

# 1.00 Lecture 26

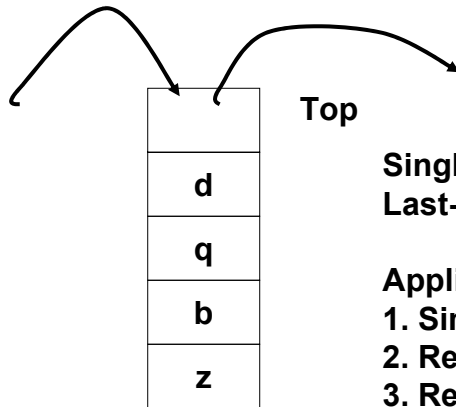
## Data Structures: Introduction Stacks

Reading for next time: Big Java: 19.1-19.3

## Data Structures

- **Set of primitives used in algorithms, simulations, operating systems, applications to:**
  - Store and manage data required by algorithm
  - Provide only the access that is required
  - Disallow all other access
- **There are a small number of common data structures**
  - We cover the basic version of the core structures
  - Many variations exist on each structure
    - It's common to make application-specific modifications
- **We'll both build them and use the Java built-in versions!**

# Stacks

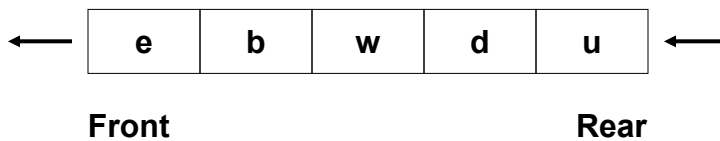


Single ended structure  
Last-in, first-out (LIFO) list

Applications:

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

# Queues

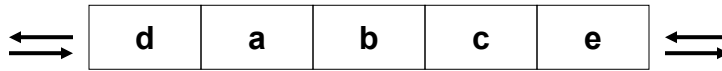


Double ended structure  
First-in, first-out (FIFO) list

Applications:

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

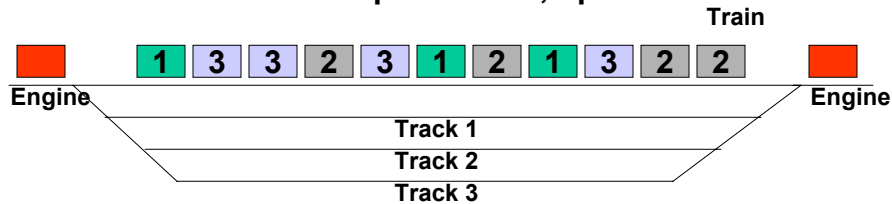
## Double ended Queues (Dequeues)



Double ended structure

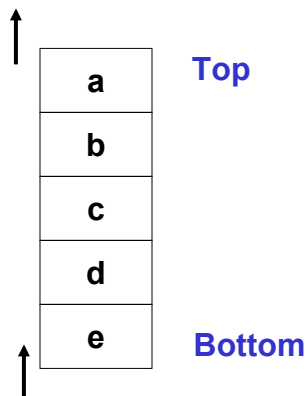
Applications:

1. Simulation: production, operations



A dequeue can model both stacks and queues

## Priority Queues or Heaps

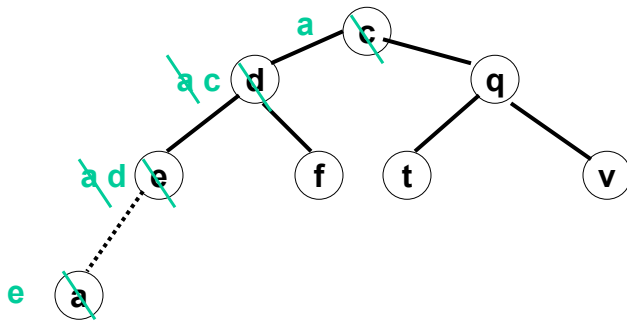


- Highest priority element at top
- “Partial sort”
- All enter at bottom, leave at top

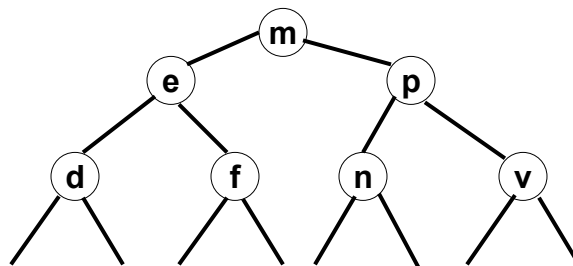
Applications:

1. Simulations: event list
2. Emergency response modeling
3. Searching (next most likely)

## Heaps Modeled as Binary Tree



## Binary Trees



<u>Level</u>	<u>Nodes</u>
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: heaps, search trees...

