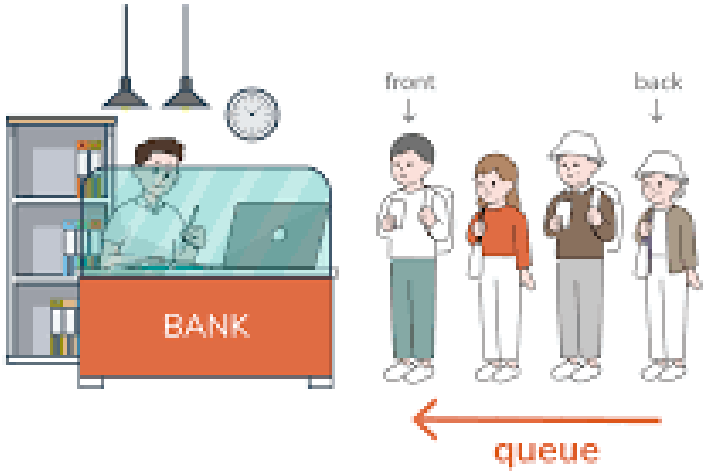
Queues



Queues

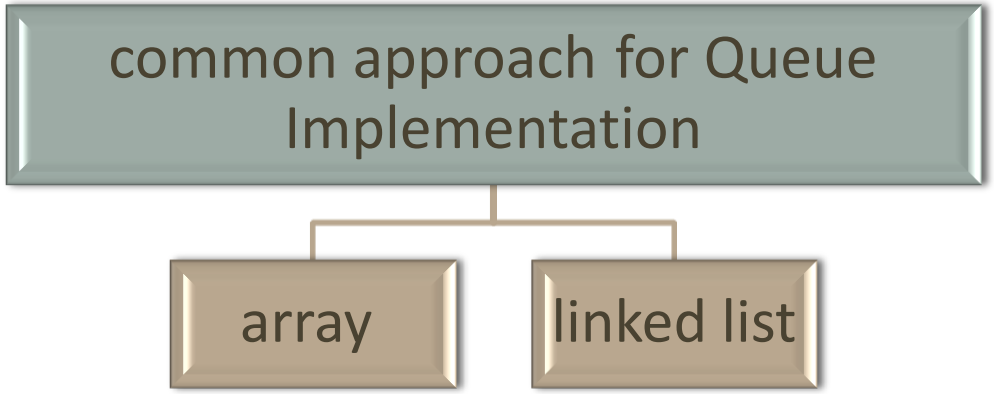
- In computer science a queue is a data structure in which the first item inserted is the first to be removed (First-In-First-Out, FIFO)
- Remember :: in a stack, the last item inserted is the first to be removed (Last-In-First-Out,LIFO)

Queue Applications

- graph search
- a printer queue
- Storing the keyboard keystroke. (if you're using a word processor but the computer is briefly doing something else, the keystroke waits in the queue until the word processor has time to read it. Using a queue guarantees the keystrokes stay in order until they can be processed).
- Modeling real-world situations such as
  - people waiting in line at a bank
  - airplanes waiting to take off
  - Transmitting data packets over the Internet

