CS 3137, Class Notes

1 THE QUEUE ADT

1. A queue is an ADT similar to a stack, but it uses a First-In, First-Out protocol (FIFO), like a line at a movie theater.
2. Queues are used quite often in operating systems where jobs need to line up to share a limited resource.
   - Execute queue: queue of processes waiting to use a single CPU
   - Disk queue: queue of users seeking data on a single disk
   - Print queue: queue of jobs waiting to be printed.
3. Queues are often used in simulation programs to keep track of events that are waiting to happen. Examples are takeoffs and landings at an airport, or traffic studies simulations.
4. Queues have a desginated designated *front* or (*head*) and a *rear* (or *tail*)
5. The usual primitive operations on a queue are:
   - Initialize_Queue
   - Enqueue: inserts a node at the rear of the list
   - Dequeue: removes a node from the front of the list
   - Is_Empty: checks to see if the queue is empty
   - Is_Full: checks to see if the queue is full

2 LINKED LIST IMPLEMENTATION OF QUEUES

1. We need 2 pointers: one pointing to rear for insertions, one pointing to front of list for removals.
2. Another method uses a **circularly linked list** for a queue. A circularly linked list has its last node (tail node) always pointing to its first node (head node). See code on class notes page.
3. If we use this method, then we only need to provide a SINGLE pointer to the tail node of the queue. This allows us to access the tail node directly, and if we follow the tail node's NEXT pointer, we will be pointing to the head node.
4. To see if queue is EMPTY, we test for NULL pointer to queue. Since we are using dynamically allocated lists, there is no notion of QUEUE_FULL

3 ARRAY IMPLEMENTATION OF QUEUES

1. As with most ADT's, we can implement them using fixed length arrays or linked lists. Arrays are faster to access and easier to program, but at a cost of allocating‘ the space at compile time; we need to know beforehand just how large the queue must be. Linked lists provide dynamic growing and shrinking of the queue, at the (small) extra cost of a pointer cell in each node, and slower access as pointers are followed.

2. Initially, 0 is the array position of the first element added to the queue, 1 is the position of the second element etc. But as we start to also remove items, the queue “moves” within the array.

3. If the queue is an array of 10 elements (array positions 0 to 9), then at some point the queue front might be 7 and the queue rear 2 - the queue “wraps around”. You can think of this as a “circular array”.

4. As we reach the last element in the array (i.e. positon 9), the next elements to be added will be added at positions 0,1,2,...etc.

5. All of this works fine by using modulo arithmetic, where the modulus is the number of queue elements.

   Front=(Front+1)%Queuesize , Rear=(Rear+1)%Queuesize

6. An enqueue operation will increment the counter for the number of elements in the queue, and increment the rear index to place the new element in the back of the queue.

7. We have to be careful about the queue “wrapping around itself” - letting the rear element pass the front element. This is caused by trying to put too many items in the queue. To prevent overfilling the queue, we use a simple counter for the number of elements in the queue, and check we never exceed this.

8. If we find the queue is filled up, we can always write a method to increase the queue size dynamically by increasing the array size.

We can simply extend the Java Collections LinkedList() class we used for Stacks to create a queue class. Below is the code:

```
// Simple Queue using a Linked List type Data Structure

import java.util.*;
public class easyQueue{
        private LinkedList queue;
        public easyQueue( )   {
           queue=new LinkedList();
        }

        public boolean isEmpty(){
            if(queue.isEmpty()) return true;
             else return false;
       }
       public void enQueue(Object i) {
           queue.addLast(i);
       }

       public Object deQueue() {
            if (queue.isEmpty()){
              System.out.println("Queue is empty!");
              return null;
           } else {
               Object result=queue.getFirst();
               queue.removeFirst();
               return result;
           }
       }

       public void printQueue()  {
           System.out.println("the Queue is: " + queue);
        }

       public static void main( String [ ] args ) {
           easyQueue Q = new easyQueue();
           Object x;  int i;
           Q.printQueue();  //should print empty queue message

           for( i = 0; i < 5; i++ ) {
                System.out.print("Enqueued " + i + ". Here is ");
                Q.enQueue( new Integer( i ));
                Q.printQueue();
           }
           System.out.println( "do some dequeues..." );
           while(!Q.isEmpty()) {
                 x = Q.deQueue();
                 System.out.print( "Dequeued " +(Integer)x +". Here is ");
                 Q.printQueue();
           }
           Q.deQueue();  //should print error: dequeue from empty queue
       }
}
Here is the output:
the Queue is: []
Enqueued 0. Here is the Queue is: [0]
Enqueued 1. Here is the Queue is: [0, 1]
Enqueued 2. Here is the Queue is: [0, 1, 2]
Enqueued 3. Here is the Queue is: [0, 1, 2, 3]
Enqueued 4. Here is the Queue is: [0, 1, 2, 3, 4]
do some dequeues...
Dequeued 0. Here is the Queue is: [1, 2, 3, 4]
Dequeued 1. Here is the Queue is: [2, 3, 4]
Dequeued 2. Here is the Queue is: [3, 4]
Dequeued 3. Here is the Queue is: [4]
Dequeued 4. Here is the Queue is: []
Queue is empty!
```

