ASSIGNMENT 1

IMPLEMENTATION OF QUEUE DATA STRUCTURE

Subject: ESE_2025 EMBEDDED OPERATING SYSTEM

Submitted by:

Name	Student ID
Zain Anwar Rajani	C0752681

INTRODUCTION:

Queue is a type of data structure that is used to store and retrieve data and is used for processing the data as per the order. Queue is a widely used data structure. Queue just like its ordinary English meaning is like a queue for a bus or purchasing a ticket for a show, etc. Thus it follows “First-In-First-Out (FIFO)” data structure mechanism. The FIFO mechanism means which ever request entered into the system first will be serviced first and then second on the list will go into service and so on.

For example, when we go to an office for a service the person standing in line first gets his work done first and then the next person in line gets its work completed. Similarly, in programming we have queue. To explain this consider a system having n no. of peripheral devices connected to it and the system is capable of servicing only five devices at a time. All the request enter into the queue window and if more than five

r

equests are encountered at a time then the devices may be asked to wait and should be queued after the first request in line is completed.

In a data structure programming a queue will have a “Head” and “Tail”. These are nothing but pointers. A simple queue should be able to perform two actions Insertion of elements (a.k.a Enqueue) and deletion of elements (a.k.a Dequeue). During the process of enqueue the head is expected to keep on moving as the elements are inserted and the time when the queue limit (queue size) is reached it should return to its initial position and start with the either over-writing of elements or dequeue and then insert the elements. During the dequeue process the tail starts to increment as the elements are thrown out of the queue. A simple queue structure looks as seen below:

As we progress through this report we shall be implementing this basic queue structure using the C programming language.

Solution/Methods of Implementation:

To implement the basic queue structure where the enqueue operation occurs at rear (end) of the queue and the dequeue process occurs at the front (start) of the queue list. All the nodes in the list will be connected to each other in a sequential manner. We shall be using the pointer wherein, the pointer of the first node shall be pointing to the value of the second node and so on.

Let's first initialize the queue elements and pointers

```
queue *initialize(int
maxElements)
{
    /*Creation of Queue*/
    queue *q;
    q=(queue
*)malloc(sizeof(queue));
    /*Initialize Queue*/
    q->elements=(int *)malloc
(sizeof (int)*maxElements);
    q->size=0;
    q->capacity=maxElements;
    q->head=0;
    q->tail=-1;
    return q;
}
```

Pseudo Code:

1. Creation of a pointer to a queue
2. Initialize the head and tail of the queue.
3. Set the maximum size of the queue (capacity)
4. Create a temporary variable to determine if the queue is empty or full.

Here we have first initialized the pointers, head, tail, size of the queue which is the temporary size and the maximum size (window of the queue). This function sets the size/capacity of the queue the element passed as maxElement will decide the size of the queue. The head is kept one step ahead of the tail i.e. the head is pointing to first

block in the queue whereas the tail points somewhere behind at -1 position of the queue. The function queue *q creates the queue and queue* initializes the pointer to the queue. At this stage we have successfully initialized the queue.

Our next step includes to create functions that would perform the basic queue operations. Let us first look at how to insert the elements in the queue with the help of following code.

```
void enqueue (queue *q, int
element)
{
    if (q->size==q->capacity)
    {
       printf("\nQueue is
Full\n");
    }
    else
    {
       q->size++;
       q->tail++;
       if
(q->tail==q->capacity)
       {
            q->tail=0;
       }

q->elements[q->tail]=element;
       printf("Element %d is
inserted in queue\n",element);
    }
    return;
}
```

Pseudo Code:

Call the enqueue function from the main code using function name e.g enqueue (q,x)

Here q is the pointer to queue created and x is the element to be enqueued. Then,

```
If (current_queue_size==max_size)
        "Queue is full"
Else
        Current_queue_size+1;
        Queue_tail+1;
        If (Queue_tail==max_size)
                Queue_tail=0
        Queue[Queue.tail]=x;
```

The above code before inserting the element in the queue checks whether or not the queue is empty by comparing the current size of the queue (*variable size)* and the maximum size of the queue *(variable capacity)*. If both the variables are equal then it

shows that queue is full meaning no element can be inserted at the current time. If they are not equal then the process of inserting the element begins. For that, first the current size of the queue and tail is increment so that they point to the location where the element can be inserted.

