Experiment No.8
Aim: Implement Linear Queue ADT using Linked list
Name: Nikhil Kamalaji Shingade
Roll no: 57
Date of Performance:
Date of Submission:

Experiment No. 8: Linear Queue using Linked list

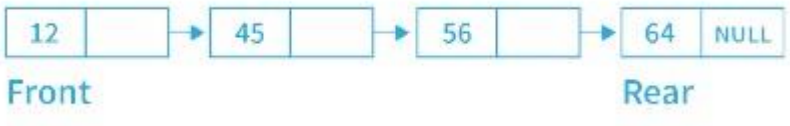
Aim: Implement Linear Queue ADT using Linked list

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:

```
#include <stdio.h>
#include <malloc.h>
struct node
{
   int data;
   struct node* next;
};
struct node *front=NULL;
struct node *rear=NULL;
void enqueue()
{
   int num;
   struct node*temp;
   temp=(struct node *)malloc(sizeof(struct node));
   printf("Enter value");
   scanf("%d",&num);
   temp->data=num;
   if (front==NULL && rear==NULL)
   {
      temp->next=NULL;
      front=temp;
      rear=temp;
   }
   else
   {
      rear->next=temp;
      rear=temp;
      temp->next=NULL;
   }
}
void dequeue() {
   int num;
   struct node *p;

   if (front == NULL) {
      printf("Queue is empty.\n");
      return;
   }

   p = front;
   num = front->data;
   printf("Delete: %d\n", num);

   front = front->next;
   if (front == NULL) {
      rear = NULL;
   }
```

```
   free(p);
}

void display()
{
   struct node*p;
      p=front;
   if(front==NULL && rear==NULL)
   {
      printf("Queue is empty");
   }
   else{
      while(p!=NULL){
   printf("%d\t",p->data);
   p=p->next;
}}}
int main()
{
   enqueue();
   enqueue();
   display();

   dequeue();
   display();
  getchar();
  return 0;
}
```

Output:

Output

```
/tmp/ejLkts4vcX.o
Enter value20
Enter value30
20  30  Delete: 20
30

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

Conclusion:

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

ANS:

Let's explore an application of a queue implemented as a linked list in C. We'll build a simple print job management system, where we can enqueue and dequeue print jobs using a linked list. Each job will be represented by a struct containing its details.

Node Structure: Each node contains a job ID, the document name, and a pointer to the next node.

Queue Structure: The queue keeps track of the front and rear of the queue.

Enqueue Function: Adds a new job to the end of the queue. If the queue is empty, it initializes both the front and rear pointers.

Dequeue Function: Removes a job from the front of the queue. If the queue becomes empty after the operation, it updates the rear pointer to NULL.

Display Function: Traverses the queue from front to rear, printing job details.

Main Function: Provides a simple user interface to add jobs, process jobs, and display the queue. It continues until the user decides to exit.