Queue Implementation

- a queue can be based on an *array* or a *linked lists*



queue operations

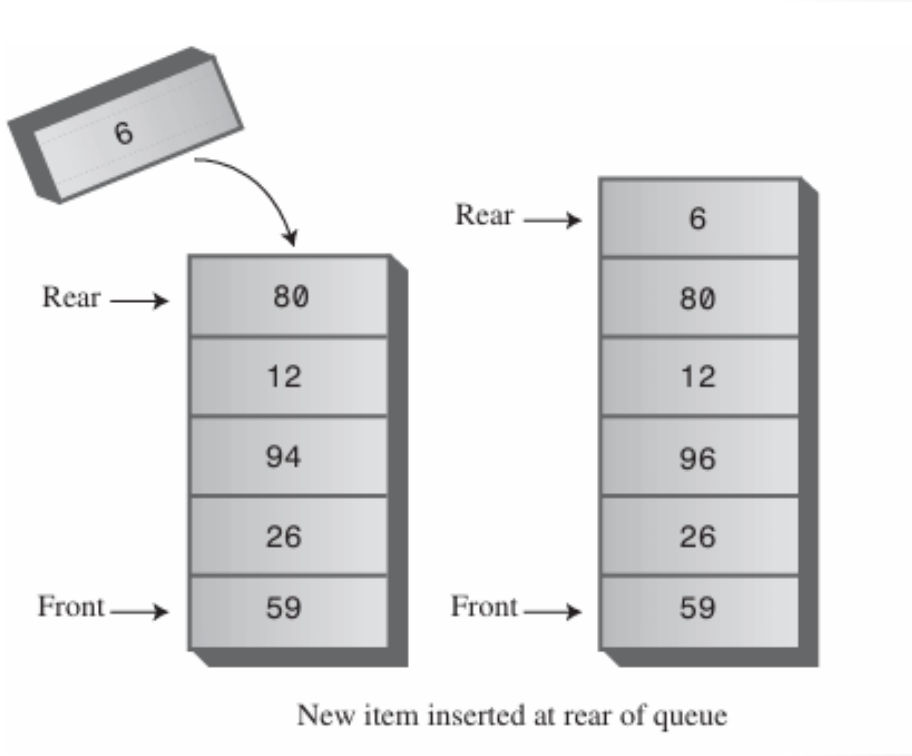
- The basic queue operations are :
  - inserting an item, which is placed at the rear of the queue
  - removing an item, which is taken from the front of the queue.

Standard terms for insertion and removing in stack and queue		
operation	insert	remove
stack	push	pop
queue	Put/add	Delete/get

- The ***rear*** of the queue, where items are inserted, is also called the *back* or *tail* or *end*.
- The ***front***, where items are removed, may also be called the *head*.
- We'll use the terms ***insert, remove, front***, and ***rear***

Queue Insert Operation

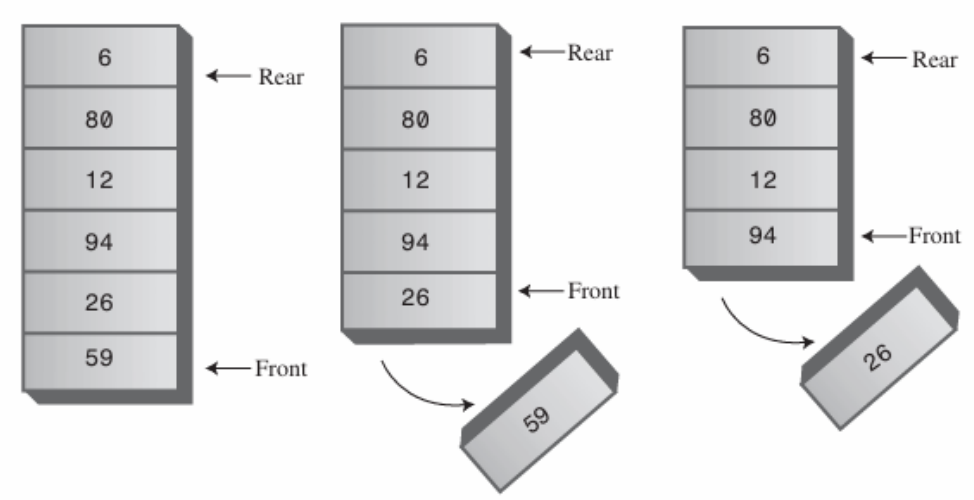
- insert an item in a queue will add it at the rear of the queue and increment the Rear arrow so it points to the new item



