9.1 QUEUE

Queue is a First-In-First-Out (FIFO) data structure, in which the first element added to the queue will be the first one to be removed.

Figure 9.1: Queue

Listing 1: Queue Interface

```java
public interface Queue<T> extends Iterable<T> {
    void enqueue(T t);
    T dequeue(); // throws EmptyStackException;
    T peek(); // throws EmptyStackException;
    boolean isEmpty();
    int size();
}
```

If we have a Queue Q:

```java
Q.enqueue(10);
10
```

If we add numbers 20,30

```java
Q.enqueue(20);
Q.enqueue(30);
[10]
[20]
[30]
```

If we pop a number:

```java
```
9.1.1 Array based implementation of Queue

In this section, we use a resizable array to implement queue. Array “Items” holds the elements of the stack. items[first] and items[last] always point to the head and tail of the queue. When an item is pushed, the item is stored in items[last]. To remove an item, items[first] will be removed. To avoid moving array elements when the items[first] is removed, a circular array is used. When “first==array.length”, then “first = first%array.length”, so that it wraps around. Same rule applies to “last”.

Listing 2: Array Based Queue

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Based on the implementation by author Robert Sedgewick and Kevin Wayne
 */
public class ArrayQueue<E> implements Queue<E> {
    private E[] q; // queue elements
    private int N = 0; // number of elements on queue
    private int first = 0; // index of first element of queue
    private int last = 0; // index of next available slot

    /**
     * Initializes an empty queue.
     */
    public ArrayQueue() {
        // cast needed since no generic array creation in Java
        q = (E[]) new Object[2];
    }

    /**
     * Is this queue empty?
     * @return true if this queue is empty; false otherwise
     */
    public boolean isEmpty() {
        return N == 0;
    }

    /**
     * Returns the number of items in this queue.
     * @return the number of items in this queue
     */
    public int size() {
        return N;
    }

    // resize the underlying array
```
private void resize(int max) {
    assert max >= N;
    E[] temp = (E[]) new Object[max];
    for (int i = 0; i < N; i++) {
        temp[i] = q[(first + i) % q.length];
    }
    q = temp;
    first = 0;
    last = N;
}
/**
 * Adds the item to this queue.
 * @param item the item to add
 */
public void push(E item) {
    if (N == q.length) resize(2*q.length);
    q[last++] = item;
    if (last == q.length) last = 0;
    N++;
}
/**
 * Removes and returns the item on this queue that was least recently added.
 * @return the item on this queue that was least recently added
 * @throws java.util.NoSuchElementException if this queue is empty
 */
public E pop() {
    if (isEmpty()) throw new NoSuchElementException("Queue underflow");
    E item = q[first];
    q[first] = null;
    N--;
    first++;
    if (first == q.length) first = 0;
    if (N > 0 && N == q.length/4) resize(q.length/2);
    return item;
}
/**
 * Returns the item least recently added to this queue.
 * @return the item least recently added to this queue
 * @throws java.util.NoSuchElementException if this queue is empty
 */
public E peek() {
    if (isEmpty()) throw new NoSuchElementException("Queue underflow");
    return q[first];
}
/**
 * Returns an iterator that iterates over the items in this queue in FIFO order.
 * @return an iterator that iterates over the items in this queue in FIFO order
 */
public Iterator<E> iterator() {
    return new ArrayIterator();
}
private class ArrayIterator implements Iterator<E> {
    private int i = 0;
    public boolean hasNext() { return i < N; }
    public void remove() { throw new UnsupportedOperationException(); }
    public E next() {
        if (!hasNext()) throw new NoSuchElementException();
        E item = q[(i + first) % q.length];
        i++;
        return item;
    }
}
Lecture 9:

9.1.2 Linked List based implementation of Queue

In this section, we implement the queue using singly linked list. We always add and remove the nodes at the head of the linked list.

Listing 3: Linked List Based Queue

```java
import java.util.Iterator;
import java.util.NoSuchElementException;
/**
 * Based on the implementation of Robert Sedgewick and Kevin Wayne
 */
public class LinkedQueue<E> implements Queue<E> {
    private int N; // number of elements on queue
    private Node first; // beginning of queue
    private Node last; // end of queue

    // helper linked list class
    private class Node {
        private E item;
        private Node next;
    }

    /**
     * Initializes an empty queue.
     */
    public LinkedQueue() {
        first = null;
        last = null;
        N = 0;
    }

    /**
     * Is this queue empty?
     * @return true if this queue is empty; false otherwise
     */
    public boolean isEmpty() {
        return first == null;
    }

    /**
     * Returns the number of items in this queue.
     * @return the number of items in this queue
     */
    public int size() {
        return N;
    }
}
```
Lecture 9: 9-5

```java
public int size() {
    return N;
}

/**
 * Returns the item least recently added to this queue.
 * @return the item least recently added to this queue
 * @throws java.util.NoSuchElementException if this queue is empty
 */
public E peek() {
    if (isEmpty()) throw new NoSuchElementException("Queue underflow");
    return first.item;
}

/**
 * Adds the item to this queue.
 * @param item the item to add
 */
public void enqueue(E item) {
    Node oldlast = last;
    last = new Node();
    last.item = item;
    last.next = null;
    if (isEmpty()) first = last;
    else oldlast.next = last;
    N++;
}

/**
 * Removes and returns the item on this queue that was least recently added.
 * @return the item on this queue that was least recently added
 * @throws java.util.NoSuchElementException if this queue is empty
 */
public E dequeue() {
    if (isEmpty()) throw new NoSuchElementException("Queue underflow");
    E item = first.item;
    first = first.next;
    N--;
    if (isEmpty()) last = null; // to avoid loitering
    return item;
}

/**
 * Returns a string representation of this queue.
 * @return the sequence of items in FIFO order, separated by spaces
 */
public String toString() {
    StringBuilder s = new StringBuilder();
    for (E item : this)
        s.append(item + " ");
    return s.toString();
}

/**
 * Returns an iterator that iterates over the items in this queue in FIFO order.
 * @return an iterator that iterates over the items in this queue in FIFO order
 */
public Iterator<E> iterator() {
    return new ListIterator();
}

// an iterator, doesn’t implement remove() since it’s optional
private class ListIterator implements Iterator<E> {
    private Node current = first;

    public boolean hasNext() { return current != null; }
    public void remove() { throw new UnsupportedOperation(); }
}
```
```java
public E next()
{
    if (!hasNext()) throw new NoSuchElementException();
    E item = current.item;
    current = current.next;
    return item;
}

/**
 * Unit tests the LinkedList data type.
 */
public static void main(String[] args)
{
    Queue<Integer> q = new LinkedList();
    for(int i = 1; i <= 5; i++)
    {
        q.enqueue(i);
    }
    while(!q.isEmpty())
    {
        System.out.print(q.peek()+",”);
        System.out.print(q.dequeue()+",”);
    }
    System.out.println("\n");
}
```