Searching

See Section 5.1.3 of the text.

Searching -- looking for the index of an element in a list or array, is both easier and harder than sorting. It is easy because there are two standard methods and both are simple to implement. It is hard because if these don't apply or aren't efficient, doing something else is very difficult.

We will design our search methods to return -1 in cases when the list being searched doesn't contain the object being sought.

The first method is the obvious brute-force technique -- look through all of the elements for the one you are seeking. If you find it return its index; if you don't return -1. This is called LinearSearch:

```
public static <E extends Comparable< ? super E>>
        int linearSearch(E[] a,  E x) {
                for (int i = 0; i < a.length; i++)
                        if (x.compareTo(a[i]) == 0)
                                return i;
                return -1;
        }
```

It should be clear that the worst-case performance of this is O(n), where n is the size of the array or list being searched.

The other standard search method is called BinarySearch. This requires the list to be sorted. It plays the classic "High, Low" game -- it starts looking in the middle of the list If the value being sought is less than the value in the middle of the list, the search is continued on the left side of the list: the portion with indices less than the middle. If the value being sought is greater than the value in the middle, the search is continued on the right side of the list -- the portion with indices greater than the middle. Of course, if the element being sought is the one at the middle, we just return the index of the middle:

```
public static <E extends Comparable< ? super E>>
        int binarySearch(E[] a, E x, int first, int last) {
                while (first <= last) {
                        int mid = (first+last)/2;
                        if (x.compareTo(a[mid]) == 0)
                                return mid;
                        else if (x.compareTo(a[mid])< 0)
                                last = mid-1;
                        else
                                first = mid+1;
                }
                return -1;
        }
```

Notice that each time we do a comparison that does not find the element we cut the size of the search area in half. We can only reduce n by a factor of 2 log(n) times, so this does no more than $\log_2(n)$ comparisons and the running time of the algorithm is O(log(n)).

What do you do if you need to search a large amount of data that isn't sorted? This is the common situation in databases. If the data changes frequently, there might not be much you can do to speed up a linear search. Where the data is less dynamic, database system often build up indexing structures to allow for something like a binary search on certain keys even without the data being sorted.