New item inserted at rear of queue

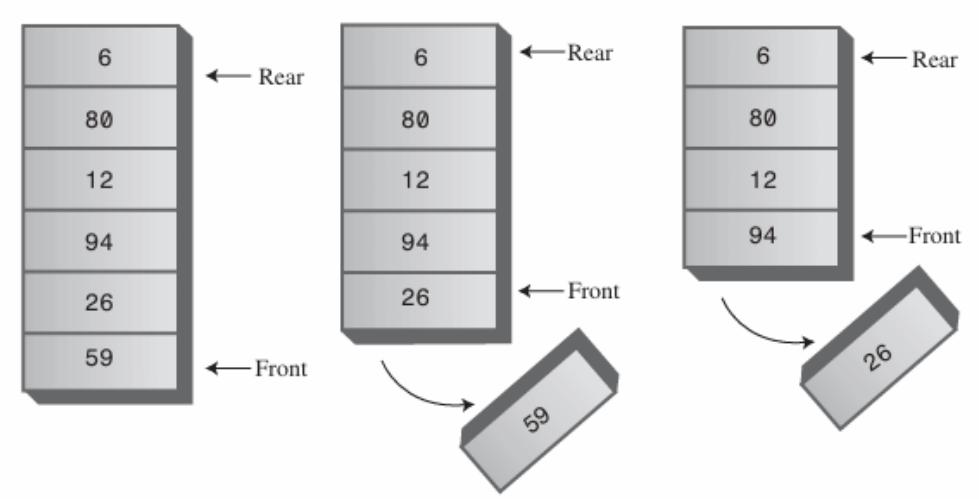
Queue Remove Operation

- when you remove an item at the front of the queue, the item is removed, the item's value is returned as returned value.
- the people in a line at the movies all move forward, toward the front, when a person leaves the line.
- We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient. Instead, we keep all the items in the same place and move the front and rear of the queue



