1.00 Lecture 35

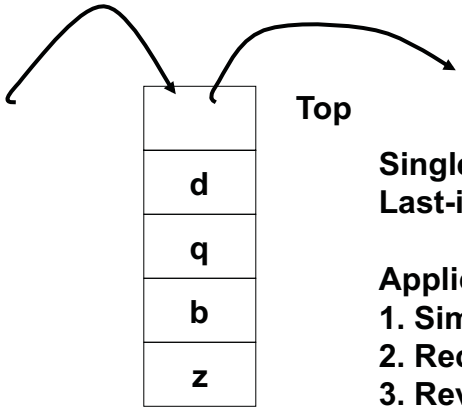
Data Structures:
Introduction
Stacks, Queues

Reading for next time: Big Java: 15.1-15.3

Data Structures

- **Set of reusable classes used in algorithms, simulations, operating systems, applications to:**
  - **Structure, store and manage data required by algorithms**
  - **Optimize the access to data required by algorithms**
- **There is a small number of common data structures**
  - **We cover the basic version of the core structures, except graphs/networks**
  - **Many variations exist on each structure**
- **Three ways to build and use a data structure**
  - **Use the Java built-in version**
  - **Build your own class, using an array to store the data**
  - **Build your own class, using a linked list to store the data**
    - **Use either the Java linked list class or your own (next lecture)**

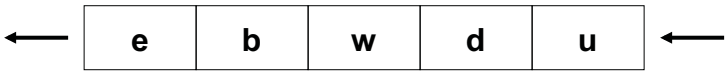
Stacks



Single ended structure
Last-in, first-out (LIFO) list

Applications:
1. Simulation: robots, machines
2. Recursion: pending function calls
3. Reversal of data

Queues

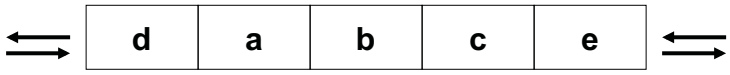


Front Rear

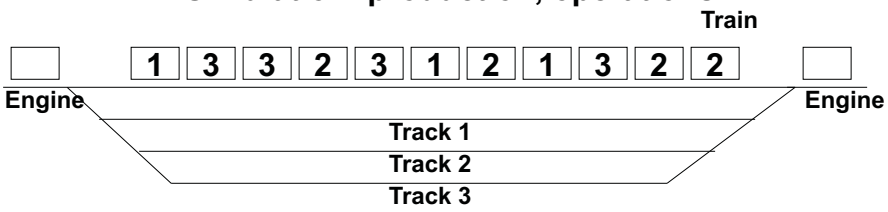
Double ended structure
First-in, first-out (FIFO) list

Applications:
1. Simulation: lines
2. Ordered requests: device drivers, routers, …
3. Searches

Double ended Queues (Dequeues)

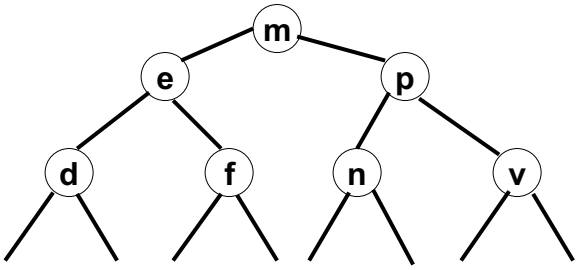


Double ended structure
Applications:
1. Simulation: production, operations



A dequeue can model both stacks and queues

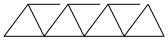
Binary Trees



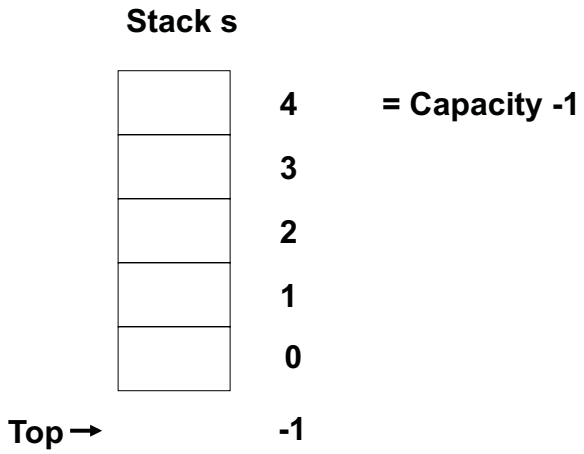
Level	Nodes
0	2^0
1	2^1
2	2^2
...	
k	2^k

