**Experiment no 7:Implementation of Circular Linked List ADT**

**Yash Khot**

**SE-3 05**

**Aim:** Implementation of Circular Linked List ADT

**Objective:** Circular Linked List can be used to manage the Computing Resources of the computer. Data Structure such as stacks and queue are implemented with the help of circular linked list

**Theory :**

In a circular linked list, the last node contains a pointer to the first node of the list. We can have

a circular singly linked list as well as a circular doubly linked list. While traversing a circular

linked list, we can begin at any node and traverse the list in any direction, forward or backward,

until we reach the same node where we started. Thus, a circular linked list has no beginning and

no ending.

The only downside of a circular linked list is the complexity of iteration. Note that there are

no NULL values in the NEXT part of any of the nodes of list. Circular linked lists are widely used in operating systems for task

maintenance.

**Algorithm:** Circular Linked List Implementation

1. Create a structure for the linked list node:

- Define a structure with two members: data and a pointer to the next node (usually called 'next').

2. Initialize the head of the circular linked list:

- Set a pointer variable (usually called 'head') to NULL to indicate an empty list.

3. Define functions for common operations:

a. Function to create a new node:

- Input: Data value to be stored in the node.

- Create a new node by dynamically allocating memory.

- Set the data field of the new node to the input value.

- Set the 'next' pointer of the new node to NULL.

- Return the new node.

b. Function to insert a node at the beginning of the circular list:

- Input: Data value to insert.

- Create a new node using the 'createNode' function.

- Check if the list is empty (head is NULL). If so, set 'head' to the new node.

- Otherwise, set the 'next' pointer of the new node to the current 'head' node.

- Update 'head' to point to the new node.

- Update the 'next' pointer of the last node in the list to point to the new node to maintain the circular connection.

c. Function to delete a node with a given data value:

- Input: Data value to delete.

- Traverse the list using two pointers, one pointing to the current node and the other to the previous node.

- When you find a node with the desired data value, update the 'next' pointer of the previous node to skip the current node.

- If the node to be deleted is the 'head,' update 'head' to the next node.

- Free the memory occupied by the deleted node.

d. Function to traverse and display the circular linked list:

- Start at the 'head' node and continue until you return to the 'head' node (indicating a complete traversal).

- Use a loop to print the data in each node during the traversal.

- To avoid infinite looping, use a 'do-while' loop to handle the case where the list is initially empty.

4. Initialize the circular linked list in the 'main' function:

- Set 'head' to NULL to create an empty list.

5. Perform operations by calling the defined functions in the 'main' function:

- Insert nodes at the beginning.

- Delete nodes with specific data values.

- Display the circular linked list to verify the operations.

6. Properly manage memory by freeing allocated nodes when they are no longer needed.

7. Implement error handling for scenarios like memory allocation failures or accessing an empty list for robustness.

8. Ensure the 'next' pointer of the last node always points back to the 'head' node, creating a circular structure.

9. Handle edge cases, such as inserting or deleting nodes in an empty list.

10. Implement the Circular Linked List ADT in the chosen programming language, translating these steps into the appropriate syntax.

**Code :**

#include <stdio.h>

#include <stdlib.h>

// Structure for a node

struct Node {

int data;

struct Node\* next;

};

// Function to insert a node at the

// beginning of a Circular linked list

void push(struct Node\*\* head\_ref, int data)

{

// Create a new node and make head

// as next of it.

struct Node\* ptr1 = (struct Node\*)malloc(sizeof(struct Node));

ptr1->data = data;

ptr1->next = \*head\_ref;

// If linked list is not NULL then

// set the next of last node

if (\*head\_ref != NULL) {

// Find the node before head and

// update next of it.

struct Node\* temp = \*head\_ref;

while (temp->next != \*head\_ref)

temp = temp->next;

temp->next = ptr1;

}

else

// For the first node

ptr1->next = ptr1;

\*head\_ref = ptr1;

}

// Function to print nodes in a given

// circular linked list

void printList(struct Node\* head)

{

struct Node\* temp = head;

if (head != NULL) {

do {

printf("%d ", temp->data);

temp = temp->next;

} while (temp != head);

}

printf("\n");

}

// Function to delete a given node

// from the list

void deleteNode(struct Node\*\* head, int key)

{

// If linked list is empty

if (\*head == NULL)

return;

// If the list contains only a

// single node

if ((\*head)->data == key && (\*head)->next == \*head) {

free(\*head);

\*head = NULL;

return;

}

struct Node \*last = \*head, \*d;

// If head is to be deleted

if ((\*head)->data == key) {

// Find the last node of the list

while (last->next != \*head)

last = last->next;

// Point last node to the next of

// head i.e. the second node

// of the list

last->next = (\*head)->next;

free(\*head);

\*head = last->next;

return;

}

// Either the node to be deleted is

// not found or the end of list

// is not reached

while (last->next != \*head && last->next->data != key) {

last = last->next;

}

// If node to be deleted was found

if (last->next->data == key) {

d = last->next;

last->next = d->next;

free(d);

}

else

printf("Given node is not found in the list!!!\n");

}

// Driver code

int main()

{

// Initialize lists as empty

struct Node\* head = NULL;

// Created linked list will be

// 2->5->7->8->10

push(&head, 2);

push(&head, 5);

push(&head, 7);

push(&head, 8);

push(&head, 10);

printf("List Before Deletion: ");

printList(head);

deleteNode(&head, 7);

printf("List After Deletion: ");

printList(head);

return 0;

}

**Output :**



**Conclusion :**In conclusion, implementing a Circular Linked List Abstract Data Type (ADT) involves creating a dynamic data structure where nodes form a circular connection, with each node pointing to the next node. This ADT is valuable for applications where a circular data structure is needed. Key aspects include initializing the structure, performing operations like insertion and deletion, proper memory management, error handling, and ensuring the circular nature of the list.