QUEUES

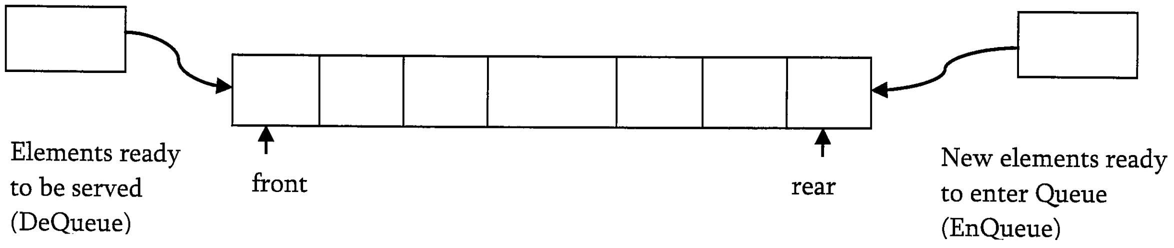
Chapter-5

5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which the data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition: A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called as First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called as *EnQueue*, and when an element is removed from the queue, the concept is called as *DeQueue*. Trying to *DeQueue* an empty queue is called as *underflow* and trying to *EnQueue* an element in a full queue is called as *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.


Elements ready to be served (DeQueue)
front
rear
New elements ready to enter Queue (EnQueue)

5.2 How is Queues Used?

Line at reservation counter explains the concept of a queue. When we enter the line we put ourselves at the end of the line and the person who is at the front of the line is the next who will be served. The person will exit the queue and will be served.

In the meanwhile the queue is served and next person at head of the line will exit the queue and will be served. While the queue is served, we will move towards the head of the line since each person that is served will be removed from the head of the queue. Finally we will reach head of the line and we will exit the queue and be served. This behavior is very useful in any cases where there is need to maintain the order of arrival.

5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in queue must follow the FIFO scheme. For simplicity we assume the elements are integers.

Main Queue Operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations

- int Front(): Returns the element at the front without removing it

- int QueueSize(): Returns the number of elements stored
- int IsEmptyQueue(): Indicates whether no elements are stored

5.4 Exceptions

As similar to other ADTs attempting execution of *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and attempting execution of *EnQueue* on an full queue throws an "*Full Queue Exception*".

5.5 Applications

Following are the some of the applications in which queues are being used.

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

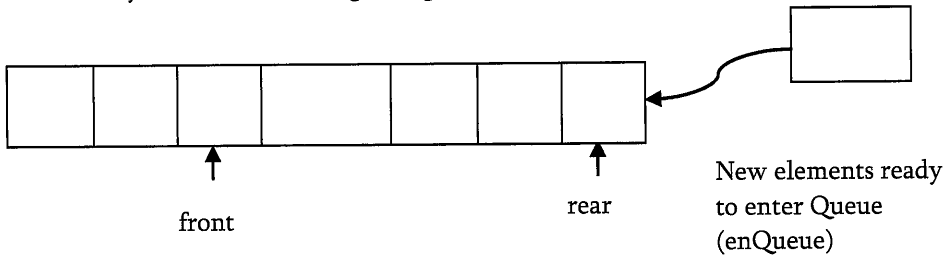
5.6 Implementation

There are many ways (similar to Stacks) of implementing queue operations and below are commonly used methods.

- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked lists implementation

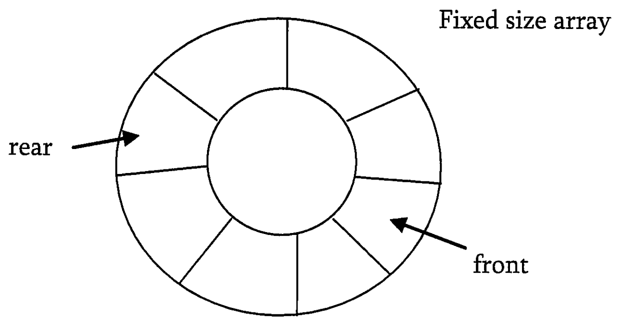
Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues which we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at other end. After some insertions and deletions it is easy to get the situation as shown below. It can be seen clearly that, the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat last element and first array elements are contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.



Note: The simple circular array and dynamic circular array implementations are very much similar to stack array implementations. Refer *Stacks* chapter for analysis of these implementations.