- Binary tree has $2^{(k+1)}$-1 nodes
- A maximum of k steps are required to find (or not find) a node
  E.g. 2^{20} nodes, or 1,000,000 nodes, in 20 steps!
- Binary trees can be built in many ways: search trees, heaps…
- Applications: fast storage and retrieval of data, search/sort
  priority queues (heaps)
- A few algorithms use general trees (variable number of nodes):
  sets (union, intersection), matrices (bases), graphics
- Hashing is a non-tree alternative for fast storage/retrieval of data

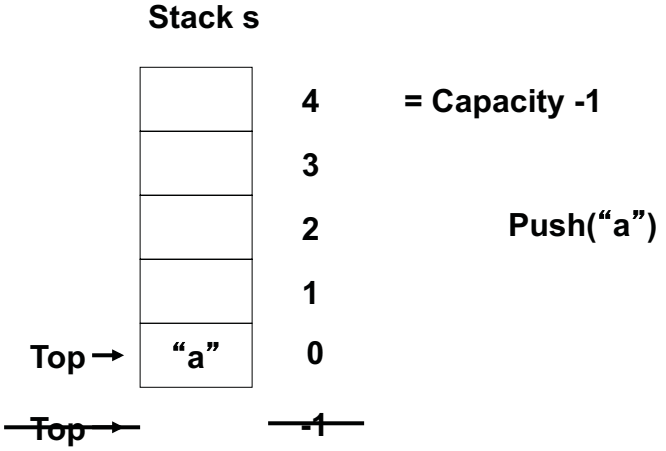
Exercise 1

- **What data structure would you use to model:**
  - **People getting on and off the #1 bus at the MIT stop thru front and back doors**
  - **A truss in a CAD program** 
  - **A conveyor belt**
  - **The emergency room at a hospital**
  - **The lines at United Airlines at Logan**
  - **The Cambridge street network**
  - **Books to be reshelved at the library**

