Data Structures : Algorithms and Applications(Lab)

EXPERIMENT No: 7

AIM: To implement various operations such as insertion, deletion, search, and display in a Circular Singly Linked List.

THEORY:
A Circular Singly Linked List is a variation of a linked list in which the last node points back to the first node, forming a circular structure. This allows efficient traversal where the end of the list links back to the start, providing continuous cyclic data processing.

In this implementation, each node consists of two fields:

Data: Stores the value of the node.
Next: A pointer that points to the next node in the list.

Key Operations:
1.Insertion:

- At the beginning: A new node is inserted as the head of the list, and the last node's pointer is updated to point to the new head.
- At the end: A new node is inserted at the end of the list, and its next pointer is made to point to the head to maintain the circular nature.

2.Deletion:

- From the beginning: The first node is removed, and the head pointer is updated. The last node's next pointer is updated to point to the new head.
- From the end: The last node is deleted, and the second last node's next pointer is updated to point to the head.

3.Search:

- The list is traversed from the head, comparing each node's data with the search key. If the item is found, its position is returned.

4.Display:

- The list is traversed starting from the head, printing each node’s data until the list loops back to the head.

Characteristics of a Circular Singly Linked List:

- Circular Structure: The last node points to the first node, allowing continuous traversal.
- Dynamic Size: Nodes can be added or removed at runtime.
- No Null Pointers: There is no NULL in the next pointer of any node unless the list is empty.
- This circular nature of the linked list provides benefits for applications that require constant cycling through data, such as real-time simulations, circular buffers, or task scheduling systems.

CODE:

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

struct Node {
    int data;
    struct Node *next;
};

// Function prototypes
void insertStart(struct Node **head, int data);
void insertLast(struct Node **head, int data);
void insertPosition(int data, int pos, struct Node **head);
void deleteStart(struct Node **head);
void deleteLast(struct Node **head);
void deletePosition(int pos, struct Node **head);
void display(struct Node *head);

int main() {
    struct Node *head = NULL;

    // Inserting nodes
    printf("Insert at beginning: ");
    insertStart(&head, 2);
    insertStart(&head, 1);
    display(head);

    printf("Insert at End: ");
    insertLast(&head, 30);
    insertLast(&head, 40);
    display(head);

    printf("Insert at Specific Position: ");
    insertPosition(5, 3, &head);
    display(head);

    // Deleting nodes
    printf("Delete from beginning: ");
    deleteStart(&head);
    display(head);

    printf("Delete from end: ");
    deleteLast(&head);
    display(head);

    printf("Delete from specific position: ");
    deletePosition(2, &head);
    display(head);
```

```
    return 0;
}

void insertStart(struct Node **head, int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;

    // If it's the first node being entered
    if (*head == NULL) {
        *head = newNode;
        (*head)->next = *head;  // Point to itself
        return;
    }

    // If LL already has >=1 node
    struct Node *curr = *head;

    // Traverse till the last node in LL
    while (curr->next != *head) {
        curr = curr->next;
    }
    // Assign LL's last node's next as this new node
    curr->next = newNode;

    // Assign newNode's next as current head
    newNode->next = *head;

    // Change head to this new node
    *head = newNode;
}

void insertLast(struct Node **head, int data) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;

    // If it's the first node being entered
    if (*head == NULL) {
        *head = newNode;
        (*head)->next = *head;  // Point to itself
        return;
    }

    // If LL already has >=1 node
    struct Node *curr = *head;

    // Traverse till the last node in LL
    while (curr->next != *head) {
```

```
        curr = curr->next;
    }

    // Assign LL's current last node's next as this new node
    curr->next = newNode;

    // Assign this new node's next as current head of LL
    newNode->next = *head;
}

void insertPosition(int data, int pos, struct Node **head) {
    struct Node *newnode, *curNode;

    if (*head == NULL) {
        printf("List is empty\n");
        return;
    }

    if (pos == 1) {
        insertStart(head, data);
        return;
    } else {
        newnode = (struct Node *)malloc(sizeof(struct Node));
        newnode->data = data;
        curNode = *head;

        // Traverse to the position before the desired position
        for (int i = 1; i < pos - 1; i++) {
            curNode = curNode->next;
            if (curNode == *head) {
                printf("Position out of bounds. Inserting at end.\n");
                insertLast(head, data);
                return;
            }
        }

        newnode->next = curNode->next;
        curNode->next = newnode;
    }
}

void deleteStart(struct Node **head) {
    if (*head == NULL) {
        printf("List is empty, cannot delete.\n");
        return;
    }

    struct Node *temp = *head;
```

```
    if ((*head)->next == *head) { // Only one node
        free(*head);
        *head = NULL;
    } else {
        struct Node *curr = *head;
        while (curr->next != *head) { // Find the last node
            curr = curr->next;
        }
        curr->next = (*head)->next; // Bypass the head node
        free(temp); // Free the old head
        *head = curr->next; // Update head to the next node
    }
}

void deleteLast(struct Node **head) {
    if (*head == NULL) {
        printf("List is empty, cannot delete.\n");
        return;
    }

    struct Node *temp = *head;
    if ((*head)->next == *head) { // Only one node
        free(*head);
        *head = NULL;
    } else {
        struct Node *prev = NULL;
        while (temp->next != *head) { // Find the last node
            prev = temp;
            temp = temp->next;
        }
        prev->next = *head; // Bypass the last node
        free(temp); // Free the old last node
    }
}

void deletePosition(int pos, struct Node **head) {
    if (*head == NULL) {
        printf("List is empty, cannot delete.\n");
        return;
    }

    if (pos == 1) {
        deleteStart(head);
        return;
    }

    struct Node *curr = *head;
    struct Node *prev = NULL;
```

```
    // Traverse to the position before the desired position
    for (int i = 1; i < pos; i++) {
        prev = curr;
        curr = curr->next;
        if (curr == *head) {
            printf("Position out of bounds.\n");
            return;
        }
    }

    prev->next = curr->next; // Bypass the node to delete
    free(curr); // Free the node
}

void display(struct Node *head) {
    // if there are no nodes in LL
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node *temp = head;

    // Need to take care of the circular structure of LL
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("\n");
}
```

OUTPUT:

```
/tmp/St1taDCumv.o
Insert at beginning: 1 2
Insert at End: 1 2 30 40
Insert at Specific Position: 1 2 5 30 40
Delete from beginning: 2 5 30 40
Delete from end: 2 5 30
Delete from specific position: 2 30


=== Code Execution Successful ===
```

CONCLUSION:

The Circular Singly Linked List provides efficient insertion, deletion, search, and display operations with continuous traversal, making it ideal for cyclic data processing in applications like task scheduling and real-time simulations. Its dynamic size and circular structure enhance flexibility and usability.