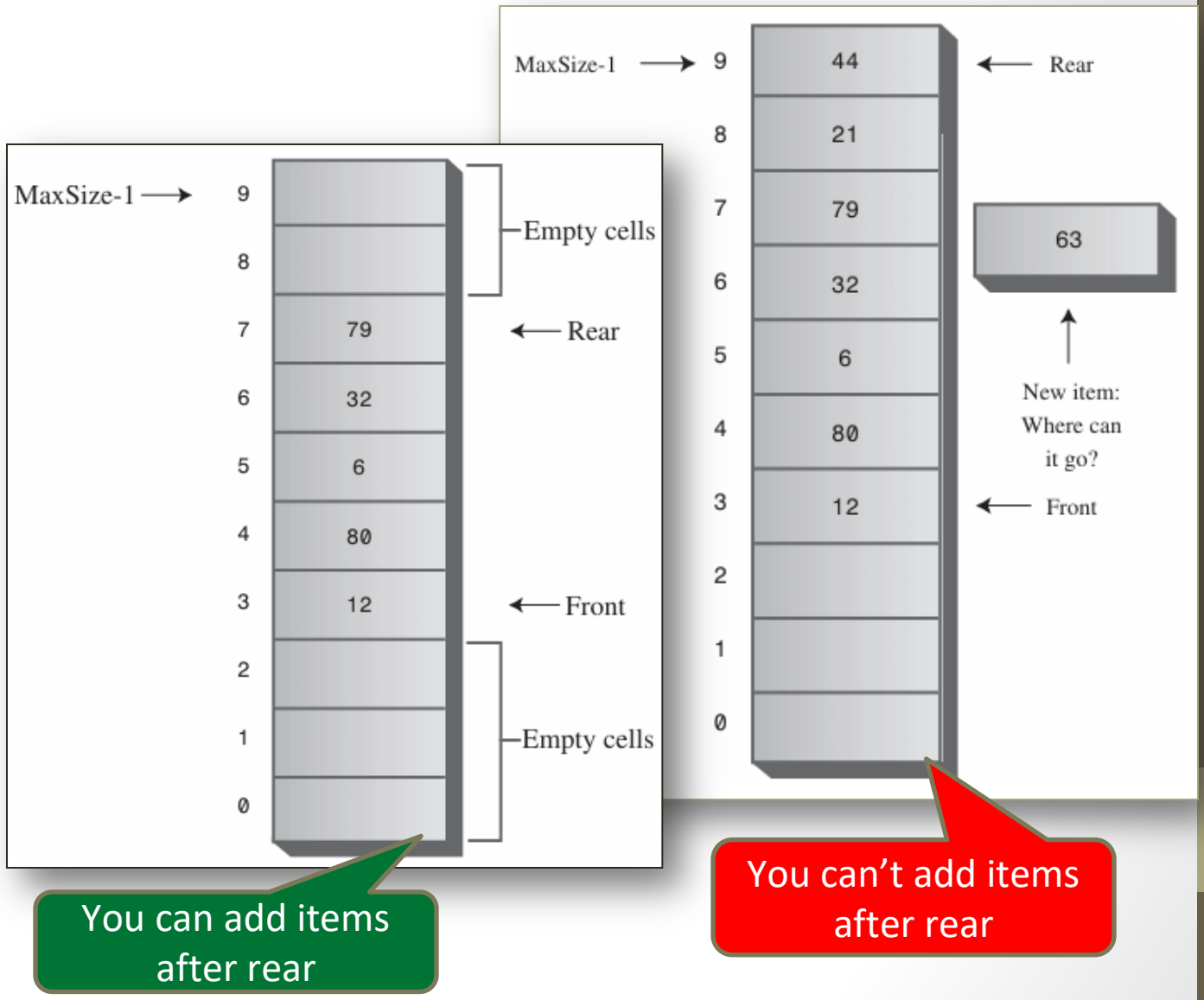
Queue Remove Operation

- when you remove an item at the front of the queue, the item is removed, the item's value is returned as returned value.
- the people in a line at the movies all move forward, toward the front, when a person leaves the line.
- We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient. Instead, we keep all the items in the same place and move the front and rear of the queue



Queue Remove Operation

- when an item is deleted from the queue, we could move all the items in a queue, but that wouldn't be very efficient. Instead, we keep all the items in the same place and move the front and rear of the queue
- The trouble with this arrangement is that soon the rear of the queue is at the end of the array (the highest index). Even if there are empty cells at the beginning of the array, you still can't insert a new item because Rear can't go any further.

