# Lab 2

**Queue Interpretation**

## 1.1 Theory

### 1.1.1 Queue:

A queue is the data structure that follows the FIFO input output structure. This data structure allows for flow of data in an orderly manner for implementations like video's data storage during buffering processes, or for loading the backends of various instances. It models a collection of elements with the following operations:

Enqueue: Adding an element to the queue. Dequeue: Removing the element first in the queue.

The extra operations, generally added are: Peek: Viewing the element next in the queue. Show all: View the entire queue.

Queues are a programmer's best friend when it comes to ordering the data when in a buffer state. They can also be used in multi-threading data streams. There are many different types of queues. Some of the types are

1. Circular Queue
2. Double Ended Queue
3. Priority Queue
4. Etc.

## 1.2 Pseudo Code:

* For Enqueue

enqueue():  
 if pos == maxsize - 1:  
 print "Stack Overflow"  
 else :  
 for i to maxsize:  
 queue[i+1] = queue[i]  
 queue[0] = val  
 pos++

* For Dequeue

dequeue():  
 if pos == -1:  
 print "Stack Underflow"  
 else :  
 print queue[pos]  
 pos--

* For Peek

peek():  
 if pos == -1:  
 print "Stack Empty"  
 else :  
 print queue[pos]

* For Showall
* showState(){  
   print "["  
   for (i = 0; i < size; i++)  
   print arr[i]  
   print "]"  
  }

## 1.3 Source Code:

#include <iso646.h>  
#include <stdio.h>  
  
void enqueue(int\* queue, int\* pos, int maxsize, int val ) {  
  
 if (\*pos == maxsize-1)  
 printf("Stack Overflow!");  
  
 else {  
 // shift all elements to right  
 int i;  
  
 for (i=\*pos ; i >= 0 ; i--)  
 queue[i+1] = queue[i] ;  
  
 queue[0] = val;  
 (\*pos)++;  
 }  
  
}  
  
void dequeue(int\* queue, int\* pos) {  
  
 if (\*pos == -1)  
 printf("Stack Underflow!") ;  
  
 else  
 printf("Dequeued: %d", queue[(\*pos)--]);  
  
}  
  
void peek(int\* queue, int\* pos) {  
  
 if (\*pos == -1)  
 printf("Stack Empty!") ;  
  
 else  
 printf("The Peek: %d", queue[(\*pos)]);  
  
}  
  
void showState(int\* point, int\* arr, int size) {  
 printf("[");  
 int i, corr=0, n;  
  
 for (i = 0; i < size; i++)  
 printf(" %d", arr[i]);  
  
 printf("]\n");  
  
 for (i = 0; i < (\*point)+1; i++) {  
 //count no of digits in arr[point]  
 n = arr[i];  
 corr=0;  
  
 while (n!=0) {  
 n=n/10;  
 corr++;  
 }  
  
 int j;  
  
 printf(" ");  
  
 for (j=0; j<corr; j++)  
 printf(" ");  
  
 }  
  
 printf("^");  
}  
  
int main() {  
 char existence;  
 int maxsize = 10, val ;  
 int queue[20], noexit=1, inp, pos=-1;  
 while (noexit) {  
 printf("\nChoose a number.\n") ;  
 printf("1) Enqueue a value. \n");  
 printf("2) Dequeue a value. \n");  
 printf("3) Peek Front Value. \n");  
 printf("4) Display all Elements. \n");  
  
 scanf("%d", &inp);  
  
 switch (inp) {  
 case 1:  
 printf("Value to Enqueue: ");  
 scanf("%d", &val);  
  
 enqueue(queue, &pos, maxsize, val);  
 break;  
  
 case 2:  
 dequeue(queue, &pos);  
 break;  
  
 case 3:  
 peek(queue, &pos);  
 break;  
  
 case 4:  
 showState(&pos, queue, maxsize);  
 break;  
 }  
  
 printf("\nDo we exit?(y/n) ");  
 getchar();  
 scanf("%c", &existence);  
 noexit = existence == 'y' ? 0 : 1;  
  
 }  
  
 return 0;  
}

## 1.4 Discussion

The laboratory experiment focused on the implementation of fundamental queue operations using a simplistic array-based approach. The code includes functions for enqueueing, dequeuing, peeking at the front element, and displaying the state of the queue. Let's delve into the key aspects of the implementation. The enqueue function is designed to add an element to the rear of the queue. In the case of a full queue, the program prevents a stack overflow by shifting all existing elements one position to the right and then adding the new element at the front. While this approach ensures the preservation of order, it is worth noting that it results in a linear time complexity proportional to the number of elements in the queue. The dequeue function removes the front element from the queue. A check for an empty queue prevents a stack underflow situation, and the front element is dequeued by printing its value. The position pointer is appropriately decremented to reflect the removal. The peek function allows the user to view the front element of the queue without removing it. Similar to the dequeue operation, it checks for an empty queue and provides feedback accordingly. The showState function was implemented to visualize the current state of the queue. It prints the entire queue array and includes a pointer (^) indicating the front of the queue. To enhance visual clarity, additional spaces are added to align the pointer correctly with the front element.

The main function orchestrates the execution, utilizing an array to store queue elements and a position pointer (pos) to keep track of the front of the queue. The user is presented with a menu to choose enqueue, dequeue, peek, or display operations, allowing for dynamic interaction with the queue. The program continues to execute operations until the user chooses to exit.

Future enhancements might involve exploring alternative data structures for implementing a queue, such as a linked list, to achieve more efficient enqueue and dequeue operations. Additionally, error handling mechanisms and considerations for dynamic resizing could be further explored.

In summary, the implementation successfully demonstrates the basic principles of queue operations, providing a hands-on exploration of key functionalities in a simple, interactive setting.