Queues

Chapter 7

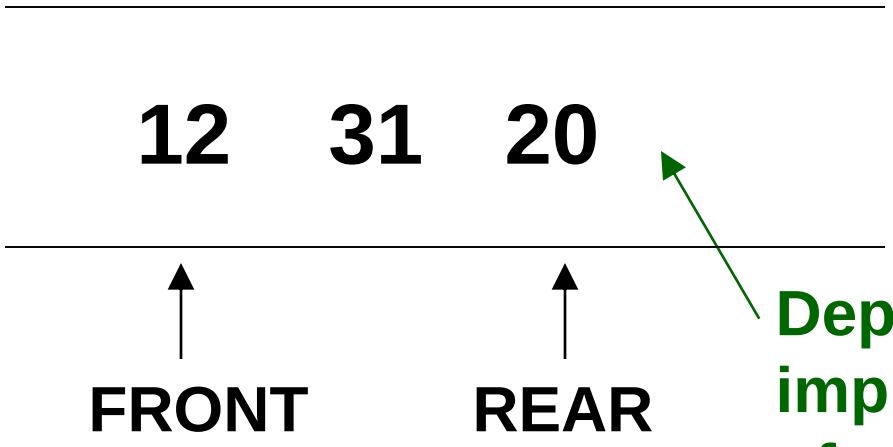
Fundamentals

- A queue is a sequence of data elements (of the same type) arranged one after another conceptually.
- An element can be added to the rear of the queue only. (“ENQUEUE”)
- An element can be removed from the front of the queue only. (“DEQUEUE”)

Applications of Queues

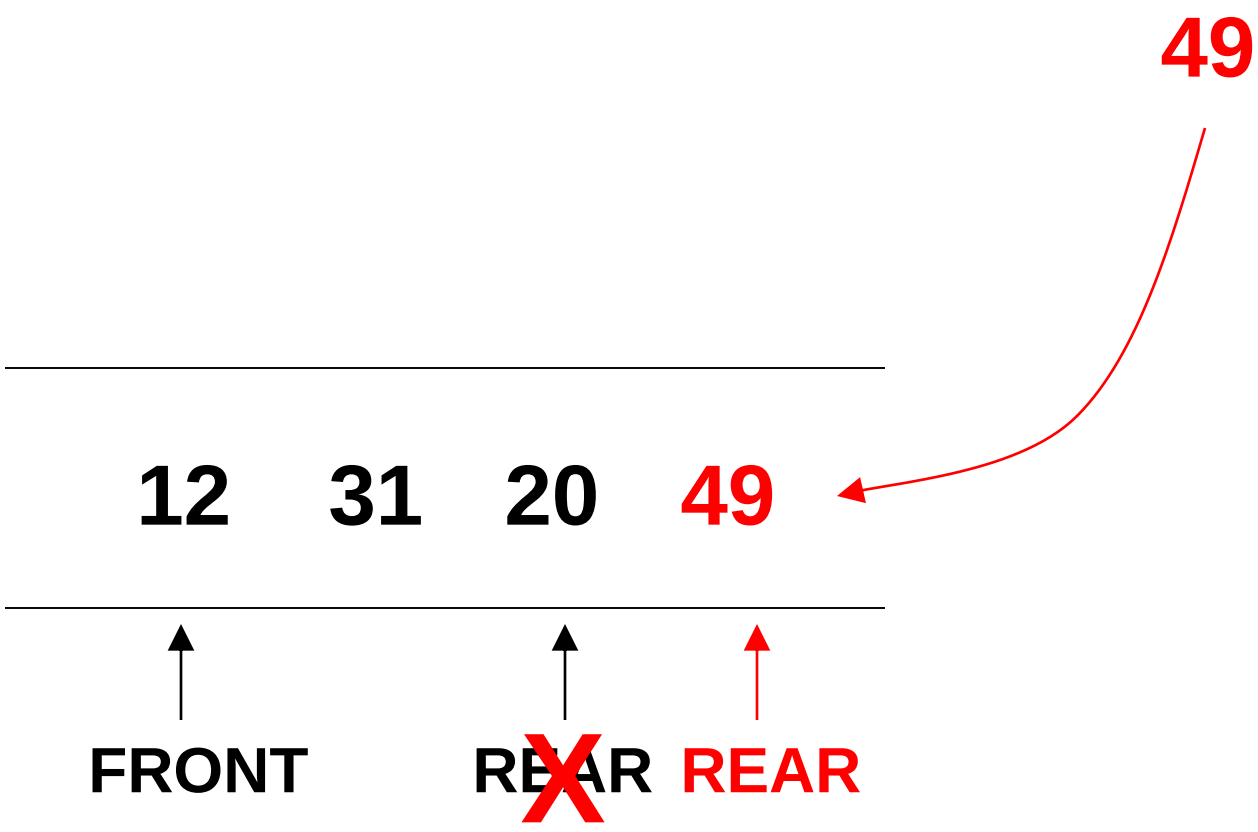
- Operating systems
  - input buffers, print queues, process scheduling
- Telecommunications and transportation engineering
  - modeling and simulation of computer and phone networks, road systems and transportation hubs

