CHAP 8 : Queue ADT

THE QUEUE ADT IS A LIST IN WHICH ALL INSERTIONS ARE DONE AT ONE END (THE 'BACK'), AND ALL DELETIONS TAKE PLACE AT THE OTHER END (THE 'FRONT'). THUS THE FIRST ELEMENT PLACED IN THE QUEUE WILL BE THE FIRST ELEMENT REMOVED. WE CALL THIS PROPERTY FIRST IN, FIRST OUT OR FIFO.

AS USUAL WE WILL SIMPLIFY THE BOOK'S EXPOSITION TO CONSIDER AN IntegerQueue.

OPERATIONS

```
isEmpty()
// RETURNS TRUE IFF Queue is EMPTY
// Pre: none.  // Post: ↑

enqueue(int item)
// Pre: none
// Post: adds item TO BACK OF Queue

dequeue()
// Pre: !isEmpty()
// Post: RETURNS & DELETES item AT FRONT OF Queue

dequeueAll()
// Pre: !isEmpty()
// Post: SETS Queue TO EMPTY STATE
```

```
Peek()
// Pre: !isEmpty
// Post: returns item at front of queue
```

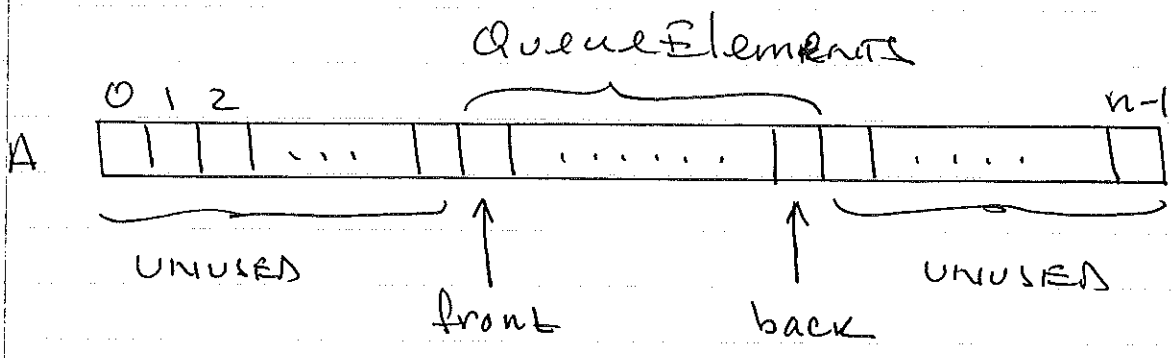
NOTE: Book does not include !isEmpty() as precond. to dequeue(), dequeueAll(), and peek().

Again we consider three implementations

- Array based with doubling
- Linked list implementation
- Based on IntegerList ADT.

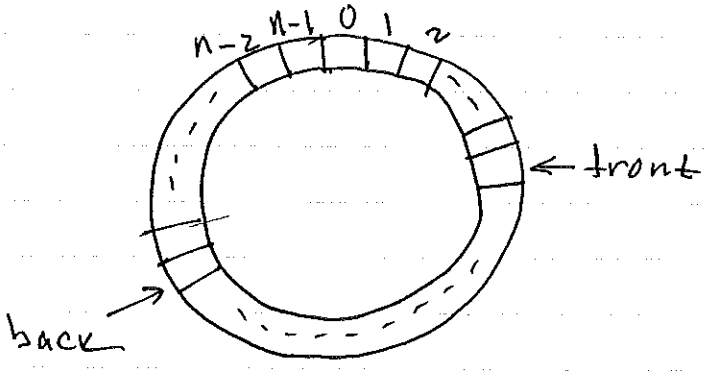
The Linked List and IntegerList ADT based implementations are quite easy and left as an exercise. We discuss the array based implementation (with array doubling.)

