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

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
15.6 – Queues

Adding an element

Removing an element

rear of queue

front of queue

javafoundations.Queue

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

public String toString();
Example using a Queue

```java
rites = new Queue<String>;
rites.enqueue ("RNaSG");
rites.enqueue ("G12HoSi1N");
rites.enqueue ("GSs");
rites.enqueue ("StCftoGST");
if (rites.first.equals("RNaSG"))
    rites.enqueue(rites.dequeue());
System.out.println("Today:"+ rites.dequeue());
System.out.println("Sunday:"+ rites.dequeue());
```

Radix Sort

- A **radix sort** is unusual because it does **not** involve **comparisons** between keys!
- 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.
- **Radix** = The number of queues, or the number of possible values
  - 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.
### Radix Sort (1st pass)

<table>
<thead>
<tr>
<th>Digit</th>
<th>1s position</th>
<th>front of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Original list:

442 503 312 145 250 341 325 102 420 143

### Radix Sort (2nd pass begins)

<table>
<thead>
<tr>
<th>Digit</th>
<th>1s position</th>
<th>front of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Original list:

8
Radix Sort (2nd pass results)

<table>
<thead>
<tr>
<th>Digit</th>
<th>10s position</th>
<th>front of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>503 102</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>312</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>325 420</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>145 143 442 341</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

Visualization: https://visualgo.net/en/sorting

Radix Sort (3rd pass results)

<table>
<thead>
<tr>
<th>Digit</th>
<th>100s position</th>
<th>front of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>145 143 102</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>341 325 312</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>442 420</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>503</td>
</tr>
</tbody>
</table>

Visualization: https://visualgo.net/en/sorting
RadixSort.java

```java
int[] list = {...} // the input array
for (int digitVal = 0; digitVal <= 9; digitVal++) { // create 10 Qs
digitQueues[digitVal] = new ArrayQueue<Integer>();

// sort the list that contains the numbers
for (int position=0; position <= 3; position++) { // 4-digit nums
    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 from the queues back into list
    num = 0;
    for (int digitVal = 0; digitVal <= 9; digitVal++) {
        while (!digitQueues[digitVal].isEmpty()) {
            list[num] = digitQueues[digitVal].dequeue().intValue();
            num++;
        }
    }
}
```

Implementing Queues with Arrays

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

```
0 1 2 3 4 5 6 7 ... ...
```

- Left-shift to correct queue
- Enqueue 'E'

```
A B C D E
```

- `count = 4`
package javafoundations;
import javafoundations.exceptions.*;

public class ArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int count;
    private T[] queue;

    //0-args constructor: Creates empty queue
    public ArrayQueue(){
        count = 0;
        queue = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public boolean isEmpty() {
    }

    public int size() {
    }

    public T first() throws EmptyCollectionException {
    }

    public void enqueue(T element) {
    }
}
public T dequeue() throws EmptyCollectionException {
    if (count == 0) throw new EmptyCollectionException
        ("Dequeue failed. Queue empty");

}

//Left as programming project:
//public String toString() {}
15.9 – Implementing Queues with Circular Arrays

15.9 – The Changing State of a Circular Array Q
javafoundations.CircularArrayQueue

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;

    public CircularArrayQueue() {
    }

    public void enqueue(T element) {
        if (count == queue.length)
            expandCapacity();
        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }

    (more...)

javafoundations.CircularArrayQueue

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;

    public CircularArrayQueue() {
    }

    public void enqueue(T element) {
        if (count == queue.length) expandCapacity();
        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }

    (more...)

javafoundations.CircularArrayQueue

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;

    public CircularArrayQueue() {
    }

    public void enqueue(T element) {
        if (count == queue.length) expandCapacity();
        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }

    (more...)
javafoundations.CircularArrayQueue

// 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 int size() { }
// public boolean isEmpty() { }
// public String toString() { }

Distributed Denial of Service (DDOS)

- To connect two computers on the internet we use routers: simple devices that control a queue
  - They’re CircularArrayQueues without expandCapacity()

- A router connects to several channels, chooses the best to send a message it received

- When (portions of) a message arrives, the router stores it temporarily, decides which channel to use next, dequeues and sends the message.

- A DDOS is caused when attackers flood routers with messages for delivery!
15.10 – Implementing Queues with Links

```java
package javafoundations;

import javafoundations.exceptions.*;

public class LinkedQueue<T> implements Queue<T> {
    private int count;
    private LinearNode<T> front, rear;

    // Represents a linked implementation of a queue.
    public LinkedQueue() {
        count = 0;
        front = rear = null;
    }

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

    // create a non-empty queue.
    public LinkedQueue(T element) {
        count = 1;
        front = new LinearNode<T>(element);
        rear = front;
    }

    // create a queue with multiple elements.
    public LinkedQueue(T[] elements) {
        for (T element : elements) {
            addLast(element);
        }
    }

    // add an element to the front of the queue.
    public void addFirst(T element) {
        LinearNode<T> newNode = new LinearNode<T>(element);
        if (front == null) {
            front = newNode;
        } else {
            newNode.next = front;
            front = newNode;
        }
        count++;
    }

    // add an element to the end of the queue.
    public void addLast(T element) {
        LinearNode<T> newNode = new LinearNode<T>(element);
        if (rear == null) {
            rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        count++;
    }

    // remove the front of the queue.
    public T removeFirst() {
        if (front == null) {
            return null;
        }
        T element = front.element;
        if (front == rear) {
            front = rear = null;
        } else {
            front = front.next;
        }
        count--;
        return element;
    }

    // remove the last of the queue.
    public T removeLast() {
        if (rear == null) {
            return null;
        }
        T element = rear.element;
        if (front == rear) {
            front = rear = null;
        } else {
            rear = rear.prev;
            rear.next = null;
        }
        count--;
        return element;
    }

    // get the front of the queue.
    public T getFirst() {
        if (front == null) {
            return null;
        }
        return front.element;
    }

    // get the last of the queue.
    public T getLast() {
        if (rear == null) {
            return null;
        }
        return rear.element;
    }

    // get the size of the queue.
    public int size() {
        return count;
    }

    // check if the queue is empty.
    public boolean isEmpty() {
        return count == 0;
    }

    // check if the queue contains an element.
    public boolean contains(T element) {
        LinearNode<T> current = front;
        while (current != null) {
            if (current.element.equals(element)) {
                return true;
            }
            current = current.next;
        }
        return false;
    }

    // print the elements of the queue.
    public void print() {
        LinearNode<T> current = front;
        while (current != null) {
            System.out.print(current.element + " ");
            current = current.next;
        }
        System.out.println();
    }
}
```
javafoundations.LinkedQueue

// Add the specified element to the rear of this queue.
public void enqueue (T element)
{
    LinearNode<T> node = new LinearNode<T>(element);
    if (count == 0)
        front = node;
    else
        rear.setNext(node);
    rear = node;
    count++;
}

Analysis of Stack and Queue Implementations

* 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
* Both stacks and queues can be implemented very efficiently