# CSE 142

Sorting



## Outline for Today

- Review
  - Sequential vs Binary Search
  - Arrays
- Maintaining an Ordered List
  - Sorting



## Linear vs Binary Search

- Recall work needed to search a list of *n* items
  - Linear search ~ *n*
  - Binary search ~ log *n*
- For all but small lists, binary search is much, much, much, much (did we remember to say MUCH?) faster
  - For *n* = 1,000, log *n* ~ 10
  - For *n* = 1,000,000, log *n* ~ 20
- But we can only do binary search if the list is *sorted*
  - So how do we sort a list?



## Design Your Sorting Algorithm Here

## A Sorted List

- Choices
  - Keep list sorted at all times
    Need to make adjustments in add method
  - Sort list before searching if not done already
    Need check in contains (search) method to sort if not currently sorted
- In either case, order of items in list is usually not the order in which they were added
  - But that's presumably ok, otherwise we'd use the original unordered list implementation



## Maintaining a Sorted List

- Almost nothing in the client interface changes
  - Except we can no longer allow client to insert arbitrary objects in the middle of the list – we need to be able to compare them for order
- What kind of objects can we put in a sorted list?
  - Answer: anything that implements the standard library interface Comparable

```
/** return <0 if this < obj, return 0 if this.equals(obj), return >0 if this > obj */
public interface Comparable {
    public int compareTo(object obj);
}
```

  - Comparable is implemented by many standard classes, including String



## Maintaining a Sorted List

- Search now relies on the list being sorted, so it's crucial that we record this information in a comment

```
// instance variables
private Comparable[ ] items;  // Items in this list are stored in
private int numItems;         //   items[0] through items[numItems-1],
                              //   and the items are sorted so that items[0] <=
                              //   items[1] <= … <= items[numItems-1]
```

  - Given this comment, the implementation should keep the items sorted at all times, instead of only sorting when needed (if we wanted to do that we'd need to change the comment to reflect what we're doing)
    But we'll also look at how we could sort the entire list when needed



## Changes to Original List Class

- The code for this class looks almost exactly like the original unordered list
- One change: parameter and result types change from Object to Comparable, since only objects that provide compareTo can be kept in order

```
/** add the object to the end of the list */
public boolean add(Comparable obj) { … }
```

## Method add

- Only method from original list that needs to be changed (True?)

```
/** Add obj to this list.  Return true if successful, otherwise return false */
public boolean add(Comparable obj) {
    if (this.numItems == this.items.length) {
       return false;
    }
    // find correct location to place obj
    …
    // shift larger elements one position to the right
    …
    // place obj in correct location
    …
    numItems++;
    return true;
}
```



## Modified method add

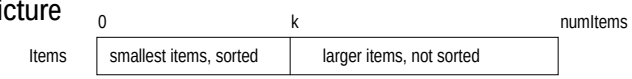
- Picture:

- Implementation details: exercise



## Selection Sort

- Larger problem: what if we want to sort an unordered list?
- Idea: At each step, pick smallest element in not-yet-sorted part of array and move it to the front
- Picture




- Detailed step (repeat until sorted)
  - Find smallest item in items[k].. items[numItems-1]
  - Swap that item with item in items [k]
  - Increase k and repeat



## Code For Selection Sort

## Code for Finding Minimum Element



## Test

- Invent some data, check the code



## Sorting on Demand

- If we didn't keep the list sorted at all times, the list class can be changed to sort the list as needed, allowing us to use binary search in contains
  - Add an instance variable to record whether the list is sorted
  - In method add, set this variable to false
  - In method contain, call the sort method if this variable is false, then do a binary search after the sort finishes
  - In method sort, set the variable to true after sorting



## Conclusion

- Performance Tradeoffs
  - Sorting is relatively expensive
  - Pays off if searches are frequent and clustered together compared to additions to the list
- Can either maintain list in sorted order at all times (expensive add operation) or sort when needed (potentially expensive lookup)
- For both algorithms, the diagrams give the key ideas
  - The code is relatively straightforward from there