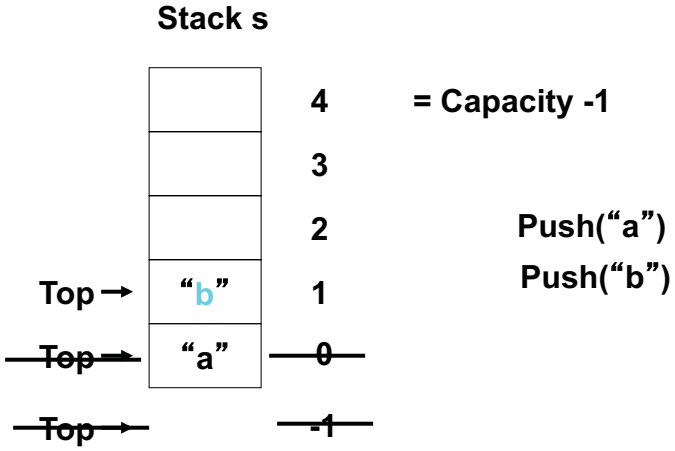
Stacks



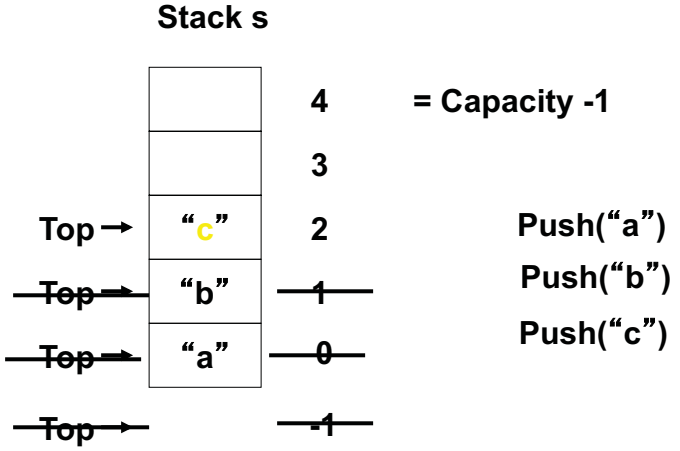
Stacks



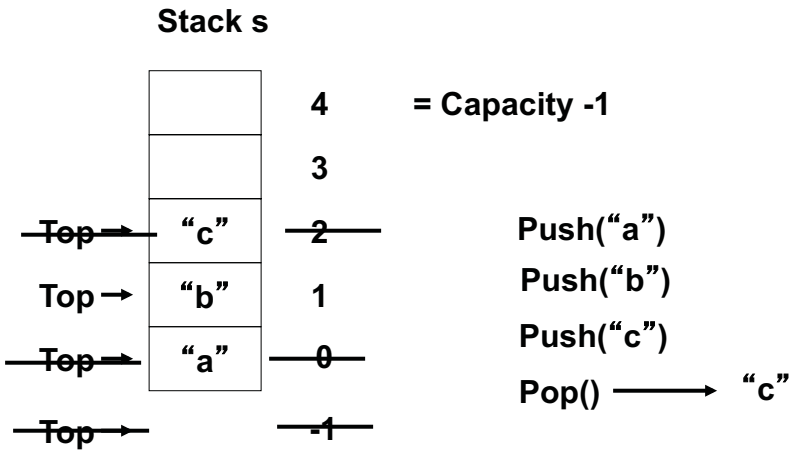
Stacks



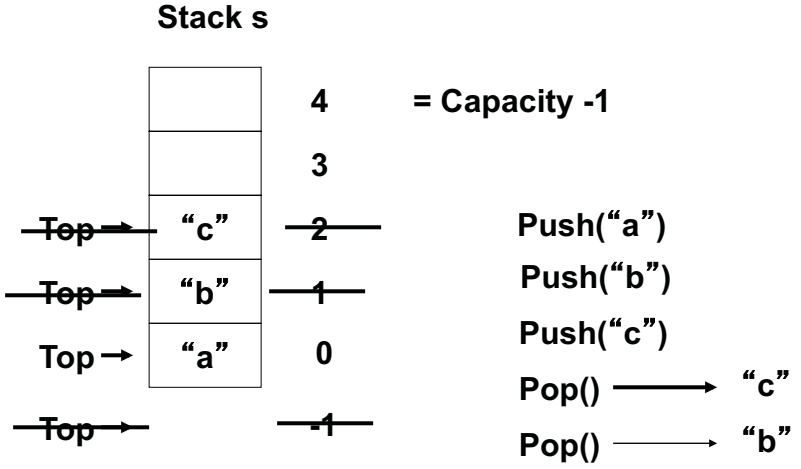
Stacks



Stacks



Stacks



Stack Interface

```
import java.util.*;        // For exception

public interface Stack
{
    public boolean isEmpty();
    public void push( Object o );
    public Object pop() throws
              EmptyStackException;
    public void clear();
}

// Interface Stack is an abstract data type
// We will implement ArrayStack as a concrete
// data type, to the Stack specification
// We don't use generics (<T>) here because they don't
// interact cleanly with simple arrays. Code requires
// ugly casts. Generic version in download.
```

Using a Stack to Reverse an Array

```
public class Reverse {
   public static void main(String args[]) {
       int[] array = { 12, 13, 14, 15, 16, 17 };
       Stack stack = new ArrayStack();
       for (int i = 0; i < array.length; i++) {
          Integer y= new Integer(array[i]);
          stack.push(y);
       }
       while (!stack.isEmpty()) {
          Integer z= (Integer) stack.pop();
          System.out.println(z);
       }
   }
}

// Output: 17 16 15 14 13 12
```

ArrayStack, 1

```
// Download ArrayStack; you'll be writing parts of it
// Download Stack and Reverse also.

import java.util.*;

public class ArrayStack implements Stack {
   public static final int DEFAULT_CAPACITY = 8;
   private Object[] stack;
   private int top = -1;
   private int capacity;

   public ArrayStack(int cap) {
      capacity = cap;
      stack = new Object[capacity];
  }
  public ArrayStack() {
     this( DEFAULT_CAPACITY );
  }
```

Exercise 2: ArrayStack, 2

```
public boolean isEmpty() {
  // Complete this method (one line)
}

public void clear() {
  // Complete this method (one line)
}
```

Exercise 3: ArrayStack, 3

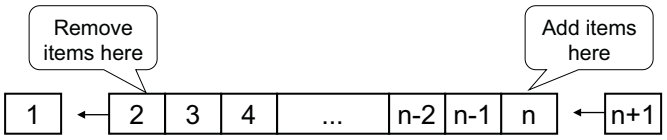
```
public void push(Object o) {
     // Complete this code
     // If stack is full already, call grow()

}

private void grow() {
  capacity *= 2;
  Object[] oldStack = stack;
  stack = new Object[capacity];
  System.arraycopy(oldStack, 0, stack, 0, top+1);
}
```