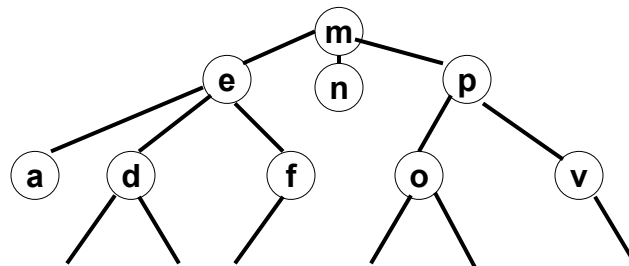
# Binary Trees

## Applications:

1. Searching and sorting (general purpose)
2. Fast retrieval of data

Find/insert any item quickly (bottom, middle or top)  
More general than earlier data structures

# Trees

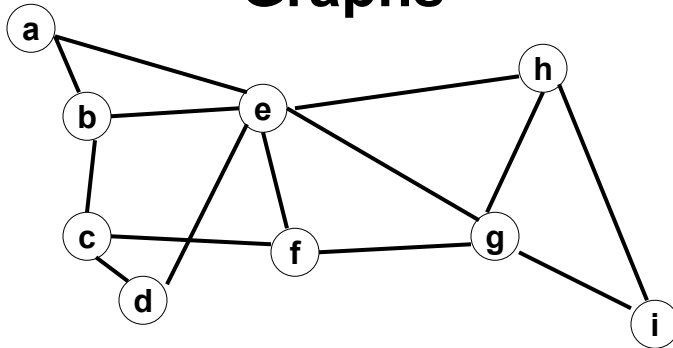


Each node has variable number of children

## Applications:

1. Set operations (unions, intersections)
2. Matrix operations (basis representation, etc.)
3. Graphics and spatial data (quadtrees, R-trees, ...)

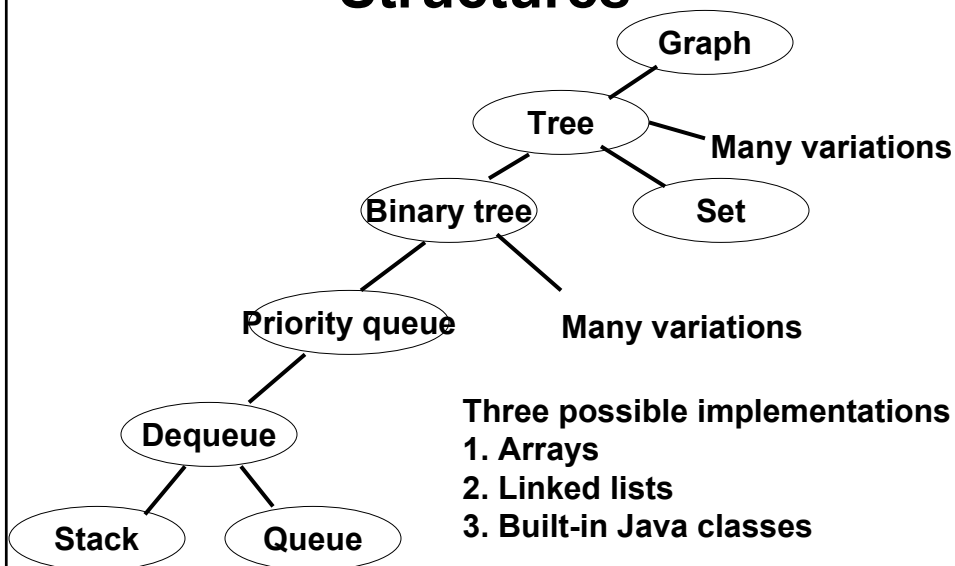
# Graphs



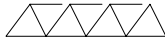
## Applications

1. Simulation
2. Matrix representation
3. General systems representation
4. Networks: telecom, transport, hydraulic, electrical, ...