Simple Circular Array Implementation



This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue. The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from empty queue then it will throw *empty queue exception*.

Note: Initially, both front and rear points to -1 which indicates that the queue is empty.

```
struct ArrayQueue {
        int front, rear;
        int capacity;
        int *array;
};
struct ArrayQueue *Queue(int size) {
        struct ArrayQueue *Q = malloc(sizeof(struct ArrayQueue));
        if(!Q) return NULL;
        Q→capacity = size;
        Q→front = Q→rear = -1;
        Q→array= malloc(Q→capacity * sizeof(int));
        if(!Q→array)
             return NULL;
        return Q;
}
int IsEmptyQueue(struct ArrayQueue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == -1);
}
int IsFullQueue(struct ArrayQueue *Q) {
        //if the condition is true then 1 is returned else 0 is returned
        return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
        return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct ArrayQueue *Q, int data) {
        if(IsFullQueue(Q))
                printf("Queue Overflow");
        else {  Q→rear = (Q→rear+1) % Q→capacity;
                Q→ array[Q→rear]= data;
                if(Q→front == -1)
```

```
                        Q→front = Q→rear;
            }
}
int DeQueue(struct ArrayQueue *Q) {
            int data = 0;//or element which does not exist in Queue
            if(IsEmptyQueue(Q)) {
                        printf("Queue is Empty");
                        return 0;
            }
            else {      data = Q→array[Q→front];
                        if(Q→front == Q→rear)
                                    Q→front = Q→rear = -1;
                        else        Q→front = (Q→front+1) % Q→capacity;
            }
            return data;
}
void DeleteQueue(struct ArrayQueue *Q) {
            if(Q) {
                  if(Q→array)
                                    free(Q→array);
                  free(Q);
            }
}
```

Performance & Limitations

Performance: Let *n* be the number of elements in the queue:

Space Complexity (for *n* EnQueue operations)	O(*n*)
Time Complexity of EnQueue()	O(1)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Limitations: The maximum size of the queue must be defined a prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

Dynamic Circular Array Implementation

```
struct DynArrayQueue {
            int front, rear;
            int capacity;
            int *array;
};
struct DynArrayQueue *CreateDynQueue() {
            struct DynArrayQueue *Q = malloc(sizeof(struct DynArrayQueue));
            if(!Q) return NULL;
            Q→capacity = 1;
            Q→front = Q→rear = -1;
            Q→array= malloc(Q→capacity * sizeof(int));
            if(!Q→array)
```

```
                return NULL;
        return Q;
}
int IsEmptyQueue(struct DynArrayQueue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == -1);
}
int IsFullQueue(struct DynArrayQueue *Q) {
        //if the condition is true then 1 is returned else 0 is returned
        return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
        return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct DynArrayQueue *Q, int data) {
        if(IsFullQueue(Q))
                ResizeQueue(Q);
        Q→rear = (Q→rear+1)% Q→capacity;
        Q→ array[Q→rear]= data;
        if(Q→front == -1)
                Q→front = Q→rear;
}
void ResizeQueue(struct DynArrayQueue *Q) {
        int size = Q→capacity;
        Q→capacity = Q→capacity*2;
        Q→array = realloc (Q→array, Q→capacity);
        if(!Q→array) {
                printf("Memory Error");
                return;
        }
        if(Q→front > Q→rear ) {
                for(int i=0; i < Q→front; i++) {
                        Q→array[i+size] =Q→array[i];
                }
                Q→rear = Q→rear + size;
        }
}
int DeQueue(struct DynArrayQueue *Q) {
        int data = 0;//or element which does not exist in Queue
        if(IsEmptyQueue(Q)) {
                printf("Queue is Empty");
                return 0;
        }
        else {  data = Q→array[Q→front];
                if(Q→front== Q→rear)
                        Q→front= Q→rear = -1;
                else
                        Q→front = (Q→front+1) % Q→capacity;
        }
        return data;
```

```
}
void DeleteQueue(struct DynArrayQueue *Q) {
        if(Q) {
                if(Q→array)
                        free(Q→array);
                free(Q→array);
        }
}
```

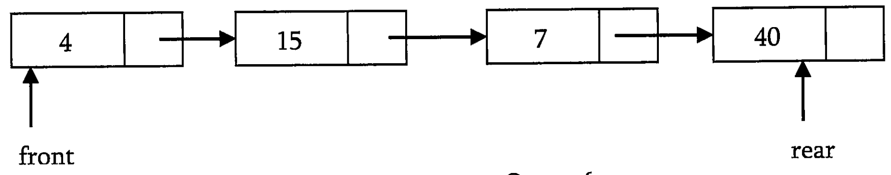
Performance

Let n be the number of elements in the queue.