ARRAY BASED QUEUE



dequeue() JUST INCREMENTS front AND enqueue() INCREMENTS back THEN INSERTS NEW ITEM AT INDEX back. AFTER ANY NUMBER OF QUEUE OPERATIONS THE AREA IN A STORING QueueElements TENDS TO DRIFT TO THE RIGHT, EVENTUALLY REACHING THE END. WE COULD SOLVE THIS BY ARRAY DOUBLING, BUT WE COULD END UP WITH A VERY SMALL QUEUE STORED IN A VERY LARGE ARRAY.

TO DEAL WITH RIGHTWARD DRIFT WE TREAT THE ARRAY AS BEING CIRCULAR.



TO ACCOMPLISH THIS ALL INDEX ARITHMETIC WILL BE PERFORMED MODULO n, WHERE n IS THE PHYSICAL SIZE OF THE UNDERLYING ARRAY.

RECALL THAT THE ARITHMETIC OPERATION $a \% n$ GIVES THE REMAINDER OF a UPON DIVISION BY n. THE POSSIBLE REMAINDERS UPON DIVISION BY n ARE $\{0, 1, \ldots, n-1\}$, WHICH ARE PRECISELY THE VALID ARRAY INDICES TO AN ARRAY OF LENGTH n.

THE BASIC OPERATIONS enqueue() AND dequeue() BECOME

enqueue():
- back = (back + 1) % n
- A[back] = newItem

dequeue():
- front = (front + 1) % n

WE DOUBLE THE ARRAY ONLY WHEN ALL ITS SLOTS ARE FILLED WITH QUEUE ELEMENTS AND WE WISH TO INSERT A NEW ELEMENT.

SINCE WE HAVE TO COPY ELEMENTS INTO THE NEW ARRAY ANYWAY, WE TAKE THIS OPPORTUNITY TO ELIMINATE RIGHTWARD DRIFT IN THE OLD ARRAY BY PLACING

ITEMS IN POSITIONS 0-(n-1) OF NEW ARRAY (REGARDLESS OF THEIR POSSITIONS IN THE OLD ARRAY.) THEN WE PLACE THE NEW ITEM IN POSITION n.

HOW DO WE KNOW WHEN THE ARRAY IS FULL? WE KEEP TRACK OF THE NUMBER OF ITEMS IN THE QUEUE AND COMPARE THAT TO THE PHYSICAL SIZE OF THE ARRAY. WE CANNOT CHECK A CONDITION LIKE back == front-1 SINCE THAT COULD INDICATE EITHER A FULL ARRAY OR AN EMPTY ARRAY.