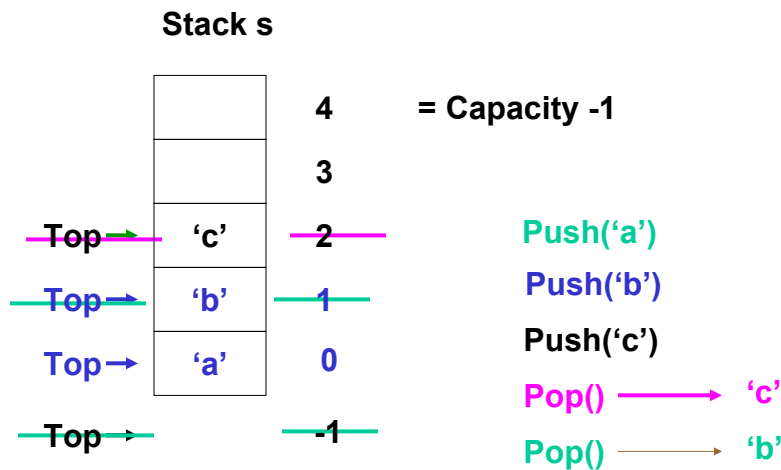
# Relationships of Data Structures



# Exercise

- 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
  - How to go from MIT to all pts in Boston
  - Books to be reshelfed at the library

# 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
```

## 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: ArrayStack, 2

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

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

## Exercise: 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);
}
```