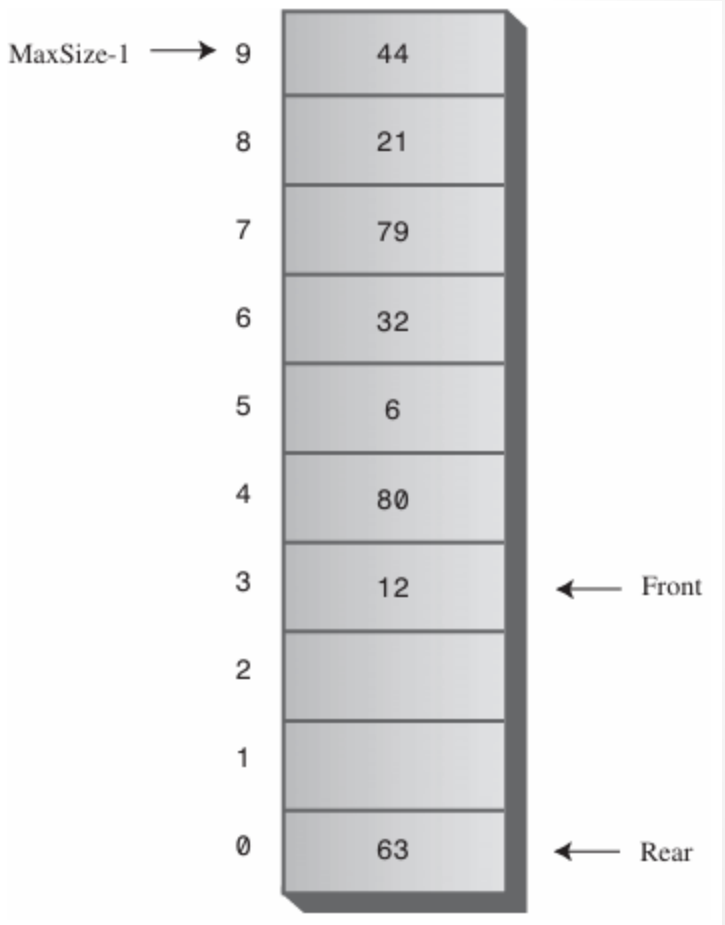


circular queue
Wrapping Around

- To avoid the problem of not being able to insert more items into the queue even when it’s not full, the Front and Rear arrows wrap around to the beginning of the array. The result is a circular queue (sometimes called a ring buffer).

circular queue
Wrapping Around

- Insert enough items to bring the Rear arrow to the top of the array (index 9).
- Remove some items from the front of the array.
- Insert another item. The Rear arrow will wrap around from index 9 to index 0; the new item will be inserted there. This situation is shown in the opposite Figure.
- Insert a few more items. The Rear arrow moves upward. Notice that after Rear has wrapped around, it's now below Front, the reverse of the original arrangement. You can call this a ***broken sequence***.
- Delete enough items so that the Front arrow also wraps around. Now you're back to the original arrangement, with Front below Rear. The items are in a **single contiguous sequence**



Java Code for a Queue

```
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;
//--------------------------------------------------------------
   public Queue(int s)          // constructor
      {
      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }
```

Java Code for a Queue

```
  public void insert(long j)   // put item at rear of queue
     {
     if(rear == maxSize-1)         // deal with wraparound
        rear = -1;
     queArray[++rear] = j;         // increment rear and insert
     nItems++;                     // one more item
     }
'-------------------------------------------------------------
  public long remove()         // take item from front of queue
     {
     long temp = queArray[front++]; // get value and incr front
     if(front == maxSize)           // deal with wraparound
        front = 0;
     nItems--;                      // one less item
     return temp;
     }
```

Java Code for a Queue

```
   public long peekFront()      // peek at front of queue
      {
       return queArray[front];
       }

   public boolean isEmpty()    // true if queue is empty
      {
      return (nItems==0);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true if queue is full
      {
      return (nItems==maxSize);
      }
//--------------------------------------------------------------
   public int size()           // number of items in queue
      {
      return nItems;
      }
//--------------------------------------------------------------
```

Java Code for a Queue

```
class QueueApp
   {
   public static void main(String[] args)
      {
      Queue theQueue = new Queue(5);  // queue holds 5 items
      theQueue.insert(10);            // insert 4 items
      theQueue.insert(20);
      theQueue.insert(30);
      theQueue.insert(40);
      theQueue.remove();              // remove 3 items
      theQueue.remove();              //    (10, 20, 30)
      theQueue.remove();
      theQueue.insert(50);            // insert 4 more items
      theQueue.insert(60);            //    (wraps around)
      theQueue.insert(70);
      theQueue.insert(80);
      while( !theQueue.isEmpty() )    // remove and display
         {                            //    all items
            long n = theQueue.remove();  // (40, 50, 60, 70, 80)
            System.out.print(n);
            System.out.print(" ");
            }
         System.out.println("");
         }  // end main()
      }  // end class QueueApp
```

