Sorting/Searching and File I/O

- Sorting
- Searching
- Reading for this lecture: L&L 10.4-10.5

Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
  - Sorting a list of test scores in ascending numeric order
  - Sorting a list of people alphabetically by last name
- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine two specific algorithms:
  - Selection Sort
  - Insertion Sort

Selection Sort

- The approach of Selection Sort:
  - Select a value and put it in its final place in the list
  - Repeat for all other values
- In more detail:
  - Find the smallest value in the list
  - Switch it with the value in the first position
  - Find the next smallest value in the list
  - Switch it with the value in the second position
  - Repeat until all values are in their proper places

Selection Sort

- An example:

```
original:          3   9   6   1   2
smallest is 1:     1   9   6   3   2
smallest is 2:     1   2   6   3   9
smallest is 3:     1   2   3   6   9
smallest is 6:     1   2   3   6   9
```

- Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled

Swapping Two Values

- The processing of the selection sort algorithm includes the *swapping* of two values
- Swapping requires three assignment statements and a temporary storage location of the same type as the data being swapped:

```
int first = 1, second = 2;
int temp = first;
first = second;  // == 2 now
second = temp;   // == 1 now
```

Polymorphism in Sorting

- Recall that a class that implements the `Comparable` interface defines a `compareTo` method that returns the relative order of its objects
- We can use polymorphism to develop a generic sort for any set of `Comparable` objects
- The sorting method accepts as a parameter an array of `Comparable` objects
- That way, one method can be used to sort a group of `People`, or `Books`, or whatever as long as the class implements `Comparable`

Selection Sort

- The sorting method doesn't "care" what type of object it is sorting, it just needs to be able to call the `compareTo` method of that object
- That is guaranteed by using `Comparable` as the parameter type passed to the sorting method
- Each `Comparable` class has a `compareTo` method that determines what it means for one object of that class to be “less than another”
- See `PhoneList.java` (page 505)
- See `Sorting.java` (page 506), specifically the `selectionSort` method
- See `Contact.java` (page 507-508)

Insertion Sort

- The approach of Insertion Sort:
  - Pick any item and insert it into its proper place in a sorted sublist
  - Repeat until all items have been inserted
- In more detail:
  - Consider the first item to be a sorted sublist (of one item)
  - Insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new addition
  - Insert the third item into the sorted sublist (of two items), shifting items as necessary
  - Repeat until all values are inserted into their proper positions

Insertion Sort

- An example:

```
original:     3   9   6   1   2
insert 9:     3   9   6   1   2
insert 6:     3   6   9   1   2
insert 1:     1   3   6   9   2
insert 2:     1   2   3   6   9
```

- See `Sorting.java` (page 506-507), specifically the `insertionSort` method

Comparing Sorts

- The Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$

Searching

- Searching is the process of finding a target element within a group of items called the search pool
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- As we did with sorting, we'll implement the searches with polymorphic `Comparable` parameters

Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered
- See `PhoneList2.java` (page 512-513)
- See `Searching.java` (page 514-515), specifically the `linearSearch` method

Binary Search

- A *binary search* assumes the list of items in the search pool is sorted
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element of the list -- if it matches the target, the search is over
- If it doesn't, only half of the remaining elements need be searched
- Since they are sorted, the target can only be in one half of the other

Binary Search

- The process continues by comparing the target to the middle element of the remaining *viable candidates*
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or there are no remaining *viable candidates* (and the target has not been found)
- See `PhoneList2.java` (page 512-513)
- See `Searching.java` (page 514-515), specifically the `binarySearch` method

Binary Versus Linear Search

- The efficiency of binary search is good for the retrieval of data from a sorted group
- However, the group must be sorted initially
- As items are added to the group, it must be kept in sorted order
- The sorting process creates inefficiency
- If you add data to a group much more often than you search it, it may be worse to use a binary search than a linear search