But if after incrementing the tail if the value is equal to the capacity of queue then the tail is passed to its initial position. If the tail is also not at its maximum queue length then the element gets inserted into the queue with elements array and position being pointed by the queue q pointing to the tail of the queue.

The next operation is to delete the element in queue (a.k.a Dequeue).

```
void dequeue(queue *q)
{
    /* If Queue is Empty i.e.
Size =0 */
    if ((*q).size==0)
    {
       printf("\nQueue is
Empty");
       return;
    }
    else
    {
       int item;
item=q->elements[(q->head)];
       printf("\nElement %d
request is completed; Hence it
is deleted\n",item);
       q->size--;
       q->head++;
       if (q->head==q->tail)
       {
             q->head=0;
       }
    }
}
```

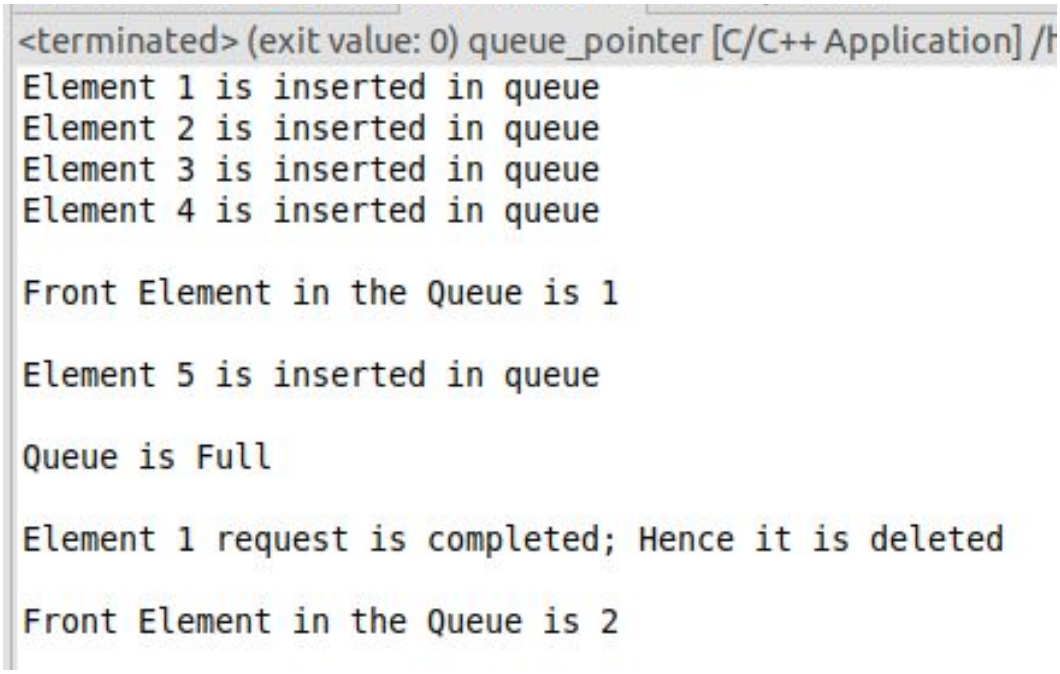
Pseudo Code:

```
dequeue(q)
        if (size==0)
                "Queue is Empty"
        else
                x=q[q.head];
                head+1;
                if (head==tail)
                        head=0
                return x;
```

A queue will delete element (a.k.a Dequeue) will not occur if the head and tail are on the same position i.e. head=tail. Also, another way to find if the queue is empty is by checking the current size of the queue, if the current size of queue is 0 that means that queue is empty. If the current size of queue or if head and tail are not equal then we delete the element currently pointed by the head and the head pointer is then incremented by 1 to point to the next value in the queue.

Results:

After writing and compiling the code for basic queue operation we see the following output on the screen. We have set the maximum size of queue to be 5.



As we see that first we insert 4 elements in the queue and then we check which is the first element in the queue and print that element. As we have already said that queue follows the FIFO data structure thus as we have inserted 1 in the first place that should be front of the queue; which the result screen proves that. Once the queue length is reached a message of “Queue is Full” is displayed. As the queue becomes full we start to delete the elements and thus the second element in the queue will

become the first element in the queue. Thus since the first element 1 is deleted from the queue and thus the second element in the queue i.e. 2 becomes the first element of the queue, which is proved from the screen.