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Linked List Implementation

The other way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting element at the ending of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.



```
struct ListNode {
   int data;
   struct ListNode *next;
};
```

```
struct Queue {
   struct ListNode *front;
   struct ListNode *rear;
};
```

```
struct Queue *CreateQueue() {
        struct Queue *Q;
        struct ListNode *temp;
        Q = malloc(sizeof(struct Queue));
        if(!Q) return NULL;
        temp = malloc(sizeof(struct ListNode));
        Q→front = Q→rear = NULL;
        return Q;
}
int IsEmptyQueue(struct Queue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == NULL);
}
void EnQueue(struct Queue *Q, int data) {
        struct ListNode *newNode;
        newNode = malloc(sizeof(struct ListNode));
        if(!newNode)
                return NULL;
        newNode→data = data;
        newNode→next = NULL;
        Q→rear→next = newNode;
        Q→rear = newNode;
        if(Q→front == NULL)
                Q→front = Q→rear;
}
int DeQueue(struct Queue *Q) {
        int data = 0;     //or element which does not exist in Queue
        struct ListNode *temp;
        if(IsEmptyQueue(Q)) {
                printf("Queue is empty");
                return 0;
        }
        else {  temp = Q→front;
                data = Q→front→data;
                Q→front== Q→front→next;
                free(temp);
        }
        return data;
}
void DeleteQueue(struct Queue *Q) {
        struct ListNode *temp;
        while(Q) {
                temp = Q;
                Q = Q→next;
                free(temp);
        }
        free(Q);
}
```

Performance

Let n be the number of elements in the queue, then

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

Comparison of Implementations

Note: Comparison is very much similar to stack implementations and *Stacks* chapter.

5.7 Problems on Queues

Problem-1 Give an algorithm for reversing a queue Q. To access the queue, we are only allowed to use the methods of queue ADT.

Solution:
```
void ReverseQueue(struct Queue *Q) {
        struct Stack *S = CreateStack();
        while (!IsEmptyQueue(Q))
                Push(S, DeQueue(Q))
        while (!IsEmptyStack(S))
                EnQueue(Q, Pop(S));
}
```
Time Complexity: O(n).

Problem-2 How to implement a queue using two stacks?

Solution: Let S1 and S2 be the two stacks to be used in the implementation of queue. All we have to do is to define the EnQueue and DeQueue operations for the queue.
```
struct Queue {
        struct Stack *S1; // for EnQueue
        struct Stack *S2; // for DeQueue
}
```
EnQueue Algorithm

- Just push on to stack S1

```
void EnQueue(struct Queue *Q, int data) {
        Push(Q→S1, data);
}
```
Time Complexity: O(1).

DeQueue Algorithm

- If stack S2 is not empty then pop from S2 and return that element.
- If stack is empty, then transfer all elements from S1 to S2 and pop the top element from S2 and return that popped element [we can optimize the code little by transferring only $n - 1$ elements from S1 to S2 and pop the n^{th} element from S1 and return that popped element].
- If stack S1 is also empty then throw error.

```
int DeQueue(struct Queue *Q) {
        if(!IsEmptyStack(Q→S2))
                return Pop(Q→S2);
        else {  while(!IsEmptyStack(Q→S1))
                        Push(Q→S2, Pop(Q→S1));
                return Pop(Q→S2);
        }
}
```
Time Complexity: From the algorithm, if the stack S2 is not empty then the complexity is O(1). If the stack S2 is empty then, we need to transfer the elements from S1 to S2. But if we carefully observe, the number of transferred elements and the number of popped elements from S2 are equal. Due to this the average complexity of pop operation in this case is O(1). Amortized complexity of pop operation is O(1).

Problem-3 Show how to efficiently implement one stack using two queues. Analyze the running time of the stack operations.