Circular Array Implementation of a Queue

```
// Simple Queue using an Array Data Structure

    public class arrayQueue{
        private Object[]  queue;
        private int current_size,front,rear;
        private final int QUEUE_SIZE=7;

        public arrayQueue( ) {
           queue = new Object[QUEUE_SIZE];
           current_size=0;
           front = 0;
           rear = -1;
        }
        public boolean isEmpty()
        {
        if(current_size==0) return true;
            else return false;
        }
        public void enQueue(Object item) {
            if(current_size==QUEUE_SIZE) {
              System.out.println("Overflow..");
                System.exit(0);
           }
            rear++;
            rear=rear%QUEUE_SIZE;
            queue[rear] = item;
            current_size++;

        }
        public Object deQueue() {
            Object result;
            if (this.isEmpty()){
                System.out.println("Queue is empty!");
                return null;
            } else {
                    result=queue[front];
                    front=(++front)%QUEUE_SIZE;
                    current_size--;
                    return result;
            }
        }
        public void printQueue()  {
            System.out.print("the Queue is: ");
            if (this.isEmpty()){
              System.out.println(" empty!");
           }else {
                   for (int i=front;i!=rear; i=(++i)%QUEUE_SIZE)
                      System.out.print(queue[i] + ", ");
                   System.out.println(queue[rear]);
            }
        }
```

```
        public static void main( String [ ] args )
        {
           arrayQueue Q = new arrayQueue();
           Object x;  int i;
           Q.printQueue();  //should print empty queue message

           for( i = 0; i < 7; i++ ) {
                System.out.print("Enqueued " + i + " . ");
                Q.enQueue( new Integer( i ));
                Q.printQueue();
           }
           System.out.println( "do some dequeues..." );
           x=Q.deQueue();
           Q.printQueue();
           x=Q.deQueue();
           Q.printQueue();
           Q.enQueue(new Integer(23));
           Q.printQueue();
           Q.enQueue(new Integer(34));
           Q.printQueue();
           while(!Q.isEmpty()) {
                 x = Q.deQueue();
                 System.out.print( "Dequeued " +(Integer)x +". Here is ");
                 Q.printQueue();
            }
            Q.deQueue();  //should print error: dequeue from empty queue
        }
}
the Queue is:  empty!
Enqueued 0 . the Queue is: 0
Enqueued 1 . the Queue is: 0, 1
Enqueued 2 . the Queue is: 0, 1, 2
Enqueued 3 . the Queue is: 0, 1, 2, 3
Enqueued 4 . the Queue is: 0, 1, 2, 3, 4
Enqueued 5 . the Queue is: 0, 1, 2, 3, 4, 5
Enqueued 6 . the Queue is: 0, 1, 2, 3, 4, 5, 6
do some dequeues...
the Queue is: 1, 2, 3, 4, 5, 6
the Queue is: 2, 3, 4, 5, 6
the Queue is: 2, 3, 4, 5, 6, 23
the Queue is: 2, 3, 4, 5, 6, 23, 34
Dequeued 2. Here is the Queue is: 3, 4, 5, 6, 23, 34
Dequeued 3. Here is the Queue is: 4, 5, 6, 23, 34
Dequeued 4. Here is the Queue is: 5, 6, 23, 34
Dequeued 5. Here is the Queue is: 6, 23, 34
Dequeued 6. Here is the Queue is: 23, 34
Dequeued 23. Here is the Queue is: 34
Dequeued 34. Here is the Queue is:  empty!
Queue is empty!
```