CSE 143
Lecture 19

Generic BSTs, Comparable interface

A Tree of Strings

- Our tree is a tree of integers. What if I wanted to make a tree of Strings?

```
public class StringTreeNode {
    public String data;              // data stored in this node
    public StringTreeNode left;      // reference to left subtree
    public StringTreeNode right;     // reference to right subtree

    // post: constructs a StringTreeNode as a leaf with given data
    public StringTreeNode(String data) {
        this(data, null, null);
    }

    // post: constructs a StringTreeNode with the given data and links
    public StringTreeNode(String data, StringTreeNode left,
                          StringTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

- What if I wanted to make a tree of People objects? BankAccount objects? Time objects?

2

SearchTreeNode (version 1)

```
public class SearchTreeNode<E> {
    public E data;                  // data stored in this node
    public SearchTreeNode<E> left;  // reference to left subtree
    public SearchTreeNode<E> right; // reference to right subtree

    // post: constructs a SearchTreeNode as a leaf with given data
    public SearchTreeNode(E data) {
        this(data, null, null);
    }

    // post: constructs a SearchTreeNode with the given data and links
    public SearchTreeNode(E data, SearchTreeNode<E> left,
                          SearchTreeNode<E> right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

3

Generalizing add

```
    // post: value is added to overall tree so as to preserve the
    //       binary search tree property
    public void add(E value) {
        overallRoot = add(overallRoot, value);
    }

    // post: value is added to given tree so as to preserve the
    //       binary search tree property
    private SearchTreeNode<E> add(SearchTreeNode<E> root, E value) {
        if (root == null) {
            root = new SearchTreeNode<E>(value);
        } else if (value <= root.data) {
            root.left = add(root.left, value);
        } else {
            root.right = add(root.right, value);
        }
        return root;
    }
```

- What do we do with the <= ?

4

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 `this` object comes "before" `other` object,
  a value > 0 if `this` object comes "after" `other` object,
  or 0 if `this` object is considered "equal" to `other`.

5

Comparable template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

6

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
        }
    }
}
```

7

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 > other.x`, then `x - other.x > 0`
    - if `x < other.x`, then `x - other.x < 0`
    - if `x == other.x`, then `x - other.x == 0`
      - NOTE: This trick doesn't work for `doubles` (see `Math.signum`)

8

Generalizing add

```
// post: value is added to overall tree so as to preserve the
//       binary search tree property
public void add(E value) {
    overallRoot = add(overallRoot, value);
}

// post: value is added to given tree so as to preserve the
//       binary search tree property
private SearchTreeNode<E> add(SearchTreeNode<E> root, E value) {
    if (root == null) {
        root = new SearchTreeNode<E>(value);
    } else if (((Comparable<E>)value).compareTo(root.data) <= 0) {
        root.left = add(root.left, value);
    } else {
        root.right = add(root.right, value);
    }
    return root;
}
```

9

SearchTreeNode

```
public class SearchTreeNode<E extends Comparable<E>> {
    public E data;                   // data stored in this node
    public SearchTreeNode<E> left;   // reference to left subtree
    public SearchTreeNode<E> right;  // reference to right subtree

    // post: constructs a SearchTreeNode as a leaf with given data
    public SearchTreeNode(E data) {
        this(data, null, null);
    }

    // post: constructs a SearchTreeNode with the given data and links
    public SearchTreeNode(E data, SearchTreeNode<E> left,
                          SearchTreeNode<E> right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

10