CSCI 136
Data Structures &
Advanced Programming

Jeannie Albrecht
Lecture 11
Mar 5, 2014

Administrative Details
• Lab 4 is today
  • Extend Vector to sort with a Comparator
  • More details on next slide…
• Midterm during lab next Wednesday (3/12)
  • Can everyone start at 1:00? Should be finished by 3:00.
  • Lab 5 will still be posted next week but is optional
  • Extra credit opportunity
• Covers book, lab, lecture material through next Monday
• I will post a sample exam and a study guide

Lab 4 Details
• Lab 4
  • Extend Vector to sort with a Comparator (described in textbook)
  • Create Student class and read in data into Vector of Students
  • Create Comparators to sort students in different ways (name, phone, mailbox, etc)
  • You might want to use a ComparableAssociation at some point
    • Same as normal Association, but has compareTo
  • I was unable to update the phone book data 😞

Last Time
• Discussed searching
  • Linear searching
  • Binary searching

Today’s Outline
• Wrap up searching
• Briefly learn about Comparables and Comparators
• Discuss sorting algorithms
• High-level Goals
  • Understand sorting algorithms
  • Understand the tradeoffs between algorithms
  • Learn how to prove properties of recursive programs using induction

Recap: Binary Search
• Find a name in the phonebook
• Guess a number between 1 and 100
• These are examples of binary search
• Why does it work?
  • Rule out as much of search space as possible with each guess
• What assumption (about the data) does it rely on?
• Is it recursive? Let’s look at the code…
  • http://www.cs.williams.edu/~jeannie/cs136/lectures/lecture9/SortSearchDemo/
Binary Search

- Complexity:
  - Each recursive call takes at most 2 array comparisons
  - So how many calls when \( n = 2^k \) (list size is a power of 2...)
  - Size of list during recursion: \([2^k, 2^{k-1}, \ldots, 2^0, 0]\)
  - \( k+1 \) calls with 2 comparisons each
  - \( 2^{(k+1)} = 2(\log n + 1) \) for list of size \( n \)
- Show: \( \text{recBinarySearch} \) takes \( 2(\log n + 1) \) comparisons. (Assume list size is power of 2...)
- Overall \( O(\log n) \) comparisons
- Worst case: \( 2(\log n + 1) \)
- Can we reduce the number of comparisons?
  - Food for thought: BinSearchAlt.java

Is this version better?

- Worst case (element not found):
  - \( \log n + 1 \) comparisons vs. \( 2(\log n + 1) \) comparisons
  - Twice as good!
- Average case
  - Only slight improvement in most cases
- Best case
  - We now continue to recurse even if \( a[\text{mid}] \) is value right away
  - This is actually worse than before...

Linear vs. Binary Search

- Which is better?
  - Linear is \( O(n) \) in average case
  - Binary is \( O(\log n) \)
- So binary is better?! Yes, but...
- Binary search requires ordering (i.e., pre-sorted data). We need the \( "<" \) and \( "\geq" \) operators
  - Some objects already have these operators (ints, Strings)
  - We want a uniform way of saying objects can be compared...
  - Make an interface! (Comparable interface)

Comparable Interface

- We want to define an interface for handling comparisons between objects
  - We need a general method for \( "<" \) and \( "\geq" \) in \( \text{recBinarySearch} \)
  - As long as our \( \text{objects implement Comparable} \), we know we can safely call \( \text{compareTo} \) (required for binary search)
  - See \( \text{BinSearchComparable.java} \)

```java
public interface Comparable {
    //post: return < 0 if smaller than other
    //return 0 is equal to other
    //return > 0 if greater than other
    int compareTo(Object other);
}
```

compareTo in Card.java

```java
public class Card implements Comparable {
    //... (Code snipped for brevity)

    public int compareTo(Object obj) {
        Card other = (Card) obj;
        if (suit != other.getSuit()) {
            return suit - other.getSuit();
        }
        return rank - other.getRank();
    }
}
```

- Note: The magnitude of the values returned is not important. We only care if it’s \( +, - \), or \( 0 \! \)
- \( \text{compareTo} \) defines a natural ordering of Objects
- Can also use parameterized data types to avoid casting
- In lab this week, you’ll explore another way to compare objects using Comparators

Comparators

- Limitations with \( \text{Comparable interface} \)
  - Only permits one order between objects
  - What if it isn’t the desired ordering?
  - What if it isn’t implemented?
- Solution: Comparators
Comparators (Ch 6.8)

• A comparator is an object that contains a method that is capable of comparing two objects.
• Sorting methods can apply a comparator to two objects when a comparison is to be performed.
• Different comparators can be applied to the same data to sort in different orders or on different keys.

```java
public interface Comparator <E> {
    // pre: a and b are valid objects, likely of similar type
    // post: returns a value <, =, or > than 0
    // if a is less than, equal to, or greater than b
    public int compare(E a, E b);
}
```

Example

```java
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; age = a;}
    public String getName() { return name; }
    public int getAge() {return age;}
}

class NameComparator implements Comparator <Patient> {
    public int compare(Patient a, Patient b) {
        return a.getName().compareTo(b.getName());
    }
}
```

Different Types of Sorting

• Bubble sort
• Insertion sort
• Selection sort
• Merge sort
• Quick sort

### Bubble Sort

• Simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order.
• Repeated until no swaps are needed.
• Gets its name from the way smaller elements "bubble" to the front of the list.
• Time complexity?
  - O(n^2)
• Space complexity?
  - O(1) total (no additional space is required)

### Third Pass:

- (5, 123, 29) -> (5, 123, 29)
- (5, 123, 29) -> (123, 5, 29)
- (123, 5, 29) -> (123, 5, 29)
- (123, 5, 29) -> (123, 5, 29)

http://www.youtube.com/watch?v=lyZQPJUT5S4
Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced sorting algorithms
- Advantages:
  - Simple to implement and efficient on small lists
  - Efficient on data sets which are already substantially sorted
- Time complexity
  - $O(n^2)$
- Space complexity
  - $O(n)$