Topics for Thursday - 10/5/2000

1. Interators
2. Revisit WordFrequency example use of HashMap
3 Ordering and Sorting - The Comparable Interface versus the Comparator Interface.
4. Use of Sorted Collections, Sets and Maps
5. One implementation of a tree store/search algorithm.
6. Performance Comparisons for Trees and HashTables

java.util Interface Iterator
All Known Subinterfaces: ListIterator

public interface Iterator
An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:
•Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
•Method names have been improved.
Method Summary

```
boolean hasNext()
Returns true if the iteration has more elements.
Object next()
Returns the next element in the interation.
void   remove()
Removes from the underlying collection the last element
returned by the iterator (optional operation).
```


public interface ListIterator extends Iterator
An iterator for lists that allows the programmer to traverse the list in either direction and modify the list during iteration.
Method Summary

```
void   add(Object o)
Inserts the specified element into the list (optional
operation).
boolean      hasNext()
Returns true if this list iterator has more elements
when traversing the list in the forward direction.
boolean      hasPrevious()
Returns true if this list iterator has more elements
when traversing the list in the reverse direction.
Object next()
Returns the next element in the list.
```



```
int    nextIndex()
Returns the index of the element that would be returned by
a subsequent call to next.
Object previous()
Returns the previous element in the list.
int    previousIndex()
Returns the index of the element that would be returned by
a subsequent call to previous.
void   remove()
Removes from the list the last element that was returned
by next or previous (optional operation).
void   set(Object o)
Replaces the last element returned by next or previous
with the specified element (optional operation).
```



```
import java.util.*;
import java.io.*;
public class WordFrequency
{
static public void main(String[] args)
{  HashMap words = new HashMap();
/* Constructs a new, empty map with a default capacity.
        A HashMap contains a (key,value) pair.  Each of key
        and value may be an instance of a user defined
        class.  The actual key and value types are
        determined when the entries are made. */
   Count count;
/* Remember that Count is a class with two instance variables, a String and an int.
        The string will be a word from the text file and the int will be the number
        of times that word appears in the text file. */
   String delim = " \t\n.,:;?!-/()[]\"\'";
```



```
try {
   BufferedReader in = new BufferedReader(new FileReader(args[0]));
/* Declares an input streamto be read from a file and declares that the file to be
        read is named in the first argument of the program invocation at the
        command line.  */
    String line, word;
     while ((line = in.readLine()) != null)
     {  /*Reads in a line of the text file, quits when there are no more lines in the
                file. */
     StringTokenizer st = new StringTokenizer(line, delim);
     /* Declares and instance of a StringTokenizer class which extracts a word
        from the text file. */
     while (st.hasMoreTokens())
     { /* Checks for end of line. */
        word = st.nextToken().toLowerCase();
        count = (Count) words.get(word);
        /*  Checks for entry in HashMap with key = (value of) word.)
```

```
if (count == null)
{ /* If an entry with key = (value of) word is not present, add an entry
        consisting of the word and a new instance of Count to the HashMap. */
   words.put(word, new Count(word, 1));
  /* If an entry with key = (value of) word is present then just increment the
        variable which counts the number of occurrences of this word. */
  } else { count.i++; }
 }
}
}
 catch (FileNotFoundException e1) {}
 catch (IOException e2) {}
```



```
 Set set = words.entrySet();
 / Declare and instantiate a Set set which has as its elements the entries which
occur in the HashMap. */
 Iterator iter = set.iterator();
 /* Declare and instantiate an iterator to loop through all the instances of the
      set of Count instances. */
 while (iter.hasNext()) {
 Map.Entry entry = (Map.Entry) iter.next();
 /* Gets a entry from the Set set and casts it to reference type Map.Entry */
 word = (String) entry.getKey();
 /* Extract the key from each entry in the the HashMap. */
 count = (Count) entry.getValue();
/* Extract the value from each entry in the HashMap and cast it to Count. */
 System.out.println(word +  (word.length() < 8 ? "\t\t" : "\t") +  count.i);
/ Print a table of entries. */
 }
}
```



```
static class Count
/* Count is a class with two instance variables, a String and an int.  The string
        will be a word from the text file and the int will be the number of times
        that word appears in the text file. */
{
   Count(String word, int i)
   {
    this.word = word;
    this.i = i;
   }
   private String word;
   private int i;
}
```


Ordering and Sorting

Order - A transitive binary relation between two objects.

Partial order - a < b and b < c => a < c.

Total order - a < b and b < a => a = b

In the integers we say that one precedes two and two precedes three is a natural order and is a total order.

Sorting - Establishes an ordering relation on a collection of objects.

Sorted Collection - A Collection of objects is structured by an ordering relation.

Java provides two interfaces for determining the ordering among a pair of object instances.

