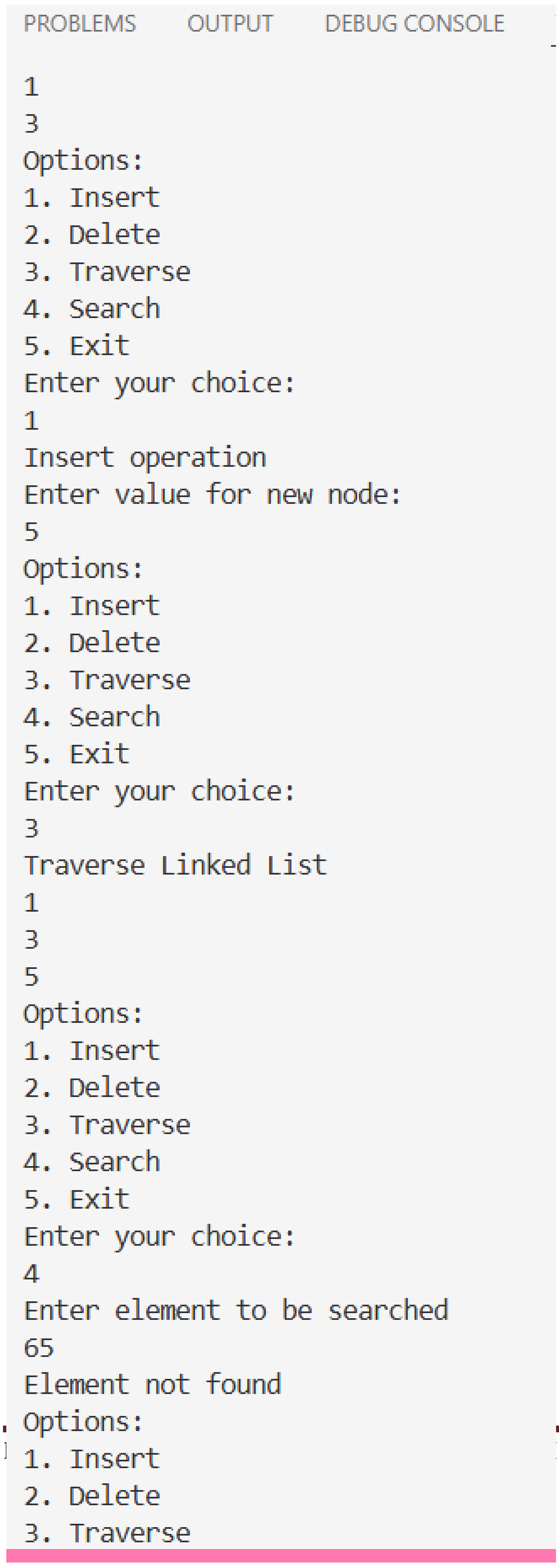
```
#include <bits/stdc++.h>

using namespace std;


struct Node {

    int value;

    Node* next;

};


int main()

{

    Node* last = NULL;

    int capacity;

    int currentSize = 0;

    bool found = false;

    Node *q;


    cout << "Enter the capacity of the queue:\n";

    cin >> capacity;


    cout << "Creating the ordered circular linked list.\n";

    while (currentSize < capacity) {

        Node* newNode = new Node();

        cout << "Enter value for new node:\n";
        cin >> newNode -> value;

        newNode -> next = NULL;

        //created a new node


        if (last == NULL) {

            last = newNode;

            last -> next = last;

        }

        else {

            if (newNode -> value < last -> next -> value) { // insert at beginning

                newNode -> next = last -> next;

                last -> next = newNode;

            }

            else if (newNode -> value >= last -> value) { // insert at end

                newNode -> next = last -> next;

                last -> next = newNode;

                last = newNode;

            }

            else { // insert in middle

                Node* current = last -> next;

                  while (current -> next != last -> next && current -> next -> value < newNode ->
value) {

                    current = current -> next;

                }//traverse to find predecessor and successor
                newNode -> next = current -> next;

                current -> next = newNode;

            }

        }

        currentSize++;

    }


    int indx; //for searching2

    while (true) {

        cout << "Options:\n1. Insert\n2. Delete\n3. Traverse\n4. Search\n5. Exit\n";

        cout << "Enter your choice:\n";

        int choice;

        cin >> choice;

        switch(choice)

        {

            case 1:

                cout << "Insert operation\n";

                if (currentSize == capacity) {

                    cout << "Queue is full\n";

                }

                else {

                    Node* newNode = new Node();

                    cout << "Enter value for new node:\n";

                    cin >> newNode -> value;
                    newNode -> next = NULL;


                    if (newNode -> value < last -> next -> value) { // insert at beginning
                        newNode -> next = last -> next;
                        last -> next = newNode;
                    }
                    else if (newNode -> value >= last -> value) { // insert at end
                        newNode -> next = last -> next;
                        last -> next = newNode;
                        last = newNode;
                    }
                    else { // insert in middle
                        Node* current = last -> next;
                            while (current -> next != last -> next && current -> next -> value <
newNode -> value) {
                            current = current -> next;
                        }
                        newNode -> next = current -> next;
                        current -> next = newNode;
                    }
                    currentSize++;
                }
                break;
            case 2:
```

```
        cout << "Delete operation\n";

        found = false;

        if(last == NULL) {

            cout << "The linked list is empty\n";

        }

        else {

            cout << "Enter node value to be deleted\n";

            int val;

            cin >> val;

            q = last -> next;

            Node* prev = last;

            do {

                if(q -> value == val) {

                    if (q == last && q == last -> next) { // only one node

                        delete q;

                        last = NULL;

                    } else {

                        prev -> next = q -> next;

                        if (q == last) last = prev;

                        delete q;

                    }

                    found = true;

                    currentSize--;

                    break;
                }
                prev = q;
                q = q -> next;
            } while(q != last -> next);

            if(!found) {
                cout << "Element not found\n";
            }
        }
        break;
    case 3:
        cout << "Traverse Linked List\n";
        if(currentSize==0)
        {
            cout << "Linked List is empty\n";
            break;
        }
        q = last -> next;
        while(q != last)
        {
            cout << q -> value << endl;
            q = q -> next;
        }
        cout << q -> value << endl; // print last node
        break;

    case 4:

        found = false;

        cout << "Enter element to be searched\n";

        int el;

        cin >> el;

        q = last -> next;

        indx=0;

        while(q -> value <= el && q != last -> next || indx==0)

        {

            if(q -> value == el && !found)

            {

                cout << "Element found at index " << indx << endl;

                found = true;

            }

            q = q -> next;

            indx++;

        }

        if(!found)

        {

            cout << "Element not found\n";

        }

        break;

    case 5:
        cout << "Exiting..." << endl;

        return 0;

        default:

        cout << "Invalid Choice" << endl;

        break;

    }

}

}
```