Conceptual Picture

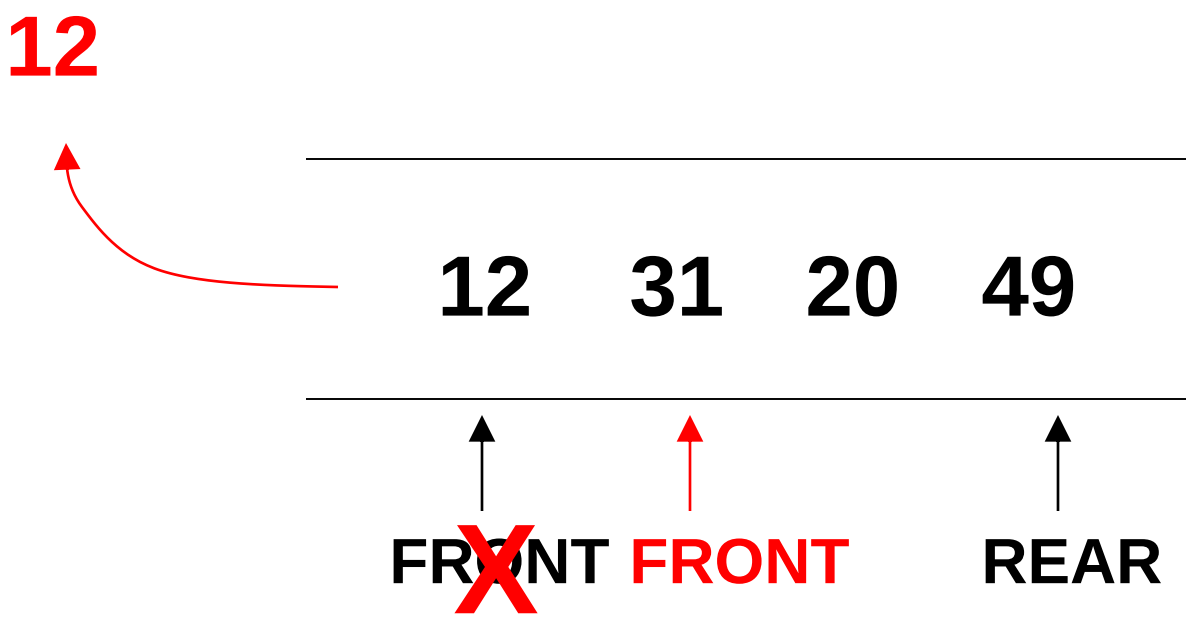


Depending on the implementation of a queue, it may or may not have a maximum capacity.