Exercise 4: ArrayStack, 4

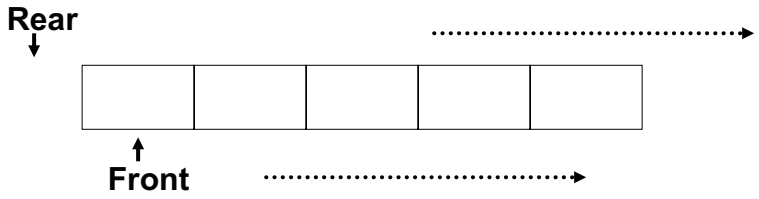
```
public Object pop()
   throws EmptyStackException
 {
      // Complete this code
      // If stack is empty, throw exception
 }


// When you finish this, save/compile and run Reverse
```

Queues

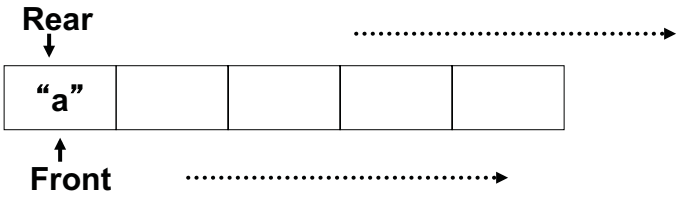
A *queue* is a data structure to which you add new items at one end and remove old items from the other.



