Keeping an ArrayList sorted

Mar 19

Midterm 1,a

Average: 14.5

```
public class Q1P1 {
    private int var=5;
    public Q1P1() {}
    public Q1P1(int var) { this.var = var; }
    public int getVar() {
        return var;
    }
    public void setVar(int v) {
        this.var = v;
    }
    public static void main(String[] args) {
        Q1P1 firstA = new Q1P1(22);
        Q1P1 secondA = firstA;
        secondA.setVar(17);
        Q1P1 thirdA = new Q1P1();
        System.out.println(firstA.getVar() + "  " + secondA.getVar() + "
" + thirdA.getVar());
    }}
```

Midterm 1b

```
public class Q1P2a {
    private String aString;
    private String bString;
    public Q1P2a(String aString, String bString) {
        this.aString = aString;
        bString = bString;
    }
    public String getAstring() {
        return aString;
    }
    public boolean equals(Object o) {
        if (o instanceof Q1P2a) {
            return this.aString.equals(((Q1P2a) o).getAstring());
        } else {
            return super.equals(o);
        }}
    public String toString() {
        return "A:" + aString + "\nB:" + bString;
    }

    public static void main(String[] args) {
        Q1P2a ob1 = new Q1P2a("A", "B");
        Q1P2a ob2 = new Q1P2a("B", "C");
        Q1P2a ob3 = new Q1P2a("A", "D");
        System.out.println(ob1 == ob2);
        System.out.println(ob3);
        System.out.println(ob3.equals(ob1));
    }}
```

Midterm 1c

```
public class Q1P3a {
    private class P {
        protected int ip=5;
        public P() {}
        public P(int v) { ip=v;}
        public String toString() { return "P:"+ip; }
    }
    private class R extends P {
        protected int ir=10;
        public R(int v) {
            ir = v;
            ip = ir - 10;}
        public String toString() { return "R:"+ip; }
    }
    private class S extends P {
        protected int is=20;
        public S(int v) { super(v); }
        public String toString() { return "S:"+is +" "+super.toString(); }
    }

    public void doo() {
        ArrayList<P> ph = new ArrayList<>();
        ph.add(new P(1));
        ph.add(new R(2));
        ph.add(new S(3));
        for (int i=ph.size()-1; i>=0; i--) {
            System.out.println(ph.get(i));
        }
    }
    public static void main(String[] args) {
        new Q1P3a().doo();
    }
}
```

Midterm 2

Average: 12.7

Write a method to compute the union of two arrays, both of which contain Objects of type String. (You just need to write the union method, not a whole class.) A union contains all of the elements in either the arrays, without any duplications. The signature of the method should be

```
public ArrayList<String> union(String[] array1, String[] array2)
```

The input arrays are proper sets; they contain no duplications. You should assume that there are duplications between the arrays. The returned ArrayList should contain the union.

What is the algorithmic complexity of your union method?

Midterm 3a,3b

Average: 15.25

```
public long part1(int[] arra) {
        long res = 0;
        for (int i=(arra.length*8675309); i>0; i--)
            res += arra[i % arra.length];
        return res;
    }

public long part2(int[] arra) {
        long res=1;
        for (int i=0; i<(arra.length*arra.length); i++) {
            res = res + (long)Math.sqrt(i);
        }
        return res;
    }
```

Midterm 3c,d

```
public long part3(int[] arra) {
   long res=0;
   for (int j=0; j<200; j++) {
       for (int i=2000; i<arra.length; i=i+1) {
           res += arra[i];
       }}
   return res;
}


public long part4(int[] arra) {
        long res=0;
        for (int i=0; i<arra.length; i++)
            for (int k = 0; k < 7899; k++)
                for (int m = 0; m < 2 * arra.length; m++) {
                    res += (i + k + m);
                }
        return res;
    }
```

Midterm 5

Average: 15.45

Write a complete java program that gets string input from a user, one entire line at a time. You may use Scanner for user input as follows:

```
Scanner s = new Scanner();
String line = s.nextLine();
```

With each input line, add the entire line to an ArrayList if either of the following is true

no line of the same length is already in the array list

a line with the same contents is already in the array list. Keep getting lines from the user until the user enters “99”.

After receiving “99”, stop asking the user for another line and print the contents of the ArrayList.

