Experiment No.8
Implement Linear Queue ADT using Linked list
Name:
Roll No:
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 8: Linear Queue using Linked list Aim:

Implement Linear Queue ADT using Linked list

Objective:

Linear queue can be implemented using linked list for dynamic allocation. Linked list implementation gives flexibility and better performance to the linear queue.

Theory:

A linear queue implemented using an array has a limitation in that it can only handle a fixed number of data values, and this size must be defined at the outset. This limitation makes it unsuitable for cases where the data size is unknown. On the other hand, linear queue implemented using a linked list is more flexible and can accommodate an unlimited number of data values, making it suitable for variable-sized data. A queue that is implemented using a linked list will continue to operate in accordance with the FIFO principle. Front and rare pointers are used to insert and delete the elements from linear queue implemented using array.

Linear queue Operations using Linked List

To implement a Linear queue using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define a two Node pointers 'front' and ‘rare’ and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

enqueue(value) - Inserting an element into the Queue

Step 1 - Create a newNode with given value.

Step 2 - Check for two conditions one is whether queue is Empty (front ==rear == NULL) or the queue contains at least one element.

Step 3 - If it is Empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.

Step 4 -If the queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point the rear pointer to the newly created node ptr

```
rear -> next = ptr;
 rear = ptr;
```

dequeue() - Deleting an Element from a Queue

Step 1 - Check whether queue is Empty or not (top == NULL).

Step 2 - If the queue is empty, i.e., front==NULL, so we just print 'underflow' on the screen and exit.

Step 3 - If the queue is not empty, delete the element at which the front pointer is pointing. For deleting a node, copy the node which is pointed by the front pointer into the pointer ptr and make the front pointer point to the front's next node and free the node pointed by the node ptr. This can be done using the following statement:.

```
*ptr = front;
 front = front -> next;
 free(ptr);
```

display() - Displaying queue of elements

Step 1 - Check whether queue is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Queue is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with front.

Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the last node in the queue. (temp → next != NULL).

Step 5 - Finally! Display 'temp → data ---> NULL'.

Code:

```
//Stack implementation using Linked List

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
	int data;
	struct node * next;
};

struct node * top = NULL;

void push()
{
	int num;
	struct node * temp;
	temp = (struct node *)malloc(sizeof(struct node));
	printf("Enter number");
	scanf("%d", &num);
	temp->data = num;
	temp -> next = top;
	top = temp;
}

void pop()
{
	struct node * p= top;
	int num;
	if(top == NULL)
	{
		printf("Stack is empty");
	}
	else
	{
		num = p-> data;
		printf("The element deleted is %d",num);
		top = top -> next;
		p -> next = NULL;
		free (p);
	}
}

void peek()
{
	if(top == NULL)
	{
		printf("Stack is empty");
	}
	else
	{
```

```
                int num;
                num = top-> data;
                printf("The value is %d",num);
        }
}

void display()
{
        struct node * p= top;
        if(top == NULL)
        {
                printf("Stack is empty");
        }
        else
        {
                while(p != NULL)
                {
                        printf("%d\t", p->data);
                        p = p -> next;
                }
        }
}


void main()
{
        int choice;
        clrscr();
        do
        {
                printf("\n1.Push");
                printf("\n2. Pop");
                printf("\n3. Display");
                printf("\n4. Peek");
                printf("\n5. Exit");
                printf("\nEnter choice: ");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: push();
                        break;
                        case 2: pop();
                        break;
                        case 3: display();
                        break;
                        case 4: peek();
                        break;
                }
        }while( choice != 5);
}
```

Output:

```
1.Push
2. Pop
3. Display
4. Peek
5. Exit
Enter choice: 1
Enter number10

1.Push
2. Pop
3. Display
4. Peek
5. Exit
Enter choice: 1
Enter number20

1.Push
2. Pop
3. Display
4. Peek
5. Exit
Enter choice: 2
The element deleted is 20
```

Conclusion:

Write in detail about an application where Queue is implemented as linked list?

1) Application: Print Queue Management System

- **Purpose**: Manages multiple print jobs in order of arrival (FIFO).
- **Node Structure**: Each node contains job details (document name, user ID, pages) and a pointer to the next job.
- **Operations**:
  - **Enqueue**: Adds a job to the rear.
  - **Dequeue**: Removes the job from the front.
- **Advantages**: Dynamic size, efficient insertions/deletions, no overflow issues.

2)Application: Task Scheduling in Operating Systems

- **Purpose**: Manages tasks for CPU processing, ensuring fair execution order.
- **Node Structure**: Each node contains task details (task ID, priority, execution time) and a pointer to the next task.
- **Operations**:
  - **Enqueue**: Adds a task to the end of the queue.
  - **Dequeue**: Removes the task at the front for execution.
- **Advantages**: Dynamic memory usage, efficient task management, and flexibility in handling varying numbers of tasks.