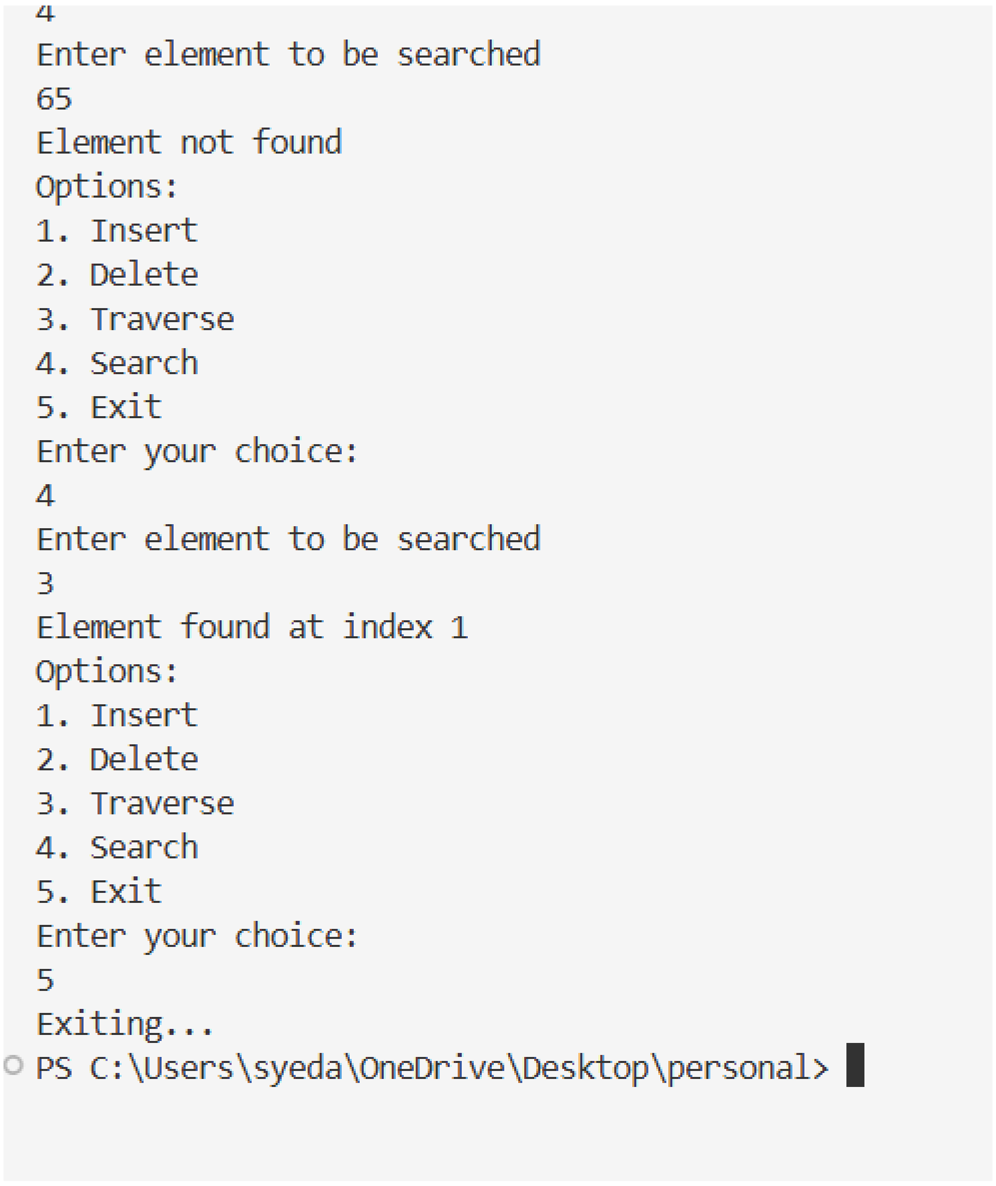
Output:-



```
Enter the capacity of the queue:
3
Creating the ordered circular linked list.
Enter value for new node:
1
Enter value for new node:
2
Enter value for new node:
3
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
1
Insert operation
Queue is full
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
2
Delete operation
Enter node value to be deleted
2
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
3
```