ENQUEUE



DEQUEUE

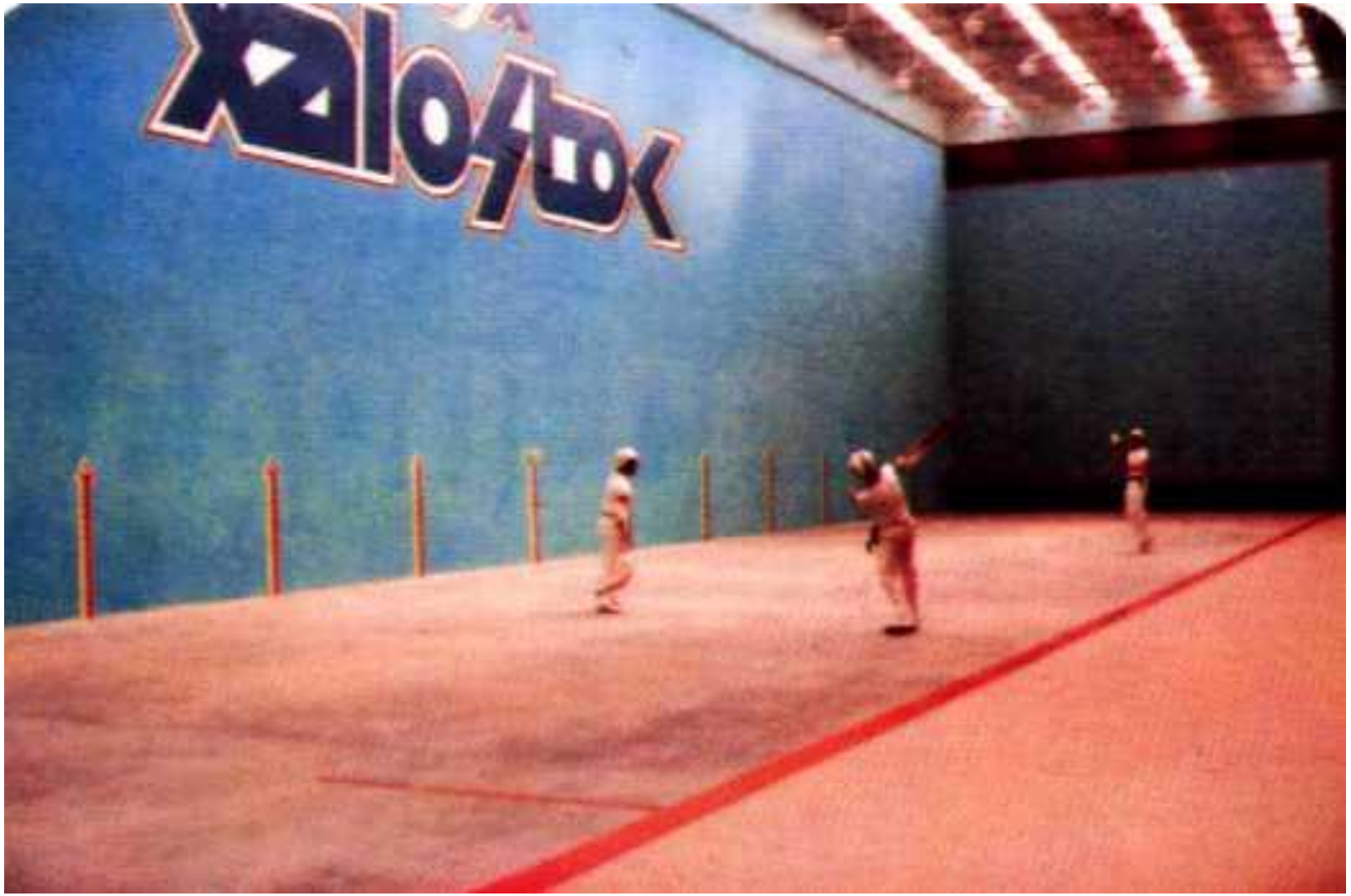


Depending on the implementation of the queue, the 12 may still be in memory but is not part of the queue.

Basic Queue Operations

- **Constructor – create an empty queue**
- **isEmpty – is the queue empty?**
- **enqueue – insert an element on to the rear of the queue (if the queue is not full)**
- **dequeue – remove the front element from the queue (if the queue is not empty)**