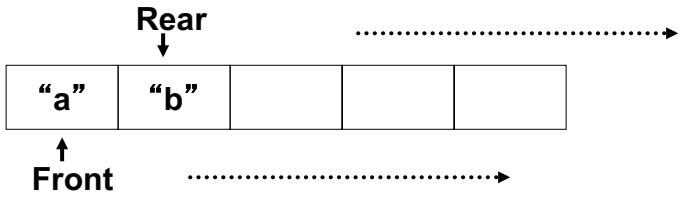
Queue



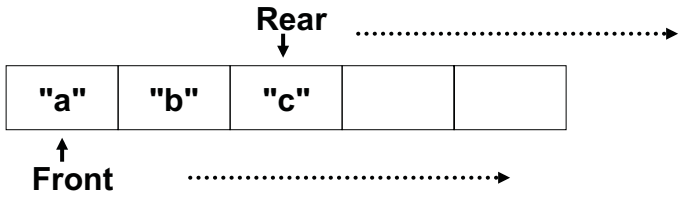
Queue



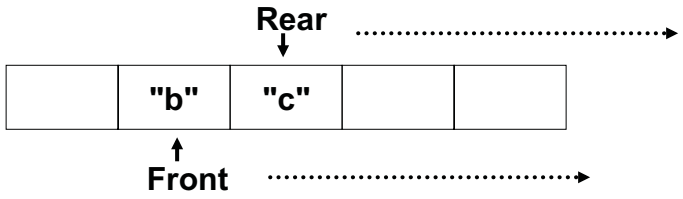
Queue



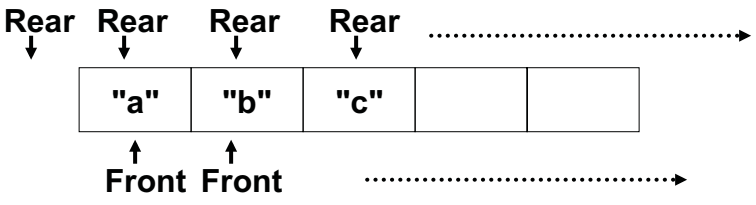
Queue

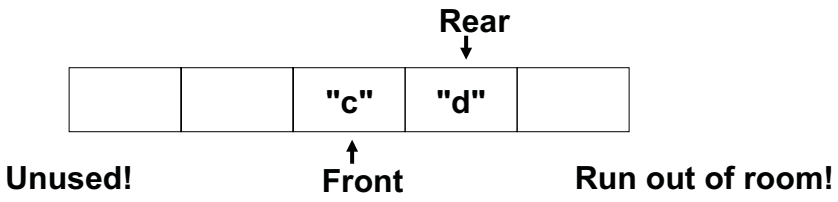


Queue

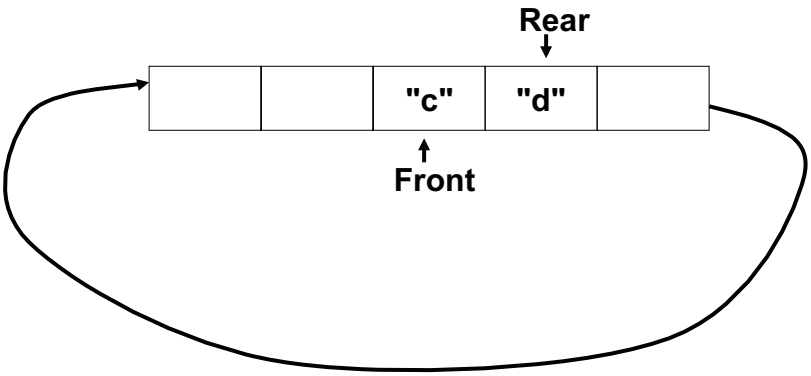

Rear
"b"
"c"
Front

Queue


Rear
Rear
Rear
Rear
"a"
"b"
"c"
Front
Front


Rear
"c"
"d"
Unused!
Front
Run out of room!

Queue

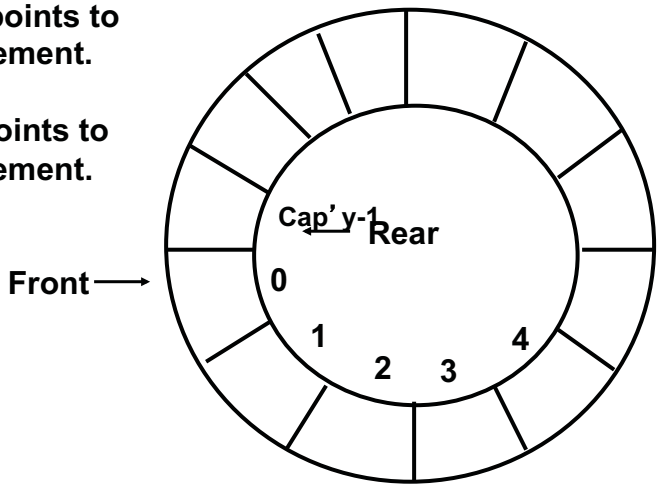


Wrap around!

Ring Queue

Front points to first element.

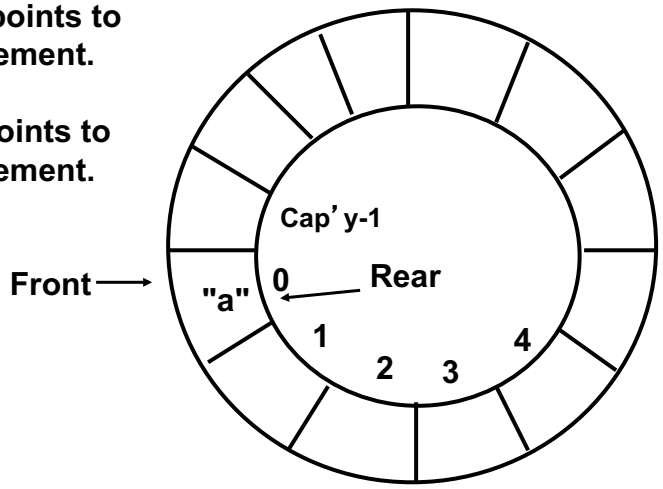
Rear points to rear element.



Ring Queue

Front points to first element.

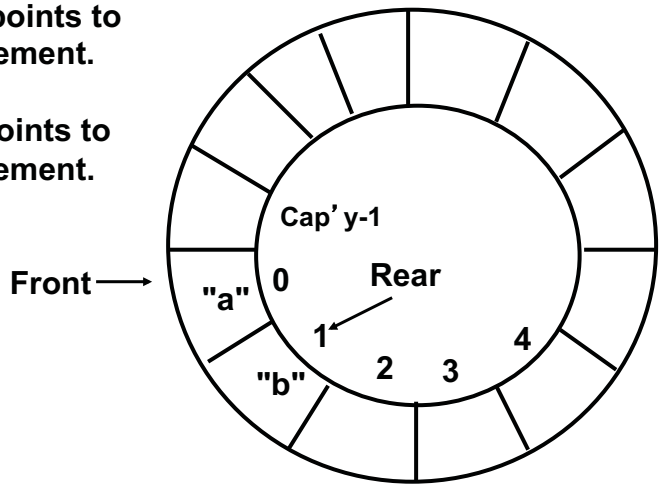
Rear points to rear element.



