CSE 142

Sorting

Outline for Today

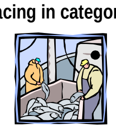
- Review
  - Sequential vs Binary Searcy
  - Arrays
- Maintaining an Ordered List
  - Sorting

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 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 in order: *sorted*
- Today's problem: how do we put a list in order?

Sorting

- In everyday life, sorting often means "placing in categories"
  - Sorting socks, sorting laundry
  - Sorting a catch of fish
  - Sorting the sheep from the goats



- In computer applications, sorting means "placing in linear order"
  - Alphabetizing a list of names
  - Listing bank account owners in order of balance size
- Like searching, sorting is generally applied to a single collection

Sorting in the Java Libraries

- We have seen that Java has methods for sorting arrays and lists.
- It works only if the elements
  - All implement the Comparable interface
  - All are in fact comparable with each other

Collections.sort(anyList);

Arrays.sort(anyArray);

What if...

- You can't use Collections or Arrays
- Or your elements are not Comparable
- Or you are working in C or C++ or some other language?
- How can you write a sort?
- How would you do it if you weren't using a computer???

Design Your Sorting Algorithm Here

Getting The List Sorted

- 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 no longer order in which added
  - But that's presumably ok – if we want really fast searches, this is a tradeoff worth making
- Terminology: a ***multiset*** or ***bag*** is like a set, but may have duplicate elements

Revised StringList Class

- StringList was an implementation of a list
- All elements were Strings
- An array was used internally to hold the elements
- Revision: maintain the elements in sorted (alphabetical) order
  - Same external interface (methods)
  - Same instance variables
  - Perhaps only one or two methods needs to change...

```
/** Ordered collection of Strings, possibly with duplicate elements */
public class StringBag { … }
```


Method add

- Revised from its implementation in the (unsorted) StringList

```
/** Add str to this StringBag.  Return true if successful, otherwise return false */
public boolean add(String str) {
    if (this.numStrings == this.strings.length) {
        return false;
    }
    // 1. find correct location to place str AND  2. shift larger elements one position to the
right
    …
    // 3. place str in correct location
    …
    numStrings++;
    return true;
}
```


Maintaining a Sorted List

- Nothing in the client interface changes
  - Except: we can no longer allow client to insert arbitrary strings in the middle of the list
    - Which method was that?
- Implementation now relies on list being sorted, so it's crucial that we record this information in a comment

```
// instance variables
private String[ ] strings;       // Strings in this StringList are stored in
private int       numStrings;    //   strings[0] through strings[numStrings-1],
                                 //   and the strings are stored in ascending
                                 //   order:  strings[0] <= strings[1] <= …
                                 //   <= strings[numStrings-1]
```


Modified method add ("insert")

- 1. Find where the new element belongs
- 2. Make room for it
- 3. Add it
- Picture:
- Notes:
- It is possible to combine steps 1 and 2.
- It is most effective to start from the <u>right</u> looking for the place to insert the new value
  - This is because you can shift values to right as you go, instead of waiting until you have found the desired position

Insertion Sort

- With an “add” (or “insert”) method that maintains order, it is easy to construct a sort
- Key observation: if a list is already sorted, adding an element gives you a longer list which is still sorted

Insertion Sort Algorithm

Algorithm: Start from a “trivially sorted array”, insert one value at a time, until all elements have been added

- Any empty array is trivially sorted
- Any empty array with just one element is trivially sorted
- It sounds like you need two arrays: an input array, and an output or result array.
- Magic trick: one array is enough! You can sort the input array in place

- Code:

```
For (int i = 0; i < array.length-1; i++) {
   Insert(array, i, array[i+1]);
   }
```


Other Sorting Algorithms

- Dozens if not hundreds of sorting algorithms exist
- We just learned “Insert Sort”
- We will now look at “Selection Sort”
- More and much better sorts in CSE143

Selection Sort

- Here’s a different algorithm for sorting an array
- Idea: At each step, pick smallest element in not-yet-sorted part of array and move it to the front
- Picture

	0	k	numStrings
strings	smallest items, sorted	larger items, not sorted	

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

Code For Selection Sort

Code for Finding Minimum Element

Testing the Code

- Invent some data, run the algorithm, check that the result is correct
- Can you write code which checks the result automatically??

Embedding in a String Collection Class

- Our original StringList class can be changed to sort the list as needed to allow binary search for *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
- Note the difference between "sorting as needed" (above) and "maintaining sorted order"

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

(c) 2001-3, University of Washington