Example: Jai Alai



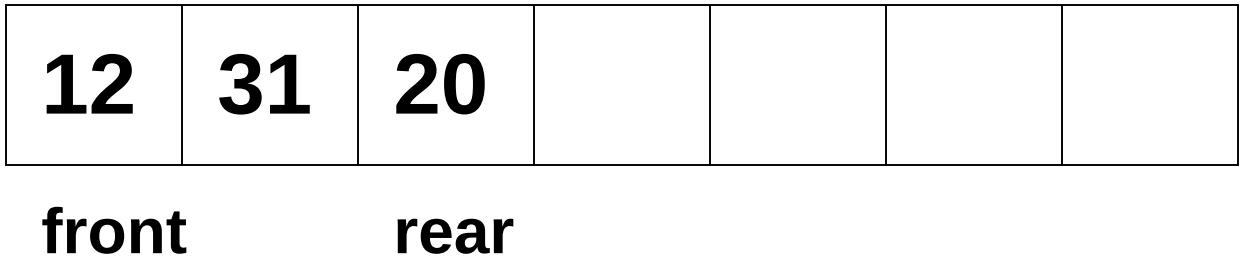
Jai Alai Rules

- **Played by 8 teams.**
- **Two teams play against each other while the others wait in a queue.**
- **After each match of two teams, the loser goes to the end of the queue, and the first team in the queue leaves and plays the previous winner.**
- **During the first seven matches, the winner of each match earns one point.**
- **After the first seven matches, the winner of each match earns two points.**
- **The game ends when one team earns 7 points.**

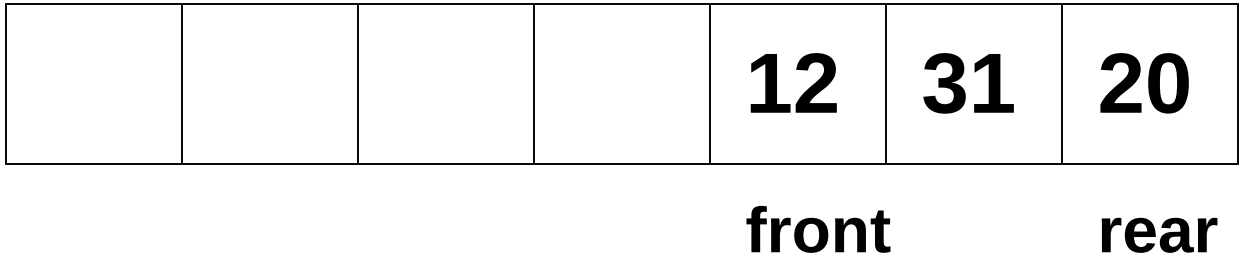
Implementation of a Queue

- **ARRAYS**

front element is always in first array cell



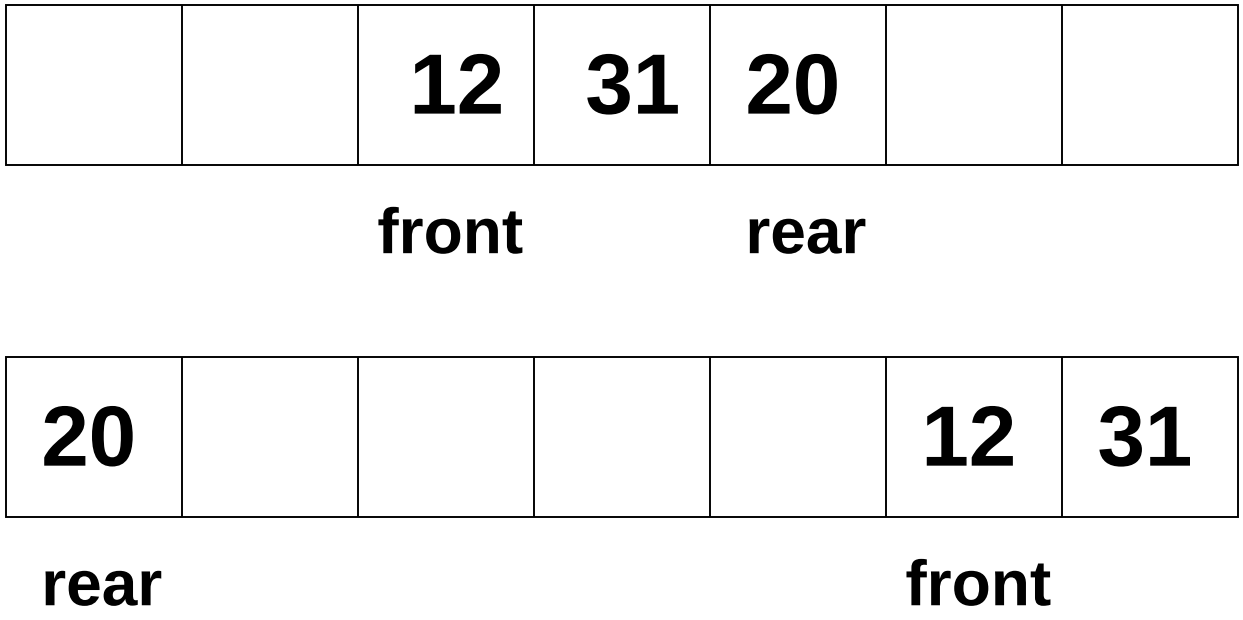
rear element is always in last array cell



Is there a better way?