Ring Queue

Front points to first element.

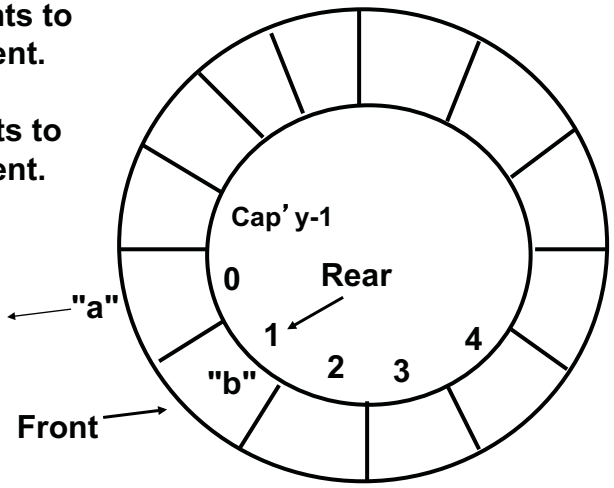
Rear points to rear element.



Ring Queue

Front points to first element.

Rear points to rear element.

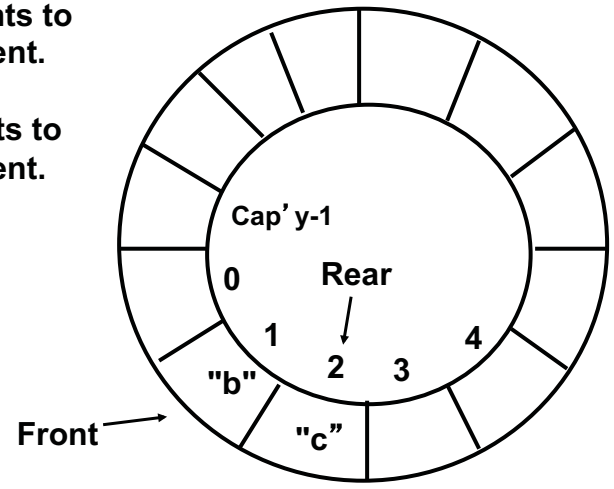


("a" is still in slot 1 but we don't show it)

Ring Queue

Front points to first element.

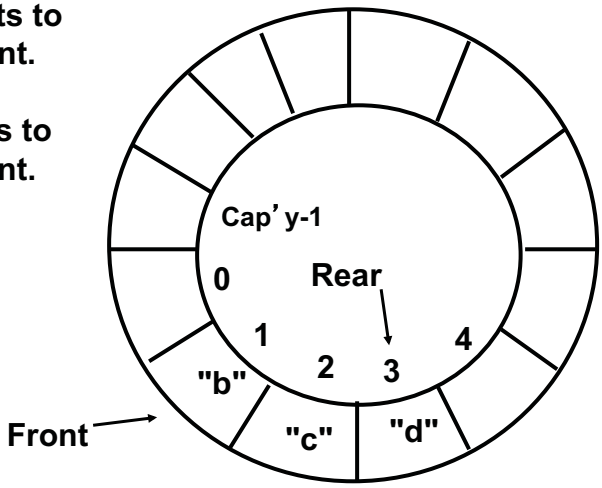
Rear points to rear element.



Ring Queue

Front points to first element.

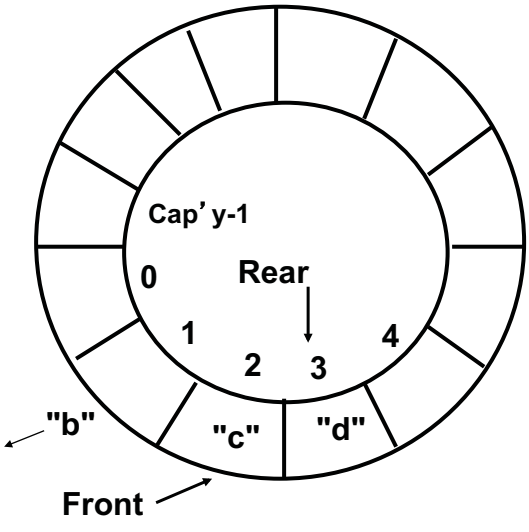
Rear points to rear element.