SAL looks a lot like ArrayList

- Should / can "extend ArrayList"
- Can — yes
- Should — almost certainly
- Why NOT?
  - I already have something working
    - lame
  - ArrayList provides lots of functions I do not want to give my users
    - reasonable (possibly)

Code for SortedArrayList

```
public void add(String stringToAdd) {
    int loc = findPlace(stringToAdd);
    insertAtLoc(stringToAdd, loc);
}

private int findPlace(String toAdd) {
    int place=0;
    while (place<size()) {
        if (toAdd.compareTo((String)get(place))<0) {
            break;
        }
        place++;
    }
    return place;
}
```

More SortedArrayList

```
private void insertAtLoc(String toAdd, int atLoc) {
    if (size()==0) {
        // use the original add function from ArrayList
        super.add(toAdd);
        return;
    }
    // Use the original Add function from arraylist
    super.add((String)get(size()-1));
    for (int ll=size()-2; ll>=atLoc; ll--) {
        set(ll+1, get(ll));
    }
    set(atLoc, toAdd);
}
```

To keep in sorted order

- Figure out where something should be put
  - O(n)
- put it there
  - O(n)
- Overall Complexity for 1 add
  - O(n) + O(n) = O(n)
- Complexity for N add
  - O(n) * O(n) = O(n^2)

SortedArrayList
further analysis

- Biggest issue is limitation to String!
- Otherwise some nice features
  - sorting would make the finding common words task from HW3 a lot easier.

Comparable Interface

- Part of Java language
- Idea, give a way for classes to define a total ordering of instances
- Java classes that implement:
  - String
  - All descendants of Number

The Comparable Interface

- `public interface` **`Comparable<T>`**
  This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred 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 as elements in a sorted set, without the need to specify a comparator.
  The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance o any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.
  It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.
  For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sor set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted does not increase) because `a` and `b` are equivalent from the sorted set's perspective.
  Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equ One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).
  For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

  ```
  {(x, y) such that x.compareTo(y) <= 0}.
  ```

Comparable interface (shortened)

`int compareTo(T 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 implementor 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))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

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.`

Comparable Interface (even shorter)

```
public interface Comparable {
int compareTo(T o);
}
```

- return 0 if they are equal
- return <0 if caller is less than compared
- return >0 if caller > compared
- new Integer(3).compareTo(4) ==> -1

Comparable Example

```
private class InsenceString implements Comparable {
        private String theString;
        public InsenceString(String strng) {theString = strng; }
        public String getString() { return theString; }
        public String toString() { return theString; }

        public int compareTo(Object o) {
            if (!(o instanceof InsenceString)) return -1;
            String s1 = theString.toLowerCase();
            String s2 = ((InsenceString) o).getString().toLowerCase();
            int l = s1.length();
            if (s2.length() < l) l = s2.length();
            for (int i = 0; i < l; i++) {
                int d = s1.charAt(i) - s2.charAt(i);
                if (d != 0) return d;
            }
            if (s1.length() == s2.length()) return 0;
            if (s1.length() > s2.length()) return 1;
            return -1;
        }    }
```

Comparable Example (pt 2)

```
    public static void main(String[] args) {
        String[] a = { "A", "B", "B", "BBB"};
        String[] b = { "B", "b", "BB", "BB" };
        new CompEx().test(a, b);
    }

    public void test(String[] ss1, String[] ss2) {
        for (int i = 0; i < ss1.length; i++) {
            InsenceString is1 = new InsenceString(ss1[i]);
            InsenceString is2 = new InsenceString(ss2[i]);
            System.out.println(is1 + " " + is2 + " " +
is1.compareTo(is2));
        }
    }
```

Re-implement SAL to use comparable

- Horrific syntax, but mostly just replace String with generic class C
- We will use the Comparable interface frequently for the rest of the semester

```
public class SalComp<C extends Comparable<C>>
      extends ArrayList<C> {

   public boolean add(C toAdd) { … }

   private int findPlace(C toAdd) { … }

   private void insertAtLoc(C toAdd,
                            int atLoc) { … }
```

In class / Lab

- Define a class MyIntPair that has two integer instance variables (i1 and i2)
- the class implements comparable such that
  - total ordering is based on the difference between i1 and i2
    - if (i1-i2) in classing class is greater than (i1-i2) in compared class then return +1
    - etc