Implementation of a Queue

- **ARRAYS (a better way)**



An IntQueue using Arrays

```
public class IntQueue implements
  Cloneable {

  public final int CAPACITY = 100;
  private int[] data;
  private int front;
  private int rear;

  // IntQueue methods (clone not shown)
}
```

IntQueue (arrays) (cont'd)

```
public IntQueue()
{
    front = -1;
    rear = -1;
    data = new int[CAPACITY];
}
public boolean isEmpty()
{
    return (front == -1);
}
```

IntQueue (arrays) (cont'd)

```
public void enqueue(int item)
{
     if ((rear+1)%CAPACITY == front)
        throw new FullQueueException();
     if (front == -1) {  // isEmpty()
         front = 0; rear = 0;
     }
     else rear = (rear+1)%CAPACITY;
     data[rear] = item;
}
```

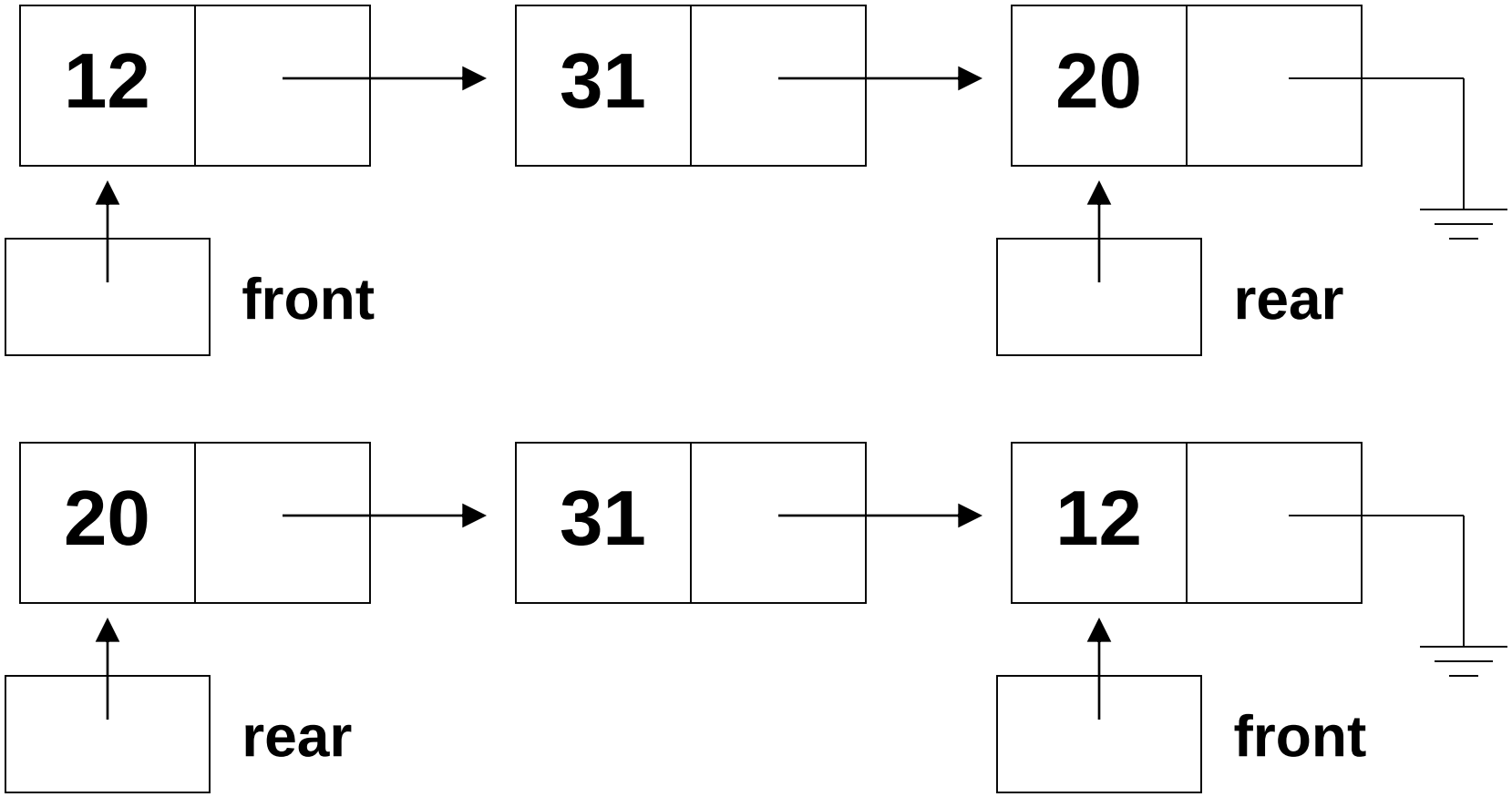
IntQueue (arrays) (cont'd)

```
public int dequeue()
{
    int answer;
    if (front == -1) // isEmpty()
       throw new EmptyQueueException();
    answer = data[front];
    if (front == rear) {
        front = -1; rear = -1;
    }
    else front = (front+1)%CAPACITY;
    return answer;
}
```

