15.6 – Queues

- A queue is a linear collection whose elements are added on one end and removed from the other.
- Queue elements are processed in a first in, first out (FIFO) manner.
- Elements are removed from the queue in the same order in which they are placed on the queue.
- A queue is consistent with the general concept of:
  - A waiting line to buy movie tickets
  - A request to print a document
  - Crawling the web to retrieve documents

---

15.6 – javafoundations.Queue

```java
package javafoundations;

public interface Queue<T> {
    // Adds the specified element to the rear of the queue.
    public void enqueue(T element);
    // Removes and returns the element at the front of the queue.
    public T dequeue();
    // Returns a reference to the element at the front of the queue
    // without removing it.
    public T first();
    // Returns true if the queue contains no elements and false otherwise.
    public boolean isEmpty();
    // Returns the number of elements in the queue.
    public int size();
    // Returns a string representation of the queue.
    public String toString();
}
```

---

15.7 – Radix Sort

- A radix sort is unusual because it does not involve comparisons between elements!
- The technique used in the radix sort is based on the structure of the sort key.
- Separate queues are created for each possible value of each digit or character of the sort key.
- The number of queues, or the number of possible values, is called the radix.
  - If we were sorting strings made up of lowercase alphabetic characters, the radix would be 26, one for each possible character.
  - If we were sorting decimal numbers, then the radix would be 10, one for each digit 0 to 9.
- The radix sort makes a pass through the values for each position in the sort key.
15.7 – Radix Sort (1st pass)

Digit | 1s position
--- | ---
0 | front of queue
1 |
2 |
3 |
4 |
5 |

442 | 503 | 312 | 145 | 250 | 341 | 325 | 102 | 420 | 143

Original list

15.7 – Radix Sort (2nd pass results)

Digit | 10s position
--- | ---
0 | 503 | 102
1 | 312 |
2 | 325 | 420
3 |
4 | 145 | 143 | 442 | 341
5 | 250 |

15.6 – RadixSort.java

```java
for (int digitVal = 0; digitVal <= 9; digitVal++)
    digitQueues[digitVal] = new ArrayQueue<Integer>();
// sort the list
for (int position=0; position <= 3; position++)
    {for (int scan = 0; scan < list.length; scan++)
    {temp = String.valueOf (list[scan]);
     digit = Character.digit (temp.charAt(3-position), 10);
     digitQueues[digit].enqueue (list[scan]);
    }
// gather numbers back into list
num = 0;
for (int digitVal = 0; digitVal <= 9; digitVal++)
    {while (!(digitQueues[digitVal].isEmpty()))
    {list[num] = digitQueues[digitVal].dequeue().intValue();
     num++;
    }
    }
(more...)
```
15.8 – Implementing Queues with Arrays

Adds a new element to the rear of the queue, which is stored at the high end of the array.

```
public void enqueue (T element) { ...
```

Left-shift to correct queue

1. Adds a new element to the rear of the queue, which is stored at the high end of the array.
2. As elements are dequeued, the front of the queue will move further into the array.
3. As elements are enqueued, the rear of the queue will also move further into the array.
4. The challenge comes when the rear of the queue reaches the end of the array.
5. When this occurs, it "wraps around" to the front of the array.
6. Use two variables, `front` and `rear`, to represent the location where the first element is stored, and where the next available slot in the array is located (respectively).
15.9 – Implementing Queues with Circular Arrays

A circular array queue is implemented by storing the front and rear of the queue in variables, and allowing these variables to wrap around the array as elements are added and removed. This allows the queue to efficiently grow and shrink in size.

15.9 – javafoundations.CircularArrayQueue

```java
package javafoundations;
import javafoundations.exceptions.*;
public class CircularArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int front, rear, count;
    private T[] queue;

    // Creates an empty queue using the default capacity.
    public CircularArrayQueue() {
        front = rear = count = 0;
        queue = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    // Adds the specified element to the rear of this queue, expanding
    // the capacity of the queue array if necessary.
    public void enqueue (T element) {
        if (count == queue.length)
            expandCapacity();
        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }

    // Creates a new array to store the contents of this queue with
    // twice the capacity of the old one.
    public void expandCapacity() {
        T[] larger = (T[]) (new Object[queue.length*2]);
        for (int index=0; index < count; index++)
            larger[index] = queue[(front+index) % queue.length];
        front = 0;
        rear = count;
        queue = larger;
    }

    // The following methods are left as Programming Projects.
    public T dequeue () throws EmptyCollectionException {
    }
    public T first () throws EmptyCollectionException {
    }
    public boolean isEmpty() {
    }
    public String toString() {
    }
}
```
15.10 – Implementing Queues with Links

---

```java
// LinkedQueue.java       Java Foundations
// Represents a linked implementation of a queue.
package javafoundations;
import javafoundations.exceptions.*;
public class LinkedQueue<T> implements Queue<T>
{
    private int count;
    private LinearNode<T> front, rear;

    //-----------------------------------------------------------------
    // Creates an empty queue.
    //-----------------------------------------------------------------
    public LinkedQueue()
    {
        count = 0;
        front = rear = null;
    }

    public void enqueue(T element)
    {
        LinearNode<T> node = new LinearNode<T>(element);
        if (count == 0)
            front = node;
        else
            rear.setNext(node);
        rear = node;
        count++;
    }

    // The following methods are left as Programming Projects.
    // public T dequeue () throws EmptyCollectionException { }
    // public T first () throws EmptyCollectionException { }
    // public boolean isEmpty() { }
    // public int size(){ }
    // public String toString() { }
}
```

---

Analysis of Stack and Queue Implementations

- Both stacks and queues can be implemented very efficiently.
- In almost all cases, the operations are not affected by the number of elements in the collection.
- All operations for a stack (push, pop, peek, etc.) are O(1).
- Almost all operations for a queue are O(1).
- The only exception is the dequeue operation for the ArrayQueue implementation – the shifting of elements makes it O(n).
- The dequeue operation for the CircularArrayQueue is O(1) because of the ability to eliminate the shifting of elements.