Batch:D3 Roll No.: 16010123294

Experiment No. 06

Grade: AA / AB / BB / BC / CC / CD /DD

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

Introduction:

Define Linked List

Types of linked list:

Circular linkedlist

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

Initialize Circular Linked List:

Set the head pointer to NULL to represent an empty list.

Insert Operation:

Input: Data to be inserted.

Create a new node with the given data.

If the list is empty:

Set the new node as the head and point its next to itself (to make it circular).

If the list is not empty:

Traverse to the last node (the one that points to the head).

Set the new node's next to point to the head.

Update the last node's next to point to the new node.

Delete Operation:

Input: Key (data) to be deleted.

If the list is empty, print that the list is empty.

If the head node contains the key:

If there is only one node, free it and set head to NULL.

Otherwise, traverse to the last node and update head to head->next, then link the last node to the new head. Free the old head.

If the key is in a non-head node:

Traverse the list to find the node containing the key.

Unlink the node and free it.

If the key is not found, print that it doesn't exist.

Search Operation:

Input: Key (data) to search.

If the list is empty, print that the list is empty.

Traverse the list starting from the head.

If the key is found, print its position.

If the key is not found after traversing, print that the element is not in the list.

Display Operation:

If the list is empty, print that the list is empty.

Otherwise, start from the head and print each node's data until you return to the head.

Repeat Menu:

Keep presenting the user with the menu until they choose to exit.

Program

```
#include <stdio.h>
#include <stdlib.h>

// Node structure for Circular Linked List
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the list
void insert(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    struct Node* temp = *head;

    if (*head == NULL) {
```

```
        *head = newNode;
        newNode->next = *head;  // Point to itself to make it circular
        return;
    }

    while (temp->next != *head) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->next = *head;
}

// Function to delete a node by its value
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node *temp = *head, *prev;

    // If the head node itself holds the key to be deleted
    if (temp->data == key) {
```

```
    if (temp->next == *head) {
        *head = NULL;
        free(temp);
        return;
    }

    while (temp->next != *head) {
        temp = temp->next;
    }

    temp->next = (*head)->next;
    free(*head);
    *head = temp->next;
    return;
}

// Search for the key to be deleted
prev = *head;
temp = (*head)->next;
while (temp != *head && temp->data != key) {
    prev = temp;
    temp = temp->next;
}
```