Implementation of a Queue

- **LINKED LISTS**

Which is more efficient?



An IntQueue using Lists

```
public class IntQueue implements
  Cloneable {

  private IntNode front;
  private IntNode rear;

  // IntQueue methods (clone not shown)
}
```

IntQueue (lists) (cont'd)

```
public IntQueue()
{
    front = null;
    rear = null;
}
public boolean isEmpty()
{
    return (front == null);
}
```

IntQueue (lists) (cont'd)

```
public void enqueue(int item)
{
     IntNode newNode = new IntNode(item);
     if (front == null) {
          front = newNode; rear = front;
     }
     else {
          rear.setLink(newNode);
          rear = newNode;
     }
}
```

IntQueue (lists) (cont'd)

```
public int dequeue()
{
    int answer;
    if (front == null)
       throw new EmptyQueueException();
    answer = front.getData();
    front = front.getLink();
    if (front == null)  rear = null;
    return answer;
}
```

Priority Queue

- **Applications**
- **Implementations**

		front						***rear***		
		27	**23**	**10**	**19**	**36**	**29**	**72**	**48**	
		2	**1**	**3**	**1**	**2**	**3**	**4**	**2**	

Random Numbers

- **Math.random() returns a random uniformly-distributed double in the range [0.0,1.0) .**
- **Generating a random double in [0.0,10.0):**

  Math.random() * 10.0
- **Generating a random int in [0,10):**

  (int)(Math.random() * 10.0)
- **Generating a random int in [5,15):**

  (int)(Math.random() * 10.0 + 5.0)

Random Numbers (cont'd)

- **An event E occurs with probability p.**
- **$0 \leq p \leq 1$**
- **Example: Roll a die. Let E = a roll of 1.**

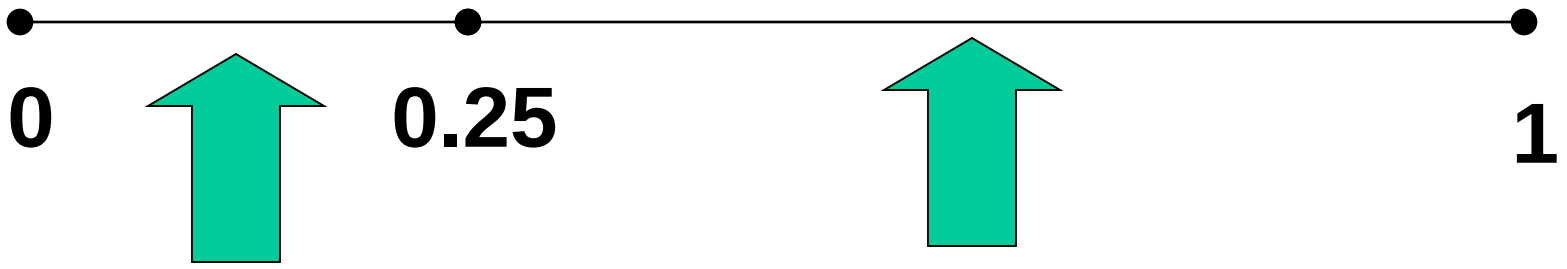
  For this event, p = 1/6 .
- **We need a function such that if we call this function N times, where N is very large, then**