Queues Efficiency

- inserting and removing items from a queue is O(1) time

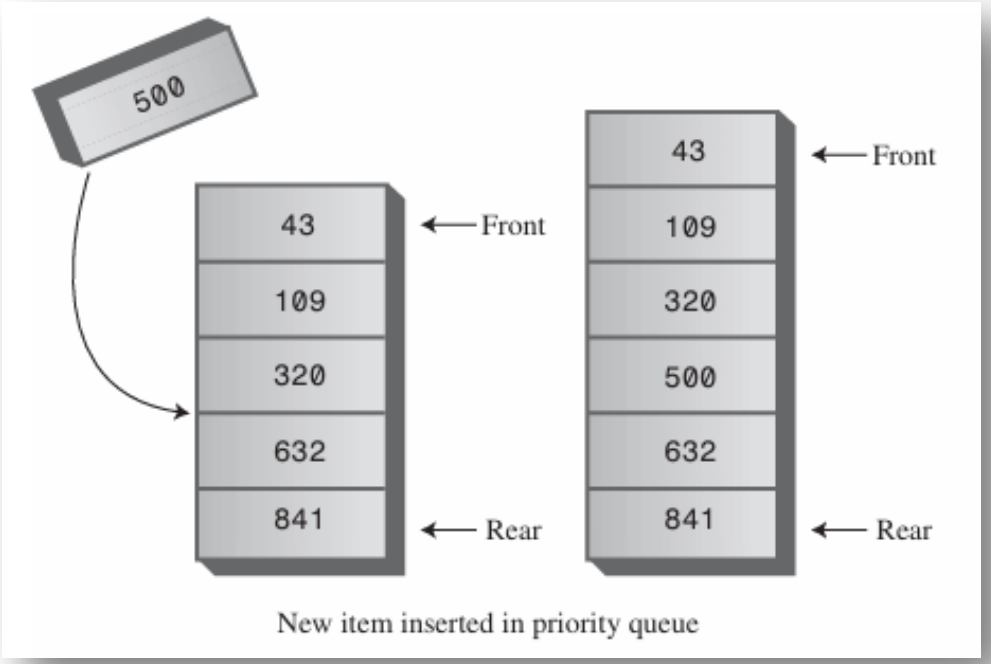
Deques

- A deque is a double-ended queue.
- You can insert items at either end and delete them from either end.
- The methods might be called insertLeft() and insertRight(), and removeLeft() and removeRight().
- If you restrict yourself to insertLeft() and removeLeft() (or their equivalents on the right), the deque acts like a stack.
- If you restrict yourself to insertLeft() and removeRight() (or the opposite pair), it acts like a queue.
- A deque provides a more versatile data structure than either a stack or a queue. However, it's not used as often as stacks and queues, so we won't explore it further here