PROBLEMS OUTPUT DEBUG CONSOLE

```
1
3
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
1
Insert operation
Enter value for new node:
5
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
3
Traverse Linked List
1
3
5
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
4
Enter element to be searched
65
Element not found
Options:
1. Insert
2. Delete
3. Traverse
```



```
4
Enter element to be searched
65
Element not found
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
4
Enter element to be searched
3
Element found at index 1
Options:
1. Insert
2. Delete
3. Traverse
4. Search
5. Exit
Enter your choice:
5
Exiting...
PS C:\Users\syeda\OneDrive\Desktop\personal>
```

Conclusion:-

In conclusion, a circular linked list offers significant advantages, especially when it comes to maintaining a continuous, cyclic structure for elements. By keeping the list ordered, we can efficiently perform insertions and deletions without disrupting the integrity of the list. This type of data structure allows for quick access to elements, as the traversal process seamlessly loops from the last back to the first element.

Post lab questions:

1. Compare and contrast SLL and DLL.

Singly Linked List	Doubly Linked List
Has pointers only to the next element	Has pointer to both next and previous elements
Takes up less memory space	Takes up more memory space
More time costly to access previously visited elements	Less time costly to access previously visited elements
Unidirectional	Bidirectional
Simpler implementation since only one pointer management is required	Harder to implement since we need to manage two pointers side by side.

2. List any 3 scenarios where circular linked lists are preferable over linear linked lists?

Circular linked lists are preferable over linear linked lists as the last element links to the first element and a more efficient usage of memory is achieved.

1. When implementing a queue using linked lists, a circular queue is better for efficient memory management as when one element leaves the queue, it creates more room for others.
2. Music playlist: For playlists that are designed to loop continuously, a circular linked list allows seamless transition from the last song back to the first without explicit checks for the end of the list.
3. In turn-based games, a circular linked list can manage player turns, ensuring each player gets their turn in a sequence and the cycle repeats once all players have had their turn, like in a bowling alley.

3. How would you implement a function to reverse a doubly linked list?

Algorithm:

```
● Start
● Set current = head.
● Set temp = NULL.
● While current != NULL:
      a. Swap current -> prev and current -> next.
      b. Move current = current -> prev.
● After loop, if temp != NULL, set head = temp -> prev.
● Return head.
● End
```

Pseudocode:

```
Node* reverseDoublyList(Node* head) {
    Node* current = head;
    Node* temp = NULL;
    while (current != NULL) {
        // swap prev and next
        temp = current -> prev;
        current -> prev = current -> next;
        current -> next = temp;
        // move to next node (which is prev now)
        current = current -> prev;
    }
    // adjust head
    if (temp != NULL) {
        head = temp -> prev;
    }
    return head;
}
```

4. What are some practical applications of circular doubly linked lists in real-world systems? How does their structure provide advantages in these scenarios?

Practical Applications:

1. Music playlist management: In a circular doubly linked list, you can go back and forth between elements and the last element connects to the first. This is cleanly implemented in a music playlist where you can go back to your previously listened song and after your playlist is over, you go back to your first song again.
2. Advertisements / Slideshows: Slideshows on website carousels allow you to go back and forth between slides, and after the last slide, the carousel rolls back to the first slide.
3. Round-robin CPU scheduling: processes arranged circularly, pointer cycles endlessly.

4. Mobile games: Games where you can play the previous or the next level. After the last game level is played, the game recoils to the first level as you restart the game.

Circular doubly linked lists in real world systems operate by allowing you to access the previous and next pointers at any time. There is no null pointer, and the last node is linked to the first node forming a loop. This structure is very helpful in implementing real world ideas where you'd want to move bidirectionally and don't want to reach the end.