## Exercise: ArrayStack, 4

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

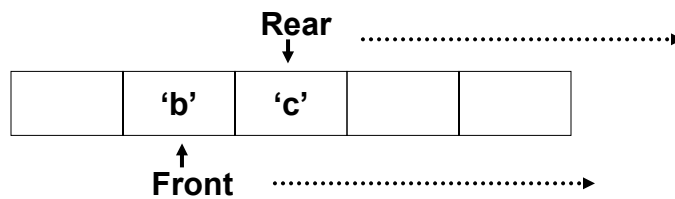
// when you finish this, save/compile and run Reverse
```

# Queues

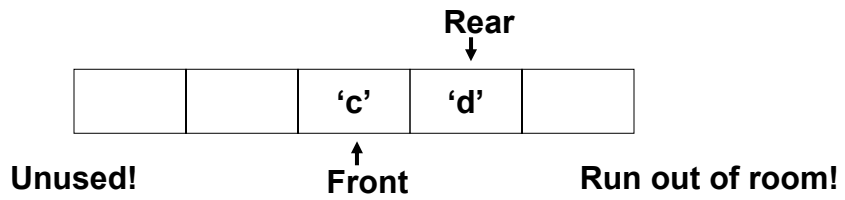
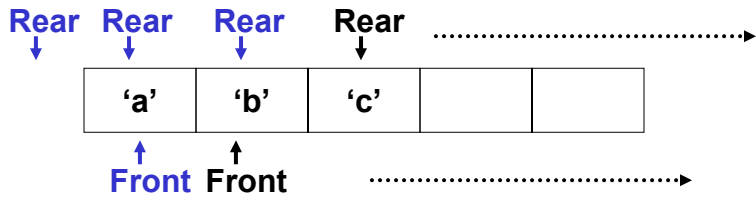
A *queue* is a data structure to which you add new items at one end and remove old items from the other.



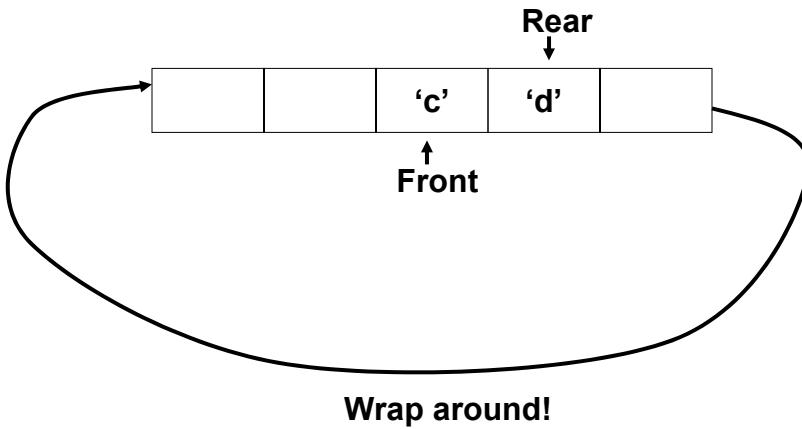
# Queue



# Queue



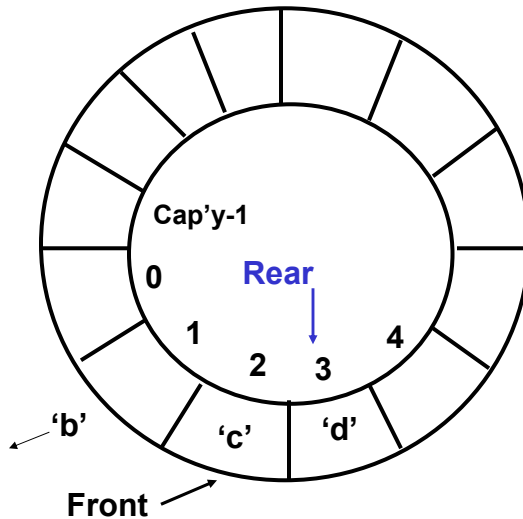
# Queue



# Ring Queue

Front points to first element.

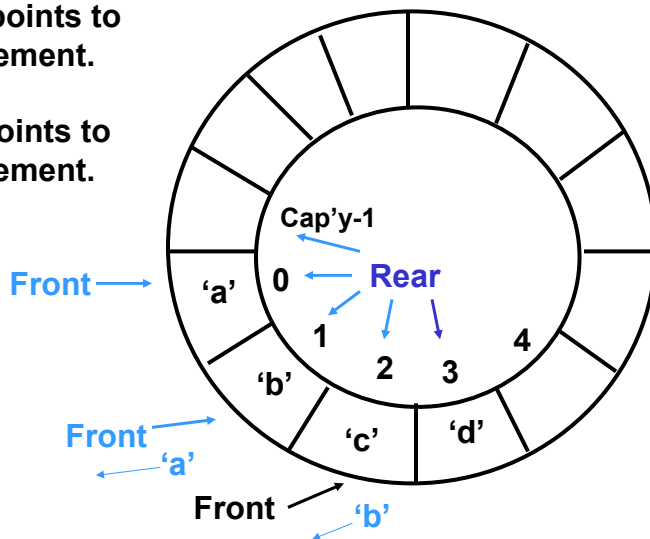
Rear points to rear element.



# 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

<b>queue:</b>	Holds a reference to the ring array
<b>front:</b>	If <code>size&gt;0</code> , holds the index to the next item to be removed from the queue
<b>rear:</b>	If <code>size&gt;0</code> , holds the index to the last item that was added to the queue
<b>capacity:</b>	Holds the size of the array referenced by queue
<b>size:</b>	Always $\geq 0$ . Holds the number of items on the queue

## RingQueue Methods

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

public RingQueue() {
    this( DEFAULT_CAPACITY );
}

public boolean isEmpty() {
    return ( size == 0 );
}

public void clear() {
    size = 0;
    front = 0;
    rear = capacity - 1;
}
```

## RingQueue Methods

```
public void add(Object o) {
    if ( size == capacity )
        grow();
    rear = ( rear + 1 ) % capacity;
    queue[ rear ] = o;
    size++;
}

public Object remove() {
    if ( isEmpty() )
        throw new NoSuchElementException();
    else {
        Object ret = queue[ front ];
        front = (front + 1) % capacity;
        size--;
        return ret;
    }
}
// See download code for grow() method
```

## Exercise

- **Download:**
  - QueueSimulation
  - RingQueue
  - ColorUtil
- **Run QueueSimulation**
  - Experiment with it
  - It runs the opposite way around the ring as our earlier example but is implemented the same way
    - Green points to front, red points to rear of queue