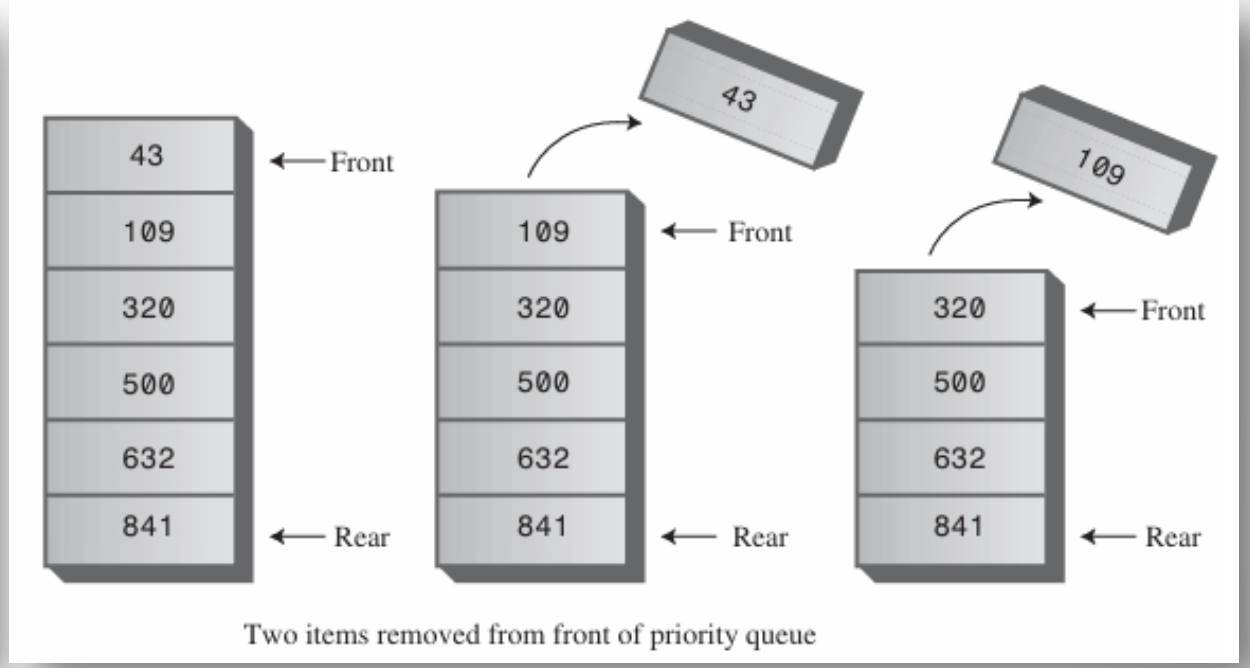
Priority Queues

- A priority queue is a more specialized data structure than a stack or a queue.
- Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front.
- But items are ordered by key value so that the item with the lowest key (or in some implementations the highest key) is always at the front.
- Items are inserted in the proper position to maintain the order.
- It's a useful tool in a surprising number of situations.
  - In a multitasking operating system, for example, programs may be placed in a priority queue so the highest-priority program is the next one to receive a time-slice that allows it to execute.

Priority Queues



New item inserted in priority queue



Two items removed from front of priority queue

Priority Queues implementation

- priority queues are often implemented with a data structure called a ***heap***.
- We'll look at heaps in coming chapters "Heaps"
- By now we'll show a priority queue implemented by a simple array.
  - This implementation suffers from slow insertion, but it's simpler and is appropriate when the number of items isn't high or insertion speed isn't critical.

Priority Queue class

Array Based

```
class PriorityQ
   {
   // array in sorted order, from max at 0 to min at size-1
   private int maxSize;
   private long[] queArray;
   private int nItems;
//--------------------------------------------------------------
   public PriorityQ(int s)          // constructor
      {
      maxSize = s;
      queArray = new long[maxSize];
      nItems = 0;
      }
```

Priority Queue insertion

```
public void insert(long item)    // insert item
   {
   int j;

   if(nItems==0)                          // if no items,
      queArray[nItems++] = item;          // insert at 0
   else                                  // if items,
      {
      for(j=nItems-1; j>=0; j--)          // start at end,
         {
         if( item > queArray[j] )       // if new item larger,
            queArray[j+1] = queArray[j]; // shift upward
         else                           // if smaller,
            break;                      // done shifting
         }  // end for
      queArray[j+1] = item;             // insert it
      nItems++;
      }  // end else (nItems > 0)
   }  // end insert()
```

Priority Queue

```
   public long remove()              // remove minimum item
      { return queArray[--nItems]; }
//--------------------------------------------------------------
   public long peekMin()             // peek at minimum item
      { return queArray[nItems-1]; }
//--------------------------------------------------------------
   public boolean isEmpty()          // true if queue is empty
      { return (nItems==0); }
//--------------------------------------------------------------
   public boolean isFull()           // true if queue is full
      { return (nItems == maxSize); }
//--------------------------------------------------------------
   }  // end class PriorityQ
```

Priority Queues Application

```
class PriorityQApp
   {
   public static void main(String[] args) throws IOException
      {
      PriorityQ thePQ = new PriorityQ(5);
      thePQ.insert(30);
      thePQ.insert(50);
      thePQ.insert(10);
      thePQ.insert(40);
      thePQ.insert(20);

      while( !thePQ.isEmpty() )
         {
         long item = thePQ.remove();
         System.out.print(item + " ");  // 10, 20, 30, 40, 50
         }  // end while
      System.out.println("");
      }  // end main()
//--------------------------------------------------------------
   }  // end class PriorityQApp
```

Efficiency of Priority Queues

Efficiency of Priority Queues

- ✓ insertion runs in O(N) time (the code contains a simple for loop)
- ✓ deletion takes O(1) time (the code contains no looping)