Improvements/Suggestions:

We have just implemented a simple queue which just inserts and deletes the element. After the queue reaches maximum size and after this we need to initialize the queue once again for the second batch of 5 other jobs. We could optimize the code in the following ways:

1. We could create a circular queue wherein, once the maximum queue limit is reached and the first element in the queue is serviced then the next element waiting to enter the queue should be inserted in the queue immediately.
2. We could make a priority queue where in we could make the highest element in the queue as the first element so that most important task gets completed first and the hierarchy follows. For this purpose we can use the binary tree concepts
3. Last improvement could be accessing the queue from both the ends. It would be like two people assisting others with their queries standing in the line. This would mean that a faster processing of services as we can delete and insert elements from both the ends. We could call it as “Two sided queue”

```
void enqueueR(dequeue *P,int x)
{
      if(empty(P))
      {
            P->rear=0;
            P->front=0;
            P->data[0]=x;
      }
      else
      {
            P->rear=(P->rear+1)%MAX;
```

```
            P->data[P->rear]=x;
      }
}

void enqueueF(dequeue *P,int x)
{
      if(empty(P))
      {
            P->rear=0;
            P->front=0;
            P->data[0]=x;
      }
      else
      {
            P->front=(P->front-1+MAX)%MAX;
            P->data[P->front]=x;
      }
}

int dequeueF(dequeue *P)
{
      int x;

      x=P->data[P->front];
```

```
      if(P->rear==P->front) //delete the last element
            initialize(P);
      else
            P->front=(P->front+1)%MAX;

      return(x);
}

int dequeueR(dequeue *P)
{
      int x;

      x=P->data[P->rear];
```

```
      if(P->rear==P->front)
            initialize(P);
      else
            P->rear=(P->rear-1+MAX)%MAX;

      return(x);
}
```

Appendix

```
/*
 *  Program Name: queue_pointer.c
 *    Program Description: Perform Basic Queue Operation in C using
pointers
 *  Created on: October 17, 2019
 *  Author: Zain Rajani
 *
 */
/* Pre-processor Directives */
#include<stdio.h>
#include<stdlib.h>
```

```c
/* Queue Elements Declaration */
typedef struct queue
{
    int capacity;
    int size;
    int head;
    int tail;
    int *elements;
}queue;
queue *initialize(int maxElements)  /* Initialize the queue */
{
    /*Creation of Queue*/
    queue *q;
    q=(queue *)malloc(sizeof(queue));
    /*Initialize Queue*/
    q->elements=(int *)malloc (sizeof (int)*maxElements);
    q->size=0;  //Current size of the queue
    q->capacity=maxElements;  // Maximum size of the queue
    q->head=0;  //Head of the queue
    q->tail=-1;    // Tail of the queue
    return q;
}

void dequeue(queue *q)  /* Dequeue Operation */
{
    /* If Queue is Empty i.e. Size =0 */
    if ((*q).size==0)
    {
       printf("\nQueue is Empty");
       return;
    }
    else
    {
       int item;
       item=q->elements[(q->head)]; //Current Element in the queue
       printf("\nElement %d request is completed; Hence it is
deleted\n",item);
```

```
      q->size--;
      q->head++;
      if (q->head==q->tail) //if head and tail point at the same
position
      {
            q->head=0;
      }
    }
}
int head(queue *q)  // Check which element is pointed by the head
{
    if (q->size==0)
    {
      printf("\nQueue is Empty");
      exit(0);
    }
    return q->elements[q->head]; // Currently pointed by the head
}

void enqueue (queue *q, int element) /* Insert element in the queue
*/
{
    if (q->size==q->capacity) // If Capacity is reached
    {
      printf("\nQueue is Full\n");
    }
    else
    {
      q->size++;
      q->tail++;
      if (q->tail==q->capacity) //If tail is reached at the max.
capacity
      {
            q->tail=0;
      }
      q->elements[q->tail]=element; //Element Inserted
      printf("Element %d is inserted in queue\n",element);
    }
```

```
    return;
}

int main()
{
    queue *q=initialize(5); //Initialize size of the queue
    /* Call the function for inserting the elements in the queue */
    enqueue(q,1);
    enqueue(q,2);
    enqueue(q,3);
    enqueue(q,4);
    /* print the element pointed by the head */
    printf("\nFront Element in the Queue is %d\n\n",head(q));
    enqueue(q,5);
    enqueue(q,6);
    /* Call the function to delete the element in the queue */
    dequeue(q);
    /* print the element pointed by the head */
    printf("\nFront Element in the Queue is %d",head(q));
    return 0;
}
```