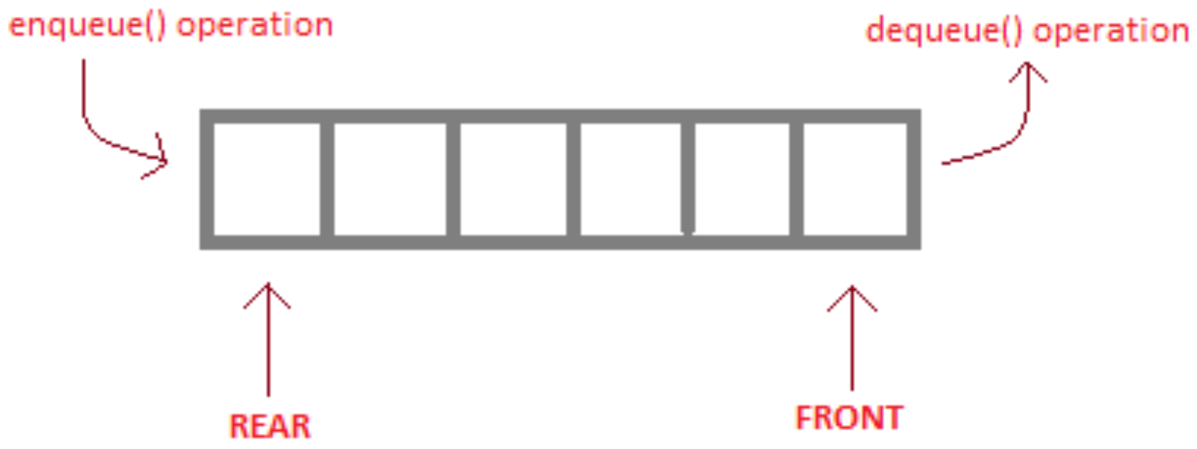
Queue

ADT QUEUE

- Queue is also an abstract data type or a linear data structure, in which the element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head).
- This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

Basic features of Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- **peek( )** function is often used to return the value of first element without dequeuing it.

ADT QUEUE operations

- **init()** :Initialize a queue to be empty
- **enQueue():** Insert a new element after the last element in a queue, if the queue is not full
- **deQueue():**Delete the first element in a queue, if it is not empty
- **isempty():** Check whether a queue is empty or not
- **isfull() :**Check whether a queue is full or not
- **peek():** Retrieve the first element of the queue, if it is not empty

Applications of Queue

- Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :
- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.
- File servers: Users needing access to their files on a shared file server machine are given access on a FIFO basis

- Threads
- Job scheduling (e.g. Round-Robin algorithm for CPU allocation)

- Queue can be implemented using an
- Array
- Linked List
- And ..

Array implementation of (circular) Queue using count

q.h

```
#define N 16
typedef struct queue {
        char *arr[N];
        int count, front, rear;
}queue;
void qinit(queue *q);
void enqueue(queue *q, char *name);
char *dequeue(queue *q);
int isqempty(queue *q);
int isqfull(queue *q);
void printq(queue *q);
```

q.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "q.h"

void qinit(queue *q){
	q->front= q->rear = q->count = 0;
}
```

```
void enqueue(queue *q, char *name){
        q->arr[q->rear] = malloc(strlen(name) + 1);
        strcpy(q->arr[q->rear], name);
        q->rear = ((q->rear) + 1) % N;
        (q->count)++;
}
```

```
char *dequeue(queue *q){
        char *tmp = q->arr[q->front];
        q->front = ((q->front) + 1) % N;
        (q->count)--;
        return tmp;
}
```

```
int isqempty(queue *q){
        return q->count == 0;
}

int isqfull(queue *q){
        return q->count == N;
}
```

```
void printq(queue *q){
        int x;
        printf("[ ");
        //for(x = 0; x < q->count; x++) {
                //printf("%s ", q->arr[(x + q->i) % N]);
                for(x = q->j ; x != q->i;  x = (x + 1)  % N) {
                printf("%s ", q->arr[x]);
                }
        printf("] \n");
}
```

prog.c

```
#include <stdio.h>
#include <stdlib.h>
#include "q.h"
int printmenu() {
        int choice;
        printf("1. enqueue \n2. dequeue \n3. exit \n");
        scanf("%d",&choice);
        return choice;
}
```

```
int main() {
        queue q;
        char name[16];
        int choice;
        qinit(&q);
        while(1) {
```

```
printq(&q);
                choice = printmenu();
                switch(choice) {
                case 1: /* insert */
                        printf("Enter name: \n");
                        scanf("%s", name);
                        if(!isqfull(&q))
                                enqueue(&q, name);
                        else
                                 printf(“Sorry\n");
                        break;
```

```
case 2: /* dequeue */
        if(!isqempty(&q))
                printf(“Patient name: %s \n",
                        dequeue(&q));
        else
                printf("No more patients\n");
                        break;
```

```
case 3: /* exit */
                    exit(0);
            }
            }
}
```

Linked list implementation of Queue

```
typedef struct node{
                int data;
                node* next;
}node;

typedef struct queue{
         node* front,* rear;
}queue;
```

Enqueue()

```
void enqueue(queue *q, int d)
{
     node* temp = (node*) malloc(sizeof (node));
     if (!temp)
         return;
     temp->data = d;
     temp->next= NULL;
     if (*q->front)
              (*q->rear) -> next= temp;
     else
              *q->front = temp;
     *q->rear = temp;
  }
```

Dequeue

```
int dequeue(queue *q) {
    node* temp = front;
    int d;
    d = temp->data;
    *q->front = temp->next;
     free(temp);
     return d;
}

//dequeu() should be called only if queue is not empty
```

```
void initQ(Queue *q){
       *q->front = NULL;
        *q->rear =NULL;
}

int isQfull(Queue *q){
         return 0;
}
int isQempty(Queue *q){
        if(*q->front)
          return 0;
        return 1;
}
```

Queue using Stack!

We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.

HOW?

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly)

- This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x)

  1) While stack1 is not empty, push everything from stack1 to stack2.

  2) Push x to stack1 (assuming size of stacks is unlimited).

  3) Push everything back to stack1.

Here time complexity will be O(n)

deQueue(q)

  1) If stack1 is empty then error

  2) Pop an item from stack1 and return it

Here time complexity will be O(1)

Method 2 (By making deQueue operation costly)

- In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q,  x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
Here time complexity will be O(n)
```

How can you..

Implement Stack using Queues

We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.