Comparable - The method compareto(object o) is implemented as a method of a class and defines a natural order among the instances of that class.

Comparator - The method compare(Object o1, Object o2) is defined in a class which implements the Comparator interface defines an ordering relational among arbitrary instances of different classes.

public interface *Comparable*

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

A class's natural ordering is said to be *consistent with equals* if and only if `(e1.compareTo((Object)e2)==0)` has the same boolean value as `e1.equals((Object)e2)` for every e1 and e2 of class C.

`public int compareTo(Object o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)
The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.
Finally, the implementer must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all z.
It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Parameters:
`o` - the Object to be compared.

Returns:
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:
ClassCastException - if the specified object's type prevents it from being compared to this Object.

public interface *Comparator*

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as TreeSet or TreeMap).

The ordering imposed by a Comparator c on a set of elements S is said to be *consistent with equals* if and only if (compare((Object)e1, (Object)e2)==0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 in S.

`public int compare(Object o1,Object o2)`

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y. (This implies that compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.)

The implementor must also ensure that the relation is transitive: ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementer must ensure that compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z.

It is generally the case, but *not* strictly required that (compare(x, y)==0) == (x.equals(y)).

Returns:

a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Throws:

ClassCastException - if the arguments' types prevent them from being compared by this Comparator.

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this Comparator. This method must obey the general contract of `Object.equals(Object)`. Additionally, this method can return true *only* if the specified Object is also a comparator and it imposes the same ordering as this comparator. Thus, comp1.equals(comp2) implies that sgn(comp1.compare(o1, o2))==sgn(comp2.compare(o1, o2)) for every object reference o1 and o2.

Note that it is *always* safe *not* to override `Object.equals(Object)`. However, overriding this method may, in some cases, improve performance by allowing programs to determine that two distinct Comparators impose the same order.

Overrides:

equals in class Object

Parameters:

obj - the reference object with which to compare.

Returns:

true only if the specified object is also a comparator and it imposes the same ordering as this comparator.

The compareTo method for Strings.

public int *compareTo*(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

The equals method for Strings

`public boolean equals(Object anObject)`

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let *k* be the smallest such index; then the string whose character at position *k* has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two string -- that is, the value: this.charAt(k)-anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings -- that is, the value:

this.length()-anotherString.length()

public class TreeMap extends AbstractMap implements SortedMap, Cloneable, Serializable

Red-Black tree based implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class (see Comparable), or by the comparator provided at creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Corman, Leiserson, and Rivest's *Introduction to Algorithms*.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be *consistent with equals* if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of *consistent with equals*.) This is so because the Map interface is defined in terms of the equals operation, but a map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal.

Constructor Summary

TreeMap()
Constructs a new, empty map, sorted according to the keys' natural order.
TreeMap(Comparator c)
Constructs a new, empty map, sorted according to the given comparator.


```
import java.util.*;
import java.io.*;
/* This program creates the output in alphabetic order by word. */
public class WordFrequency2 {
  static public void main(String[] args) {
    TreeMap words = new TreeMap();
    Count count;
    String line, word;
    try {
    String delim = " \t\n.,:;?!-/()[]\"\'";
    BufferedReader in = new BufferedReader(new FileReader(args[0]));
```



```
 while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
         word = st.nextToken().toLowerCase();
         if (words.containsKey(word)) {
          count = (Count) words.get(word);
          count.i++;
         } else {
          words.put(word, new Count(word, 1));
         }
        }
   }
  } catch (FileNotFoundException e1) {}
   catch (IOException e2) {}
```



```
    Set set = words.entrySet();
    Iterator iter = set.iterator();
    while (iter.hasNext()) {
      Map.Entry entry = (Map.Entry) iter.next();
      word = (String) entry.getKey();
      count = (Count) entry.getValue();
      System.out.println(word +  (word.length() < 8 ? "\t\t" : "\t") +  count.i);
    }
  }
  static class Count {
    Count(String word, int i) {
      this.word = word;
      this.i = i;
    }
    String word;
    int i;
  }
}
```



```
import java.util.*;
import java.io.*;
/* This program creates the output in reverse alphabetic order. */
public class WordFrequency3 {
  static public void main(String[] args) {
    Comparator NSC = (Comparator) new StringComparator();
    TreeMap words = new TreeMap(NSC);
    /* Note we are using the different constructor for TreeMap which
          the compare() method of the Comparator interface.  */
    Count count;
    String line, word;
    try {
    String delim = " \t\n.,:;?!-/()[]\"\'";
    BufferedReader in = new BufferedReader(new FileReader(args[0]));
```



```
 while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
         word = st.nextToken().toLowerCase();
         if (words.containsKey(word)) {
          count = (Count) words.get(word);
          count.i++;
         } else {
          words.put(word, new Count(word, 1));
         }
        }
   }
  } catch (FileNotFoundException e1) {}
   catch (IOException e2) {}
```

```
Set set = words.entrySet();
  Iterator iter = set.iterator();
  while (iter.hasNext()) {
    Map.Entry entry = (Map.Entry) iter.next();
    word = (String) entry.getKey();
    count = (Count) entry.getValue();
    System.out.println(word +
                       (word.length() < 8 ? "\t\t" : "\t") +
                       count.i);
  }

 }
```



```
static class Count {
  Count(String word, int i) {
    this.word = word;
    this.i = i;
  }
  String word;
  int i;
 }
}
import java.util.Comparator;
public class StringComparator implements Comparator
{       public int compare(Object o1,Object o2)
        {
         String s1 = (String) o1;
         String s2 = (String) o2;
         return - s1.compareTo(s2);
        }
}
```