Ring Queue

Front points to first element.

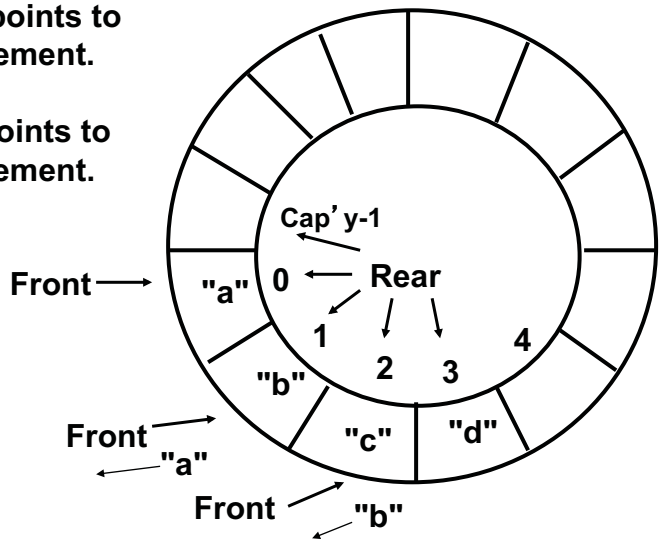
Rear points to rear element.



Ring Queue

Front points to first element.

Rear points to rear element.



Queue Interface

```
import java.util.*;

public interface Queue
{
    public boolean isEmpty();
    public void add( Object o );
    public Object remove() throws
      NoSuchElementException;
    public void clear();
}
```

Implementing a Ring Queue

```
public class RingQueue implements Queue {
      private Object[] queue;
      private int front;
      private int rear;
      private int capacity;
      private int size = 0;
      static public final int DEFAULT_CAPACITY= 8;
```

RingQueue Data Members

queue:	Holds a reference to the ring array
front:	If size>0, holds the index to the next item to be removed from the queue
rear:	If size>0, holds the index to the last item that was added to the queue
capacity:	Holds the size of the array referenced by queue
size:	Always >=0. Holds the number of items on the queue

Exercise 5: RingQueue Methods 1

```
public RingQueue(int cap) {
    capacity = cap;
    front = 0;
    rear = capacity - 1;
    queue= new Object[capacity];
}

public RingQueue() {
   this( DEFAULT_CAPACITY );
}

public boolean isEmpty() {
    // Complete this method
}

public void clear() {
    // Complete this method
}
```

Exercise 6: RingQueue Methods 2

```
public void add(Object o) {
       // Complete this method
       // If queue full, call grow()
       // Add Object to rear, update size
   }

 public Object remove() {
   if ( isEmpty() )
     throw new NoSuchElementException();
   else {
     // Complete this method
     // Get front object, move front index
     // Update size, return front object
   }
 }
// See download code for grow() method
```

Java Dequeue, ArrayDeque

- For stacks, queues and dequeues, Java has
  - interface Deque
  - class ArrayDeque implements Deque
- For a stack use:
  - push() and pop()
- For a queue use:
  - addLast() and removeFirst()
- For a deque use:
  - addFirst(), addLast(), removeFirst(), removeLast()
- For all use:
  - clear(), contains(), isEmpty(), size(), etc.
  - See javadoc for ArrayDeque, Deque

Deque example: stack, queue

```
import java.util.*;
public class DequeExample {
   public static void main(String[] args) {
        // Stack: use only push() and pop()
        Deque<String> stack= new ArrayDeque<String>();
        stack.push("Al");
        stack.push("Bob");
        stack.push("Claire");
        stack.push("Deb");
        System.out.println("Stack:");
        while (!stack.isEmpty())
                System.out.println(stack.pop());
        // Queue: use only addLast() and removeFirst()
        Deque<String> queue= new ArrayDeque<String>();
        queue.addLast("Al");
        queue.addLast("Bob");
        queue.addLast("Claire");
        queue.addLast("Deb");
        System.out.println("\nQueue:");
        while (!queue.isEmpty())
                System.out.println(queue.removeFirst()); } }
```

Example output

```
Stack:
Deb
Claire
Bob
Al

Queue:
Al
Bob
Claire
Deb
```

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012