Announcements

• FINAL EXAM Monday May 21, 1:30pm
• Review Session
  – Wednesday May 16, 7-9pm
Recitation 12

Root Finding, Sorting, Stacks, Queues
Outline

• Linked Lists
• Sorting
• Queues
Let’s start with a class called “Node”

Node.java

class Node{
}

Node (Node object)

To create an instance of Node:

Node n = new Node();
Node n2 = n;

These are object references. Notice how there is only ONE object, and each reference “refers” to it.
Fun with References

Let’s add to our Node class

Now, let’s use the following code:

```java
Node n1 = new Node();
Node n2 = new Node();
n2.next = n1;
n1.next = n2;
```
Serious Work with References

Let’s add a data member to store some information. (It could be a primitive type or an object reference... or anything else you want)

```java
class Node{
    Node next;
    int value;
}
```

Now, let’s use the following code:

```java
Node first = new Node();
first.value=1;
first.next = new Node();
first.next.value=2;
first.next.next = new Node();
first.next.next.value = 3;
```

It’s a Linked List!
Linked Lists

• A linked list is made of a series of Nodes, each with:
  – an associated item object
  – a reference to the next node of the list

• Simplified “double-box” picture:

• Traverse the list by following each node’s “next” reference
For convenience, create a class with references to the first & last nodes, with methods, so we don’t have to re-write the manipulation code each time.

```java
public class SLinkedList implements List{
    private int length = 0;
    private Node first = null;
    private Node last = null;

    private static class Node {
        Object item;
        Node next;
        Node( Object o, Node n ){ item = o; next = n; }
    }

    public int size() { /*code...*/ }
    public boolean isEmpty() { /*code...*/ }
    public boolean contains(Object o) { /*code...*/ }
    public void clear() { /*code...*/ }
    // various add() and remove() methods...
}
```
What’s so great about Linked Lists?

BIG Pros:

• A Linked List can grow dynamically.
  (To resize an array you have to create a new, larger array, and copy everything over)
• A linked list does NOT need contiguous memory.
  (A Java 1-D array has to occupy contiguous memory. When storing large amounts of data, finding back-to-back-to-back... memory can be impossible)

Cons:

• The references add overhead.
• Access is slower than an array.
• The code to maintain a Linked List can be complex.
• Depending on how the ‘links’ (references) are structured, you may only be able to traverse one way...
Linked List: Tips

- Always think of special cases
  - What if your list is empty?
  - What if there is only one element?

- Always draw a diagram!
Sorting

• Sortable objects implement Comparable<Object> or have Comparator defined

• Comparable:
  – Define compareTo()
  – For object.compareTo(other):
    • returns 1 if other higher ranked than object
    • returns 0 if equally ranked
    • returns -1 otherwise
Sorteding

• Comparator:

  – New class `Object1Object2Comparator`
  – Implements `Comparator<Object>`
  – Must define `compare()`. For `compare(a, b)`:  
    • returns 1 if `b` higher ranked than `a`
    • return 0 if equally ranked
    • returns -1 otherwise
Sorting Exercise

• Sort restaurants by rating (high to low) then distance (close to far)

```java
public class Restaurant {
    String name;
    int rating;
    double distance;

    public Restaurant(String n, int r, double d){
        name = n;
        rating = r;
        distance = d;
    }

    public String toString(){
        return name + ": " + rating + "/5.0, " + distance + " meters away.";
    }
}
```
Stacks and Queues

• Structures store, manage data
• For data with an inherent order
  – Think of structures like a line to get into a movie
• Stacks: people are added and removed from same end of line
  – Last person in an elevator is the first person out of the elevator
• FIFO Queue: people added to back of line, removed from front
  – First In First Out, the way you expect a ticket line to work
Stacks

• Single end
• LIFO: Last In First Out
  – push(): add an element
  – pop(): remove top element
• Applications:
  – Simulation: robots, machines
  – Recursion: pending function calls
  – Reversal of data
Stack Interface

import java.util.*;  // For exception

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

• Two ends
• FIFO: First In First Out
  – push(): add an element to top
  – pop(): remove bottom element
• Applications:
  – Simulation: lines
  – Ordered requests: device drivers, routers, ...
  – Searches
Queue Interface

import java.util.*;

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

• What is the final output?
  – Add \{2, 4, 6, 8\} to a stack #1
  – Remove three items from stack, place in queue
  – Remove two items from queue, place in stack #2
  – Remove one item from stack #2, place in queue
  – Remove one item from stack #1, place in stack #2
Exercise

• Write a class to store a queue in a linked list
  – What happens when you remove the last object?
  – What happens when you try to remove an object from an empty list?

```java
public interface Queue {
    public void enqueue(int item); // add to end
    public int dequeue() throws Exception; // remove from front
}
```