- SEE ARRAY BASED IMPLEMENTATION -

- READ APPLICATION: SIMULATION P. 317-326.

```
// IntegerQueueInterface.java
// interface for the IntegerQueue ADT

public interface IntegerQueueInterface{

   // isEmpty()
   // pre: none
   // post: returns true if this IntgerQueue is empty, false otherwise
   public boolean isEmpty();

   // enqueue()
   // adds x to back of this IntegerQueue
   // pre: none
   // post: !isEmpty()
   public void enqueue(int x);

   // dequeue()
   // deletes and returns item from front of this IntegerQueue
   // pre: !isEmpty()
   // post: this IntegerQueue will have one fewer element
   public int dequeue() throws QueueEmptyException;

   // peek()
   // pre: !isEmpty()
   // post: returns item at front of Queue
   public int peek() throws QueueEmptyException;

   // dequeueAll()
   // sets this IntegerQueue to the empty state
   // pre: !isEmpty()
   // post: isEmpty()
   public void dequeueAll() throws QueueEmptyException;
}
```

```
// IntegerQueue.java
// Array based implementation of IntegerQueue ADT (with array doubling)

public class IntegerQueue implements IntegerQueueInterface{

   private static final int INITIAL_SIZE = 1;
   private int physicalSize;  // current length of underlying array
   private int[] item;        // array of IntegerQueue items
   private int numItems;      // number of items in this IntegerQueue
   private int front;         // index of front element
   private int back;          // index of back element

   // doubleItemArray()
   // doubles the physical size of the underlying array item[]
   private void doubleItemArray(){
      int[] newArray = new int[2*physicalSize];
      for(int i=0; i<numItems; i++) newArray[i] =
      item[(front+i)%physicalSize];
      item = newArray;
      physicalSize *=2;
      front = 0;
      back = numItems-1;
   }

   // IntegerQueue()
   // default constructor for the IntegerQueue class
   public IntegerQueue(){
      physicalSize = INITIAL_SIZE;
      item = new int[physicalSize];
      numItems = 0;
      front = 0;
      back = physicalSize-1;
   }

   // isEmpty()
   // pre: none
   // post: returns true if this IntgerQueue is empty, false otherwise
   public boolean isEmpty(){
      return(numItems == 0);
   }

   // enqueue()
   // adds x to back of this IntegerQueue
   // pre: none
   // post: !isEmpty()
   public void enqueue(int x){
      if( numItems == physicalSize ) doubleItemArray();
      numItems++;
      back = (back+1)%physicalSize;
      item[back] = x;
   }

   // dequeue()
   // deletes and returns item from front of this IntegerQueue
   // pre: !isEmpty()
   // post: this IntegerQueue will have one fewer element
   public int dequeue() throws QueueEmptyException{
      if( numItems==0 ){
         throw new QueueEmptyException("cannot dequeue() empty queue");
      }
      int returnValue = item[front];
      front = (front+1)%physicalSize;
      numItems--;
      return returnValue;
   }
```

```
    // peek()
    // pre: !isEmpty()
    // post: returns item at front of Queue
    public int peek() throws QueueEmptyException{
       if( numItems==0 ){
          throw new QueueEmptyException("cannot peek() empty queue");
       }
       return item[front];
    }

    // dequeueAll()
    // sets this IntegerQueue to the empty state
    // pre: !isEmpty()
    // post: isEmpty()
    public void dequeueAll() throws QueueEmptyException{
       if( numItems==0 ){
          throw new QueueEmptyException("cannot dequeueAll() empty queue");
       }
       numItems = 0;
       front = 0;
       back = physicalSize-1;
    }

    // toString()
    // overrides Object's toString() method
    public String toString(){
       String s = "";
       for(int i=0; i<numItems; i++){
          s += item[(front+i)%physicalSize] + " ";
       }
       return s;
    }

    // equals()
    // overrides Object's equals() method
    public boolean equals(Object rhs){
       IntegerQueue R = null;
       boolean eq = false;
       int i, n, m;

       if(rhs instanceof IntegerQueue){
          R = (IntegerQueue)rhs;
          eq = ( this.numItems == R.numItems );
          for(i=0; eq && i<numItems; i++){
             n = (this.front + i)%this.physicalSize;
             m = (R.front + i)%R.physicalSize;
             eq = ( this.item[n] == R.item[m] );
          }
       }
       return eq;
    }
}
```

```
// IntegerQueueTest.java
// Test Client for the IntegerQueue class

public class IntegerQueueTest {

   public static void main(String[] args){
      IntegerQueue A = new IntegerQueue();
      A.enqueue(5); A.enqueue(3); A.enqueue(9); A.enqueue(7); A.enqueue(8);
      System.out.println(A);
      System.out.println(A.peek());
      A.dequeue(); A.dequeue(); A.dequeue();
      System.out.println(A.peek());
      System.out.println(A);
      IntegerQueue B = new IntegerQueue();
      System.out.println(A.isEmpty());
      System.out.println(B.isEmpty());
      B.enqueue(7); B.enqueue(8);
      System.out.println(A.equals(B));
      A.enqueue(12);
      B.enqueue(13);
      System.out.println(A);
      System.out.println(B);
      System.out.println(A.equals(B));
      A.dequeueAll();
      System.out.println(A);
      System.out.println(A.isEmpty());
   }
}
```

```
// QueueEmptyException.java

public class QueueEmptyException extends RuntimeException{
   public QueueEmptyException(String s){
      super(s);
   }
}
```