Solution: Let Q1 and Q2 be the two queues to be used in the implementation of stack. All we have to do is to define the push and pop operations for the stack.
```
struct Stack {
        struct Queue *Q1;
        struct Queue *Q2;
}
```

In below algorithms, we make sure that one queue is empty always.

Push Operation Algorithm: Whichever is queue is not empty, push the element into it.

- Check whether queue Q1 is empty or not. If Q1 is empty then Enqueue the element into Q2.
- Otherwise EnQueue the element into Q1.

```
Push(struct Stack *S, int data) {
        if(IsEmptyQueue(S→Q1))
                EnQueue(S→Q2, data);
        else    EnQueue(S→Q1, data);
}
```

Time Complexity: O(1).

Pop Operation Algorithm: Transfer $n - 1$ elements to other queue and delete last from queue for performing pop operation.

- If queue Q1 is not empty then transfer $n - 1$ elements from Q1 to Q2 and then, DeQueue the last element of Q1 and return it.
- If queue Q2 is not empty then transfer $n - 1$ elements from Q2 to Q1 and then, DeQueue the last element of Q2 and return it.

```
int Pop(struct Stack *S) {
        int i, size;
        if(IsEmptyQueue(S→Q2)) {
                size = size(S→Q1);
                i = 0;
                while(i < size-1) {
                        EnQueue(S→Q2, DeQueue(S→Q1));
                        i++;
                }
                return DeQueue(S→Q1);
        }
        else {  size =   size(S→Q2);
                while(i < size-1) {
                        EnQueue(S→Q1, DeQueue(S→Q2));
                        i++;
                }
                return DeQueue(S→Q2);
        }
}
```

Time Complexity: Running time of pop operation is $O(n)$ as each time pop is called, we are transferring all the elements from one queue to other.

Problem-4 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	**Max**
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5

1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output**: An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]. **Requirement**: Find a good optimal way to get B[i]

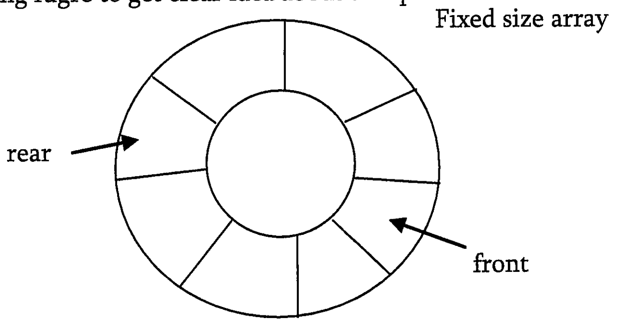
Solution: This problem can be solved with doubly ended queue (which support insertion and deletions at both ends). Refer *Priority Queues* chapter for algorithms.

Problem-5 Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue and push and pop operations for the stack, outline an efficient O(n) algorithm to accomplish the above task, using only a constant amount of additional storage.

Solution: Assume the elements of queue Q are $a_1, a_2 \ldots a_n$. Dequeuing all elements and pushing them onto the stack will result in a stack with an at the top and a_1 at the bottom. This is done in O(n) time as dequeue and push each require constant time per operation. The queue is now empty. By popping all elements and pushing them on the the queue we will get a_1 at the top of the stack. This is done again in O(n) time. As in big-oh arithmetic we can ignore constant factors, the process is carried out in O(n) time. The amount of additional storage needed here has to be big enough to temporarily hold one item.

Problem-6 A queue is set up in a circular array A[0..n - 1] with front and rear defined as usual. Assume that $n - 1$ locations in the array are available for storing the elements (with the other element being used to detect full/empty condition). Give a formula for the number of elements in the queue in terms of $rear$, $front$, and n.

Solution: Consider the following fugre to get clear idea about the queue.



- Rear of the queue is somewhere clockwise from the front
- To enqueue an element, we move rear one position clockwise and write the element in that position
- To dequeue, we simply move front one position clockwise
- Queue migrates in a clockwise direction as we enqueue and dequeue
- Emptiness and fullness to be checked carefully.
- Analyze the possible situations (make some drawings to see where $front$ and $rear$ are when the queue is empty, and partially and totally filled). We will get this:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if rear == front} \\ rear - front + n & \text{otherwise} \end{cases}$$