**NAME:** AGNIV PRAMANICK **SECTION:** A **USN:** 1NT21IS017 **DATE:** 27/02/23

**Q. WRITE A PROGRAM TO IMPLEMENT CIRCULAR QUEUES USING LINKED LIST.**

**THEORY**

* Linked lists can also be used to build circular queues because the last member in a linked list always points back to the first, forming a circular structure. In a circular queue that is built on a linked list, the elements are kept in a linked list where each node has an element and a pointer to the one after it. The circular structure is created by the last node in the list pointing back to the initial node.
* We create a new node and modify its pointer to point to the first node in the list in order to add an element to the circular queue. The last node's pointer is then updated to point at the new node.
* Simply delete the first node from the list and change the final node's pointer to point to the new first node in the list to remove an element from the circular queue. As linked lists can dynamically expand and contract as necessary, utilising a circular queue based on them has the benefit of allowing for any number of elements.

* However, due to the computer's limited memory, linked list-based circular queues also have a restricted size, just like circular queues employing arrays. Consequently, to manage overflow or underflow issues, we could also need to apply a resizing approach.
* To remove a component from the circular queue, merely delete the first node from the list and modify the final node's pointer to link to the new first node in the list. Using a circular queue based on linked lists has the advantage of allowing for unlimited number of elements because they can dynamically expand and contract as necessary.
* However, just as circular queues using arrays, linked list-based circular queues also have a finite size due to the computer's limited memory. Thus, we could also need to use a resizing strategy to tackle overflow or underflow difficulties.

**ALGORITHM**

**For enqueue**

1. Create a new node with the given data
2. If the queue is empty, set both front and rear pointers to the new node and set the next pointer of the node to itself.
3. If the queue is not empty, set the next pointer of the new node to the node pointed by the rear pointer.
4. Set the next pointer of the node pointed by the front pointer to the new node.
5. Set the rear pointer to the new node.

**For dequeue**

1. If the queue is empty, print an error message and return.
2. If the queue has only one element, set both front and rear pointers to NULL and free the memory occupied by the node.
3. If the queue has more than one element, set the next pointer of the node pointed by the front pointer to the node next to it.
4. Free the memory occupied by the node pointed by the front pointer.
5. Set the front pointer to the node pointed by the next pointer of the previous front node.

**For display**

1. If the queue is empty, print an appropriate message and return.
2. Start from the node pointed by the front pointer and print the data of each node.
3. Continue printing until the next pointer of the node is equal to the front pointer.

**CODE**

#include<stdio.h>

#include<stdlib.h>

// define the structure for each node in the linked list

struct Node {

int data;

struct Node \*next;

};

// define the structure for the circular queue

struct Queue {

struct Node \*rear, \*front;

};

// function to create an empty circular queue

struct Queue\* createQueue() {

struct Queue \*q = (struct Queue\*)malloc(sizeof(struct Queue));

q->rear = q->front = NULL;

return q;

}

// function to add an element to the circular queue

void enqueue(struct Queue \*q, int data) {

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

temp->data = data;

if(q->front == NULL) { // if the queue is empty

q->front = q->rear = temp;

q->rear->next = q->front; // create a circular queue by linking rear to front

return;

}

q->rear->next = temp; // add the new node at the end of the queue

q->rear = temp; // update the rear pointer

q->rear->next = q->front; // link rear to front to create a circular queue

}

// function to remove an element from the circular queue

void dequeue(struct Queue \*q) {

if(q->front == NULL) { // if the queue is empty

printf("\nQueue is empty");

return;

}

if(q->front == q->rear) { // if there is only one element in the queue

free(q->front);

q->front = q->rear = NULL;

} else {

struct Node \*temp = q->front; // remove the front node

q->front = q->front->next; // update the front pointer

q->rear->next = q->front; // link rear to front to create a circular queue

free(temp); // free the memory occupied by the removed node

}

}

// function to display the elements in the circular queue

void display(struct Queue \*q) {

if(q->front == NULL) { // if the queue is empty

printf("\nQueue is empty");

return;

}

struct Node \*temp = q->front; // start from the front node

printf("\nElements in the queue are: ");

do { // traverse the queue and print the data of each node

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

temp = temp->next;

} while(temp != q->front); // stop when we reach the front node again

}

// main function to test the circular queue

int main() {

struct Queue \*q = createQueue(); // create an empty queue

int choice, data;

while(1) {

printf("\n\n1. Enqueue");

printf("\n2. Dequeue");

printf("\n3. Display");

printf("\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d",&choice); // ask the user for choice

switch(choice) {

case 1:

printf("\nEnter data to enqueue: ");

scanf("%d",&data); // ask the user for data to be enqueued

enqueue(q,data); // add the element to the queue

break;

case 2:

dequeue(q); // remove the front element from the queue

break;

case 3:

display(q); // display the elements in the queue

break;

case 4: // to exit

exit(0);

default: //any invalid options

printf("invalid choice\n");

}

}

}

**OUTPUT**



