Comparisons and the Comparable Interface

Comparison

- Something that we do a lot
- Can compare all kinds of data with respect to all kinds of comparison relations
  - Identity
  - Equality
  - Order
  - Lots of others

Identity vs. Equality

- For primitive types (e.g., int, long, float, double, boolean)
  - == and != are equality tests
- For reference types (i.e., objects)
  - == and != are identity tests
    - In other words, they test if the references indicate the same address in the Heap
- For equality of objects: use the equals() method
  - equals( ) is defined in class Object
  - Any class you create inherits equals from its parent class, but you can override it (and probably want to)

Identity vs. Equality for Strings

- Quiz: What are the results of the following tests?
  - "hello".equals("hello") true
  - "hello" == "hello" true
  - "hello" == new String("hello") false

Notions of equality

- A is equal to B if A can be substituted for B anywhere
  - Identical things must be equal: == implies equals
  - Immutable values are equal if they represent same value!
    - (new Integer(2)).equals(new Integer(2))
    - == is not an abstract operation
- Mutable values can be distinguished by assignment.
  - class Foo { int f; Foo(int g) { f = g; } }
  - Foo x = new Foo(2);
  - Foo y = new Foo(2);
  - x.equals()? Not really (x.f = 2), but Java fudges equality
- Shallow equality: x equals y if all components are ==
- Deep equality: x equals y if all components are (deep) equal

Order

- For all other reference types
  - <, >, <=, >= do not work
  - Not clear you want them to work: suppose we compare People
    - Name
    - Height
    - SSN
    - CUID?
  - Java provides Comparable interface
  - Or can use a Comparator
Comparable Interface

```java
interface Comparable {
    int compareTo(Object x);
}
```

- (Note: this is Java 1.4.2 – Java 5.0 has generics)
- `x.compareTo(y)` returns a negative, zero, or positive integer based on whether `x` is less-than, equal-to, or greater-than `y`, respectively.
- `less-than`, `equal-to`, and `greater-than` are defined for that class by the implementation of `compareTo`

Example

```java
// To compare people by weight:

class Person implements Comparable {
    private int weight;
    ...
    public int compareTo(Object obj) {
        return ((Person)obj).weight - weight;
    }
    public boolean equals(Object obj) {
        return obj instanceof Person &&
                ((Person)obj).weight == weight;
    }
}
```

Consistency

If a class has an `equals` method and also implements `Comparable`, then it is advisable (but not enforced) that

```
public boolean equals(Object obj) {
    return this.equals(obj);
}
```

exactly when

```
public int compareTo(Object obj) {
    return 0;
}
```

Odd behavior can result if this is violated

Generic Code

- The `Comparable` interface allows generic code for sorting, searching, and other operations that only require comparisons

```java
static void mergeSort(Comparable[] a) {...}
static void bubbleSort(Comparable[] a) {...}
```

Another Example

- Lexicographic comparison of `Comparable` arrays
  - for int arrays, `a < b` lexicographically iff either:
    - `a[i] == b[i] for i < j and a[j] < b[j]`, or
    - `a[i] == b[i] for all i < a.length and b is longer`

```java
//compare two Comparable arrays lexicographically
static int arrayCompare(Comparable[] a, Comparable[] b) {
    for (int i = 0; i < a.length && i < b.length; i++) {
        int x = a[i].compareTo(b[i]);
        if (x != 0) return x;
    }
    return b.length - a.length;
}
```
Comparable Interface Update

- Java 5.0 allows the use of “Generic Types”
  - Aka parameterized types
  - Here’s the Java 5.0 Comparable interface

```java
interface Comparable<T> {
    int compareTo(T x);
}
```

- compareTo is only defined for arguments of type T
  - Attempts to use a different type are caught at compile time

Example

- In the Java source code, class String looks sort of (other interfaces are also implemented) like this:

```java
public final class String implements Comparable<String>{
    public int compareTo(String s) {...}
}
```

- Code such as
  - “hello”.compareTo(new Integer(3))
  - generates a compile-time error
  - This implies that the runtime code can be more efficient

Using Comparable for Sorting

- Sorting of arrays provided by Java Collections Framework:

```java
import java.util.Arrays;
...
String[] names;
...
Arrays.sort(names)
```

- This works for arrays of type comparableType[ ] (the base type must implement the Comparable interface)
- (Class java.util.Arrays also contains sort methods for arrays of type primType[ ] for each primitive type)

