Last Time

- Assertions
- SLL Improvements
  - Tail pointers
  - Circularly Linked Lists
- Doubly Linked Lists
  - Practice with recursion on lists
Today’s Outline

• The Structure5 Universe
• Search
• The Comparable Interface
• “Basic” Sorting
  • Bubble, Insertion, Selection Sorts
• Comparator interfaces for flexible sorting
• More Efficient Sorting Algorithms
  • MergeSort, QuickSort
The Structure5 Universe (almost)
The Structure5 Universe (so far)
Search!

- What is search?
  - Locating an element among our data
- Later we will talk about data structures designed for efficient search
  - Search trees (binary, Tries, B-trees, Be-trees)
  - Hash tables
  - Dictionary interface
- But right now we have the List interface...
Leveraging Order

• I’m thinking of a number between 1 and 1,000
  • How do you guess?
  • Brute force search (linear scan) is $O(n)$ in the worst case
  • But natural numbers are ordered
• When data is sorted, binary search!
• BinarySearch.java
public class BinarySearch {

    public static int binarySearch(int a[], int value) {
        return recBinarySearch(a, value, 0, a.length-1);
    }

    protected static int recBinarySearch(int a[], int value, int low, int high) {
        if (low > high) { //value not found
            return -1;
        } else {
            int mid = (low + high) / 2; //find midpoint
            if (a[mid] == value) //found!
                return mid;
            else if (a[mid] < value) //search upper half
                return recBinarySearch(a, value, mid+1, high);
            else //search lower half
                return recBinarySearch(a, value, low, mid-1);
        }
    }
}
Recall: Binary Search

• Why does it work?
  • Because items can be ordered they can be sorted then searched based on ordering

• Why is it fast?
  • Cut search space in half with each comparison!
  • Runtime???
    • $O(\log_2(n))$  (# of times we can divide by `2` before we get `1`)

• Precondition: data is comparable and ordered

• If items are not comparable, we typically need to do a linear search
Linear Search

- Complexity analysis of linear search:
  - Best case: $O(1)$
  - Worst case: $O(n)$
  - Average case: $O(n)$
  - Why?
    - Assume all locations equally likely
    - The average number of comparisons is
      \[
      (1 + 2 + 3 + \ldots + n)/n = (n+1)/2, \text{ so } O(n)
      \]
  - Here’s a *generic* linear search method
public class LinearSearchGeneric {
    // post: returns index of value in a, or -1 if not found
    // Note the <E> between static and int: a generic method!
    public static <E> int linearSearch(E a[], E value) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String args[]) {
        // search a String array
        System.out.println(linearSearch(args, "cow"));
        // search an Integer array
        Integer odds[] = new Integer[] { 1,3,5,7,9 };;
        System.out.println(linearSearch(odds, 7));
    }
}
Linear vs. Binary Search

- Clearly binary is preferable
- But it requires ordered (i.e., sorted) data.
  - We need *comparable* items
  - Unlike with equality testing, the Object class doesn’t define a “*compare()*” method
  - We want a uniform way of saying objects can be compared, so we can write generic versions of methods like binary search
- Solution: Use an interface!
Comparable Interface

- Java provides an interface for comparisons between objects
  - Provides a replacement for “<” and “>” in recBinarySearch
- Java provides the Comparable interface, which specifies a method compareTo()
  - Any class that implements Comparable, provides compareTo()

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

- Player.java
  - Orders basketball players from shortest to tallest
  - compareTo() subtracts their heights… why?
Notes on `compareTo()`

Notes

- The magnitude of the values returned by `compareTo()` are not important.
  - We only care if the return value is positive, negative, or 0!
- `compareTo()` defines a “natural ordering” of Objects
  - There’s nothing “natural” about it….
- We can use `compareTo()` to implement sorting algorithms!
Comparable & compareTo

- The Comparable interface (Comparable<T>) is part of the java.lang (not structure5) package.
- Other Java-provided structures can take advantage of objects that implement Comparable
  - Strings, or the Arrays class in java.util
- **Note:** Users of Comparable are urged to ensure that compareTo() and equals() are *consistent*. That is,
  - x.compareTo(y) == 0 exactly when x.equals(y) == true
- **Note** that Comparable limits user to a *single ordering*
- The syntax can get kind of dense
  - See BinSearchComparable.java : a generic binary search method
  - And even more cumbersome....
ComparableAssociation

- Think back to the WordGen lab...
- Suppose we want an ordered Dictionary, so that we can use binary search instead of linear scanning
- Structure5 provides a ComparableAssociation class that implements Comparable.
- The class declaration for ComparableAssociation is
  ...wait for it...
  
  public class ComparableAssociation<K extends Comparable<K>, V> Extends Association<K,V> implements Comparable<ComparableAssociation<K,V>>
  
  (Yikes!)
- Example: Since Integer implements Comparable, we can write:
  ComparableAssociation<Integer, String> myAssoc = new ComparableAssociation(567, “Bob”);
- We could then sort an array of these!
Sorting Preview: Bubble Sort

• Simple sorting algorithm that works by ascending 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 larger elements "bubble" to the end of the list
Bubble Sort

5 1 3 2 9

• First Pass:
  • \((5 \underline{1} 3 2 9) \rightarrow (1 \underline{5} 3 2 9)\)
  • \((1 \underline{5} 3 2 9) \rightarrow (1 \underline{3} 5 2 9)\)
  • \((1 \underline{3} 5 2 9) \rightarrow (1 \underline{3} 2 5 9)\)
  • \((1 \underline{3} 2 5 9) \rightarrow (1 \underline{3} 2 5 9)\)

• Second Pass:
  • \((1 \underline{3} 2 5 9) \rightarrow (1 \underline{3} 2 5 9)\)
  • \((1 \underline{3} 2 5 9) \rightarrow (1 \underline{2} 3 5 9)\)
  • \((1 \underline{2} 3 5 9) \rightarrow (1 \underline{2} 3 5 9)\)

• Third Pass:
  • \((1 \underline{2} 3 5 9) \rightarrow (1 \underline{2} 3 5 9)\)
  • \((1 \underline{2} 3 5 9) \rightarrow (1 \underline{2} 3 5 9)\)

• Fourth Pass:
  • \((1 \underline{2} 3 5 9) \rightarrow (1 \underline{2} 3 5 9)\)

http://www.youtube.com/watch?v=lyZQPjUT5B4
Bubble Sort

• Simple sorting algorithm that works by ascending 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 larger elements "bubble" to the end of the list
• Time complexity?
  • $O(n^2)$
• Space complexity?
  • $O(n)$ total (no additional space is required)