  - the function returns true pN times

  - the function returns false (1-p)N times

BooleanSource

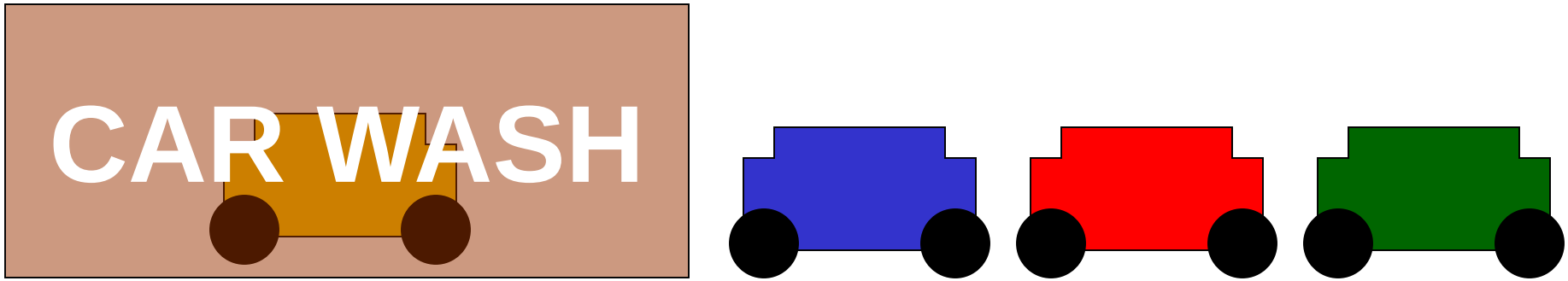
- **The probability of an event occurring is ¼.**
- **Does the event occur?**
- **Generate a uniform random number between 0 and 1. (Any number is equally-likely to be generated in that range.)**



Random Numbers (cont'd)

```
public class BooleanSource
{
  private double probability;
  public BooleanSource(double p) throws ... {
    if (p < 0.0 || p > 1.0)
      throw new IllegalArgumentException();
    probability = p;
  }
  public boolean occurs() {
    return (Math.random() < probability);
  }
}
```

Car Wash Simulation



How long, on average, does a car have to wait once it arrives on line before it is washed?

Car Simulator (cont'd)

- **Assumptions:**

  - One car can be washed at a time.

  - At most one car can arrive per second.
- **Simulation time unit is the second.**
- **Queue will store arrival time of each car.**
- **What can happen during any given second?**

  1. A car can arrive.

  2. A car can enter the car wash.

  3. A car can be washed for one second.

Simulator

```
public static void carWash(int washTime,
  double arrivalProb, int totalTime) {
  if (washTime <= 0 || arrivalProb < 0.0
   || arrivalProb > 1.0 || totalTime < 0)
  {
     System.out.println("NO SIMULATION");
     return;
  }
  // simulation variables
  IntQueue cars = new IntQueue();
  BooleanSource arrival =
          new BooleanSource(arrivalProb);
```

Car Wash Simulation (cont'd)

```
// simulation variables (cont'd)
int totalWaitTime = 0;
int carsWashed = 0;
double avgWaitTime;
int currentSecond;
int timeLeftInWasher = 0;


// loop simulates each second of time
for  ( currentSecond = 1;
       currentSecond <= totalTime;
       currentSecond++)
{
```

Car Wash Simulation (cont'd)

```
// EVENT 1: has a car arrived?
if (arrival.occurs())
     cars.enqueue(currentSecond);

// EVENT 2: can we take a car off the
// queue and put it in the car wash?
if ((timeLeftInWasher == 0)
                && (!cars.isEmpty())) {
   timeLeftInWasher = washTime;
   totalWaitTime +=
     (currentSecond - cars.dequeue());
   carsWashed++;
}
```

Car Wash Simulation (cont'd)

```
      // EVENT 3: wash a car for 1 second
      if (timeLeftInWasher > 0)
         timeLeftInWasher--;
   } // end for loop

   // calculate final statistics
   avgWaitTime =
      (double)totalWaitTime/carsWashed;
   System.out.println("Avg wait time = "
     + avgWaitTime + " seconds.");
}
```