public class Collections extends Object

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by sort does not have to be a mergesort, but it does have to be *stable*.)

static void sort(List list, Comparator c)
Sorts the specified list according to the order induced by the specified comparator.

public class ArrayList extends AbstractList implements List, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

ArrayList(Collection c)
Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.


```
import java.util.*;
import java.io.*;
/* This program outputs a list sorted by frequency of occurrence. */
public class WordFrequency4 {
  static public void main(String[] args) {
    Map words = new HashMap();
    String delim = " \t\n.,:;?!-/()[]\"\'";
    String line, word;
    Count count;
    try {
      BufferedReader in = new BufferedReader(new FileReader(args[0]));
      while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
          word = st.nextToken().toLowerCase();
          count = (Count) words.get(word);
          if (count == null) {
            words.put(word, new Count(word, 1));
          } else {
            count.i++;
          }
        }
      }
    }
```



```
    catch (FileNotFoundException e1) {}
    catch (IOException e2) {}
    List list = new ArrayList(words.values());
    Collections.sort(list, new CountComparator());
    Iterator iter = list.iterator();
    while (iter.hasNext()) {
      count = (Count) iter.next();
      word = count.word;
      System.out.println(word +
                         (word.length() < 8 ? "\t\t" : "\t") +
                         count.i);
    }
  }
  static class Count {
    Count(String word, int i) {
      this.word = word;
      this.i = i;
    }
    String word;
    int i;
  }
```

```
static class CountComparator implements Comparator {
  public int compare(Object o1, Object o2) {
   if (o1 != null &&
          o2 != null &&
          o1 instanceof Count &&
          o2 instanceof Count) {
         Count c1 = (Count) o1;
         Count c2 = (Count) o2;
         return (c2.i - c1.i);
   } else {
         return 0;
   }
  }
 }
}
```



```
public class TreeNode
{
        // Data Fields
        Object info;
        TreeNode left;
        TreeNode right;
        // Methods
        // Constructors
        public TreeNode() {};
        public TreeNode(Object i)
                {
                info = i;
                left = null;
                right = null;
                }
```



```
        public boolean insertNode(Object item)
        {  if (item.toString().compareTo(info.toString()) ==0)
                return false;
          else if  (item.toString().compareTo(info.toString())<0)
                // target key < root key - check left subtree
                if (left == null)
                        { left = new TreeNode(item);
                        // Insert item.
                        return true;
                        }
                else  // Insert in left subtree.
                        return left.insertNode(item);
                // target key > root key - check right subtree
                else
                if (right == null)
                        { right = new TreeNode(item);  // Insert item.
                        return true; }
                else  // Insert in right subtree.
                        return right.insertNode(item);
        }
}
```



```
public class SearchTree
{
// Data fields
        private TreeNode root;
//Methods
        public boolean insert(Object item)
        /* Inserts item into a binary search tree.
        precondition : none
        postcondition: Returns true if the insertion is performed.
        Returns false if there is a node with the same key as item. */
        { if (isEmpty())
        {       // empty search tree
                root = new TreeNode(item); // Insert item in root.
                return true;
        }
        else  // Find insertion point and insert item.
                return root.insertNode(item);
        }
        public boolean isEmpty()
        { return (root == null);
        }
```



```
public boolean search(Object target)
/* Searches for the tree item that matches target
precondition : none.
postcondition: Returns true if target's key is found otherwise, returns false.*/
        {return search(root, target);}
private boolean search(TreeNode root, Object target)
{       if (isEmpty()) return false; // Tree is empty.
        else if (target.toString().compareTo(root.info.toString())== 0)
                return true;  // Target is found.
        else if (target.toString().compareTo(root.info.toString())< 0)
        // target key < root key - check left subtree
        if (root.left == null)
                return false;  // Left subtree empty.
        else  // Search left.
                return search(root.left, target);
        // target key > root key - check right subtree
        else if  (root.right == null)
                return false; //Right subtree empty.
        else  // Search right.
                return search(root.right, target);      }
```



```
        /* Displays a binary search tree in key order.
        precondition : none
        postcondition: Displays each node in key order.         */
        public void display()
        { display(root);     }
