

CSE 143

Lecture 16 (A)

Searching and Comparable

reading: 13.1 - 13.3; 10.2

slides created by Marty Stepp

<http://www.cs.washington.edu/143/>

Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
 - How many elements will it need to examine? **$O(\log N)$**
 - Can be implemented with a loop or recursively
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Diagram illustrating the search range for the value 42 in the array. The array is sorted, and the search range is defined by the minimum (min) and maximum (max) values. The current search range is from index 0 to index 16, with the middle element (mid) at index 8 (value 30). The target value 42 is highlighted in yellow.

Binary search code

```
// Returns the index of an occurrence of target in a,  
// or a negative number if the target is not found.  
// Precondition: elements of a are in sorted order  
public static int binarySearch(int[] a, int target) {  
    int min = 0;  
    int max = a.length - 1;  
  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (a[mid] < target) {  
            min = mid + 1;  
        } else if (a[mid] > target) {  
            max = mid - 1;  
        } else {  
            return mid;    // target found  
        }  
    }  
  
    return -(min + 1);    // target not found  
}
```

Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
 - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index = binarySearch(data, 42); // 10
int index2 = binarySearch(data, 66); // -14
```

Exercise solution

```
// Returns the index of an occurrence of the given value in
// the given array, or a negative number if not found.
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

// Recursive helper to implement search behavior.
private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) {
        return -1;           // target not found
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {           // too small; go right
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) {    // too large; go left
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;           // target found; a[mid] == target
        }
    }
}
```

Binary search and objects

- Can we `binarySearch` an array of `Strings`?
 - Operators like `<` and `>` do not work with `String` objects.
 - But we do think of strings as having an alphabetical ordering.
- **natural ordering**: Rules governing the relative placement of all values of a given type.
- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
 - $A < B$, $A == B$, $A > B$

The compareTo method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
 - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```
- A call of `A.compareTo(B)` will return:
 - a value < 0 if **A** comes "before" **B** in the ordering,
 - a value > 0 if **A** comes "after" **B** in the ordering,
 - or 0 if **A** and **B** are considered "equal" in the ordering.

Using compareTo

- compareTo can be used as a test in an if statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
if (a < b) { ...	if (a.compareTo(b) < 0) { ...
if (a <= b) { ...	if (a.compareTo(b) <= 0) { ...
if (a == b) { ...	if (a.compareTo(b) == 0) { ...
if (a != b) { ...	if (a.compareTo(b) != 0) { ...
if (a >= b) { ...	if (a.compareTo(b) >= 0) { ...
if (a > b) { ...	if (a.compareTo(b) > 0) { ...

Binary search w/ strings

```
// Returns the index of an occurrence of target in a,  
// or a negative number if the target is not found.  
// Precondition: elements of a are in sorted order  
public static int binarySearch(String[] a, int target) {  
    int min = 0;  
    int max = a.length - 1;  
  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (a[mid].compareTo(target) < 0) {  
            min = mid + 1;  
        } else if (a[mid].compareTo(target) > 0) {  
            max = mid - 1;  
        } else {  
            return mid;    // target found  
        }  
    }  
  
    return -(min + 1);    // target not found  
}
```

compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};  
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's `TreeSet/Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();  
for (String s : a) {  
    set.add(s);  
}  
System.out.println(s);  
// [al, bob, cari, dan, mike]
```

Ordering our own types

- We cannot binary search or make a `TreeSet/Map` of arbitrary types, because Java doesn't know how to order the elements.
 - The program compiles but crashes when we run it.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();  
tags.add(new HtmlTag("body", true));  
tags.add(new HtmlTag("b", false));  
...
```

```
Exception in thread "main" java.lang.ClassCastException  
    at java.util.TreeSet.add(TreeSet.java:238)
```

Comparable (10.2)

```
public interface Comparable<E> {  
    public int compareTo(E other);  
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.
- A call to your `compareTo` method should return:
 - a value < 0 if the `other` object comes "before" this one,
 - a value > 0 if the `other` object comes "after" this one,
 - or 0 if the `other` object is considered "equal" to this.
- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

Comparable template

```
public class name implements Comparable<name> {  
    ...  
    public int compareTo(name other) {  
        ...  
    }  
}
```

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

compareTo tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;    // different x
    } else {
        return y - other.y;    // same x; compare y
    }
}
```

– The idea:

- if $x > \text{other.x}$, then $x - \text{other.x} > 0$
- if $x < \text{other.x}$, then $x - \text{other.x} < 0$
- if $x == \text{other.x}$, then $x - \text{other.x} == 0$

– NOTE: This trick doesn't work for `doubles` (but see `Math.signum`)

compareTo tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```


Exercises

- Make the `HtmlTag` class from HTML Validator comparable.
 - Compare tags by their elements, alphabetically by name.
 - For the same element, opening tags come before closing tags.

```
// <body><b></b><i><b></b><br/></i></body>
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));    // <body>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("i", true));      // <i>
tags.add(new HtmlTag("b", true));      // <b>
tags.add(new HtmlTag("b", false));     // </b>
tags.add(new HtmlTag("br"));           // <br/>
tags.add(new HtmlTag("i", false));     // </i>
tags.add(new HtmlTag("body", false));  // </body>
System.out.println(tags);
// [<b>, </b>, <body>, </body>, <br/>, <i>, </i>]
```

Exercise solution

```
public class HtmlTag implements Comparable<HtmlTag> {
    ...
    // Compares tags by their element ("body" before "head"),
    // breaking ties with opening tags before closing tags.
    // Returns < 0 for less, 0 for equal, > 0 for greater.
    public int compareTo(HtmlTag other) {
        int compare = element.compareTo(other.getElement());
        if (compare != 0) {
            // different tags; use String's compareTo result
            return compare;
        } else {
            // same tag
            if ((isOpenTag == other.isOpenTag()) {
                return 0; // exactly the same kind of tag
            } else if (other.isOpenTag()) {
                return 1; // he=open, I=close; I am after
            } else {
                return -1; // I=open, he=close; I am before
            }
        }
    }
}
```