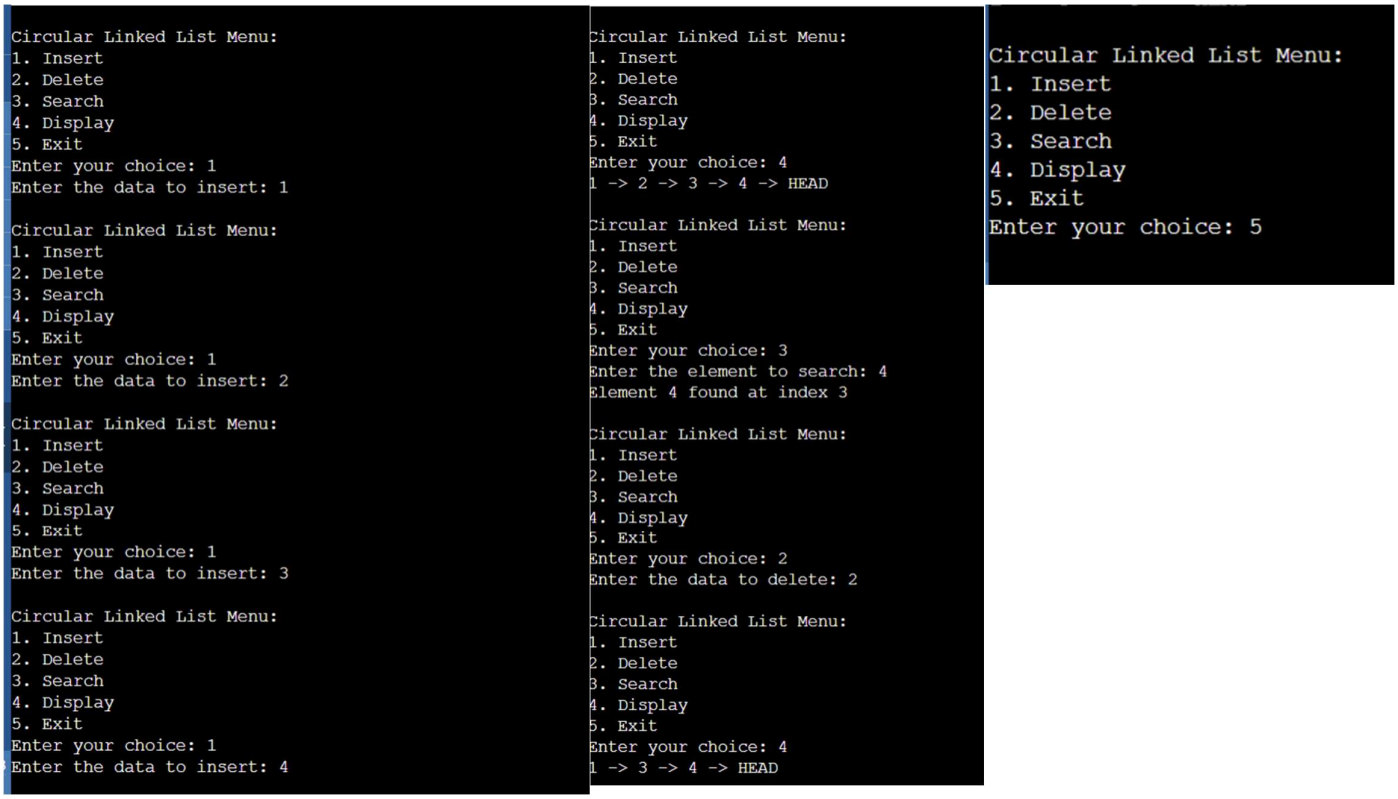
```
    // If key was not present in the list
    if (temp == *head) {
        printf("Key not found.\n");
        return;
    }

    prev->next = temp->next;
    free(temp);
}

// Function to search for a value in the circular linked list
void search(struct Node* head, int key) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    int index = 0;
    do {
        if (temp->data == key) {
            printf("Element %d found at index %d\n", key, index);
            return;
        }
```

```
        temp = temp->next;
        index++;
    } while (temp != head);

    printf("Element %d not found in the list.\n", key);
}

// Function to display the circular linked list
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("HEAD\n");
}

// Main function with menu
int main() {
```

```
struct Node* head = NULL;
int choice, data;

while (1) {
    printf("\nCircular Linked List Menu:\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Search\n");
    printf("4. Display\n");
    printf("5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter the data to insert: ");
            scanf("%d", &data);
            insert(&head, data);
            break;

        case 2:
            printf("Enter the data to delete: ");
            scanf("%d", &data);
            deleteNode(&head, data);
```

```
            break;

        case 3:
            printf("Enter the element to search: ");
            scanf("%d", &data);
            search(head, data);
            break;

        case 4:
            display(head);
            break;

        case 5:
            exit(0);

        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Output:-



Conclusion:-

Learned implementing circular linkedlist and provided the codes as well

Post lab questions:

1. Compare and contrast SLL and DLL.

Node Structure:
SLL: Each node has one pointer to the next node.
DLL: Each node has two pointers—one to the next node and one to the previous node.
Traversal:
SLL: Can be traversed in only one direction (forward).
DLL: Can be traversed in both directions (forward and backward).
Memory Usage:
SLL: Requires less memory since it has a single pointer per node.
DLL: Requires more memory due to the extra pointer to the previous node.
Insertion/Deletion Complexity:
SLL: Insertion and deletion are simpler but require extra steps to update links (especially for deleting nodes other than the head).
DLL: Easier insertion and deletion, especially in the middle, since both next and previous pointers are available.
Reversing the List:
SLL: More complex to reverse as it requires changing the next pointers for each node.
DLL: Easier to reverse as each node has a pointer to the previous node.
Use Cases:
SLL: Suitable when memory is constrained and only forward traversal is needed.
DLL: Preferred when bidirectional traversal is required or frequent node deletions/insertion in the middle of the list are needed.

2. List any 3 scenarios where circular linked lists are preferable over linear linked lists?
   - **Round-Robin Scheduling**: In operating systems, circular linked lists are used for processes that need to be cycled through repeatedly (e.g., CPU scheduling).
   - **Continuous Looping**: Circular linked lists are ideal for applications that require continuous traversal of elements without restarting from the beginning (e.g., multiplayer games).
   - **Buffer Management**: In cases like a circular buffer (ring buffer), circular linked lists are efficient for managing buffers with fixed sizes, where overwriting of old data is needed.

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

Initialize a temporary pointer temp and a current pointer curr pointing to the head of the list.
Traverse the list:
For each node, swap its next and prev pointers.
Move curr to the original prev (which is now the next node after the swap).
Once the traversal is complete, set the **new head** to the last node (which is temp->prev after the loop).

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

Music/Video Playlists:
Used for implementing "next" and "previous" navigation in a loop without resetting to the start.
Advantage: Allows both forward and backward traversal with continuous looping.

Browser Tabs:
Circular doubly linked lists can manage open tabs where users can switch between them in either direction.
Advantage: Efficient navigation between tabs in both directions without needing to start over.

Real-Time Simulations:
Common in scenarios like simulations or multiplayer games where entities or players are processed in a loop.
Advantage: The ability to traverse forward and backward seamlessly without an end.