        private void display(TreeNode root)
        {if (!isEmpty())
        {
                // Display left subtree.
                if (root.left != null)
                display(root.left);
                // Display root key.
                displayResult((root.info).toString());
                // Display right subtree.
                if (root.right != null)
                display(root.right);}
        }
}
```

```
public class TreeTest
{ public static void main (String[] args)
        { SearchTree t = new SearchTree();
        // Insert 12 names and nicknames in the tree
        t.insert("Richard");
        t.insert("Debbie");
        t.insert("Robin");
        t.insert("Chris");
        t.insert("Jacquie");
        t.insert("Jeff");
        t.insert("Dustin");
        t.insert("Richie");
        t.insert("Deborah");
        t.insert("Jeffrey");
        t.insert("Jacqueline");
        t.insert("Christopher");
        // Display the tree contents.
        t.display();
        }
}
```



```
import java.util.*;
public class TreePerformanceTest
{       public static void main (String[] args)
        {
        TreeMap t = new TreeMap();
        int n = 1;
        int i;
        Object present;
        while (n > 0 )
        {System.out.println(" Enter Number of items to be stored");
        n = SavitchIn.readLineInt();
        RandomStringGenerator RandomKey = new RandomStringGenerator(8);
         // fill the hashtable with random values
        long startTime = System.currentTimeMillis();
        for (i = 0; i < n; i++)
        {t.insert(RandomKey.generateString()); }
        long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;
        System.out.println("Store time in milliseconds = " + runTime);
        float runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Store time per item = " + runTimePerItem);
```



```
        startTime = System.currentTimeMillis();
        for (i = 0; i < n; i++)
        { present = t.search(t.root,(Object)RandomKey.generateString());}
        endTime = System.currentTimeMillis();
        runTime = endTime - startTime;
        System.out.println("Fetch time in milliseconds = " + runTime);
        runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Fetch time per item = " + runTimePerItem);
        System.out.println(" Enter Number of items to be stored");
        t.clear();}
    }

}
```



```
import java.util.*;
import corejava.*;
import structure.*;
public class HashPerformance
{    public static void main(String[] args)
  {     int n = 1;
        while (n > 0)
        {System.out.println(" Enter Number of items to be stored");
        n = SavitchIn.readLineInt();
        Hashtable a = new Hashtable();
        int i;
        Object Value;
        Object Key;
        RandomStringGenerator RandomKey = new RandomStringGenerator(6);
        RandomStringGenerator RandomValue = new RandomStringGenerator(20);
        // fill the hashtable with random values
```



```
        long startTime = System.currentTimeMillis();
        for (i = 0; i < n; i++)
        {Value = a.put((Object)RandomKey.generateString(),
                (Object) RandomValue.generateString());
        }
                long endTime = System.currentTimeMillis();
        long runTime = endTime - startTime;
        System.out.println("Store time in milliseconds = " + runTime);
        float runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Store time per item = " + runTimePerItem);
        startTime = System.currentTimeMillis();
        for (i = 0; i < a.size(); i++)
        { Value = a.get((Object)RandomKey.generateString());
        }
```



```
        endTime = System.currentTimeMillis();
        runTime = endTime - startTime;
        System.out.println("Fetch time in milliseconds = " + runTime);
        runTimePerItem = (float)runTime/(float)n;
        System.out.println(" Fetch time per item = " + runTimePerItem);
        a.clear();
        }
    }
  }
```

java.util class Stack
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.Vector
                    |
                    +--java.util.Stack

public class Stack extends Vector

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

boolean
empty()
Tests if this stack is empty.

Object
peek()
Looks at the object at the top of this stack without removing it from the stack.

Object
pop()
Removes the object at the top of this stack and returns that object as the value of this function.

Object
push(Object item)
Pushes an item onto the top of this stack.

int
search(Object o)
Returns the 1-based position where an object is on this stack.

Application of a Stack - Maze Traversal

Maze - Structure of partially connected cells with Start cell and End cell connected by at least one path. Each cell has k neighboring cells.

Maze Traversal -

1. Enter Start cell
2. Choose neighboring cell to enter.
3. Enter neighboring cell.
4. Continue to enter neighboring cells until a cell with no exits is encountered or until the End cell is encountered.
5. If a cell with no exits is encountered, go back one cell and take a different exit until either all exits have been taken and the End cell not reached or the End cell is reached.
6. Repeat 5 until either you have returned to the Start cell and exhausted all exits or you have reached the End cell.

Stack automates path reversal.