Unnatural Sorting

- The ordering given by compareTo is considered to be the natural ordering for a class
- Sometimes you need to sort based on a different ordering
  - Example: we may normally sort students by CUID, but we might want to produce a list alphabetized by name

```java
interface Comparator<T> {
    int compare (T x, T y);
}
```

- Can use a Comparator (a class that implements the Comparator interface)
  - Arrays.sort(students, comparator)
  - String, for example, has a predefined Comparator:
    - String.CASE_INSENSITIVE_ORDER

Efficient Programs

- Have been talking a lot about how to make writing programs efficient
  - Interfaces, encapsulation, inheritance, type checking, recursion vs. iteration, ...
- Haven’t talked much about how to make the programs themselves run efficiently
  - How long does it take program to run?
  - Is there an efficient data structure that should be used?
  - Is there a faster algorithm?

Linear Search

- Input:
  - Unsorted array A of Comparables
  - Value v of type Comparable
- Output:
  - True if v is in array A, false otherwise
- Algorithm: examine the elements of A in some order until you either
  - Find v: return true, or
  - You have unsuccessfully examined all the elements of the array: return false
Code for Linear Search

// Linear search on possibly unsorted array
public static boolean linearSearch(Comparable[] a, Object v) {
    for (int i = 0; i < a.length; i++)
        if (a[i].compareTo(v) == 0) return true;
    return false;
}

Linear search:
7 4 6 19 3 7 8 10 32 54 67 98

Binary Search

• Input:
  • Sorted array A[0..n-1] of Comparable
  • Value v of type Comparable
• Output:
  • True if v is in array A, false otherwise
• Algorithm: similar to looking up telephone directory
  • Let m be the middle element of the array
  • If (m == v) return true
  • If (m < v) search right half of array
  • If (m > v) search left half of array

Invocation: assume array named data contains values
...... binarySearch(data, 0, data.length -1, v)......

Comparing Algorithms

• If you run binary search and linear search on a computer, you will find that binary search runs much faster than linear search
• Stating this precisely can be quite subtle
• One approach: asymptotic complexity of programs
  • Big-O analysis
• Two steps:
  • Compute running time of program
  • Running time => asymptotic running time

Running Time of an Algorithm

• In general, running time of a program such as linear search depends on many factors
  • Machine on which program is executed
    • Laptop vs. supercomputer
  • Size of input (array A)
  • Big array vs. small array
  • Values in array and value we search for
    • v is first element examined in array vs. v is not in array
• To talk precisely about running times of programs, we must specify all three factors above

Defining an Algorithm’s Running Time

1. Machine on which algorithm (i.e., program) is executed
   • Random-access Memory (RAM) model of computing
   • Measure of running time: number of operations executed
   • Other models used in CS: Turing machine, Parallel RAM model,…
2. Simplified RAM model for now:
   • Each data comparison is one operation.
   • All other operations are free.
   • Evaluate searching/sorting algorithms by estimating number of comparisons they execute
   • It can be shown that, for comparison-based searching and sorting algorithms, the total number of operations executed on RAM model is proportional to number of data comparisons executed
Defining Running Time (cont’d)

2. Dependence on size of input
   - Rather than compute a single number, we will compute a function from problem size to number of comparisons
     - E.g., \( f(n) = 32n^2 - 2n + 23 \) where \( n \) is problem size
   - Each program has its own measure of problem size
   - For searching/sorting, natural measure is size of array on which you are searching/sorting

Defining Running Time (cont’d)

3. Dependence of running time on input values
   - Consider set \( I_n \) of all possible inputs of size \( n \)
   - Find number of comparisons for each possible input in this set
   - Compute
     - Average: usually hard to compute
     - Worst-case: easier to compute
   - We will use worst-case complexity

Computing Running Times

Linear search:
   - \([7, 6, 19, 2, 6, 10, 2, 34, 2, 67, 98]\)
   - Assume array is of size \( n \).
   - Worst-case number of comparisons: \( v \) is not in array.
   - Number of comparisons = \( n \).
   - Running time of linear search: \( T_L(n) = n \)

Binary search: sorted array of size \( n \)
   - \([2, 3, 8, 18, 22, 34, 45, 56, 78]\)
   - Worst-case number of comparisons: \( v \) is not in array.
   - \( T_B(n) = \lceil \log_2(n) \rceil + 1 \)