

LINKED STRUCTURES IN JAVA

Prof. Chris Jermaine
cmj4@cs.rice.edu

Say You Want To Implement a Container

- That can should be able to hold a variable amount of data
 - Where “variable” means not known at compile time
 - Example: want to implement the “ListWRemove” interface

```
interface ListWRemove <T extends Comparable <T>> {  
  
    // insert an item into the list  
    public void insert (T insertMe);  
  
    // remove a specific item  
    public T remove (T removeMe);  
  
    // print the list so the first item inserted is first  
    public void print ();  
}
```

Three Ways You Could Do This

- Option one: you could use an existing container
- Option two: you could use an array (since can be sized at runtime)
- Option three: can build a linked structure
 - This is what we'll consider today!

Linked List Example

```
public class ChrisList <T...> implements ListWRemove <...> {  
  
    private class Node {  
        private Node next;  
        private myData T;  
  
        private Node (T holdMe, Node nextIn) {  
            myData = holdMe; next = nextIn;  
        }  
    }  
  
    private Node myList = null;  
    ...  
}
```

Linked Structures

- Always have the sort of recursive structure in linked structs
 - By “always” I mean that they always have this recursive structure
 - Where you have a class that contains a reference to an object of its own type
 - But naturally, are infinite variations on basic idea!

How They Grow

- Can hold a variable amount of data ‘cause you can add new objects to the chain
- How to write the “insert (T insertMe)” method?

How They Grow

```
public class ChrisList <T...> implements ListWRemove <...> {
    private class Node {
        ...
        private Node (T holdMe, Node nextIn) {
            myData = holdMe; next = nextIn;
        }
    }

    private Node myList = null;

    public void insert (T insertMe) {
        myList = new Node (insertMe, myList);
    }
}
```

Why Does This Follow Recursion in the Syl?

- Because linked structures are recursive in nature
- Often easiest to write methods for them recursively
- Consider writing the “remove (T removeMe)” method...

Why Does This Follow Recursion in the Syl?

```
public class ChrisList <T..> implements ListWRemove <...> {
    ...
    private class TContainer {
        private T data = null;
        protected void addData (T addMe) {...}
    }

    private class Node {
        ...
        // removes the node containing removeMe from the list, and returns the new list
        private Node remove (T removeMe, TContainer putMeHere) {
            if (myData.compareTo (removeMe) == 0) {
                putMeHere.addData (myData);
                return next;
            } else if (next != null) {
                next = next.remove (removeMe, putMeHere);
                return this;
            } else {
                return this;
            }
        }
    }

    public T remove (T removeMe) {
        TContainer returnVal = new TContainer ();
        if (myList != null)
            myList = myList.remove (removeMe, returnVal);
        return returnVal.data;
    }
}
```

How About Printing In Order of Insertion?

- Note that the last item added is at the front of the list
- So we want to print from the back to the front

How About Printing In Order of Insertion?

```
public class ChrisList <T...> implements ListWRemove <...> {  
  
    private class Node {  
        ...  
        public void print () {  
            if (next != null) {  
                next.print ();  
            }  
            System.out.println (myData);  
        }  
    }  
  
    public void print () {  
        if (myList != null)  
            myList.print ();  
    }  
}
```

Easy, Right?

- Well, linked structures can be of arbitrary complexity
- Consider implementing the following interface

```
interface ListWFastFind <T extends Comparable <T>> {  
  
    // insert an item into the list  
    public void insert (T insertMe);  
  
    // find a specific item  
    public boolean isThere (T findMe);  
}
```

Basic BST Structure

- A common (more complex) linked structure is suitable: BST
- Each BST node has two children: left and right
- When inserting, maintain the invariant:
 - Data in root is no smaller than everything in left subtree
 - Data in root is less than everything in right subtree
- Allows fast, $\log(n)$ lookups if “balanced”
 - At every node, depth of left subtree and right subtree differs by at most a constant
- Are many flavors of BSTs

Basic BST Structure

```
public class ChrisBST <T...> implements ListWFastFind <...> {  
  
    private class Node {  
        private Node leftSubtree;  
        private Node rightSubtree;  
        private myData T;  
  
        private Node (T holdMe) {  
            myData = holdMe; leftSubtree = rightSubtree = null;  
        }  
    }  
  
    private Node root = null;  
}
```

Insertion Into a Simple BST

```
public class ChrisBST <T extends Comparable <T>> implements ... {
    ...
    private class Node {
        ...
        public void insert (T insertMe) {
            if (myData.compareTo (insertMe) >= 0) {
                if (leftSubtree == null)
                    leftSubtree = new Node (insertMe);
                else
                    leftSubtree.insert (insertMe);
            } else {
                if (rightSubtree == null)
                    rightSubtree = new Node (insertMe);
                else
                    rightSubtree.insert (insertMe);
            }
        }
    }

    private Node root = null;

    public void insert (T insertMe) {
        if (root != null) {
            root.insert (insertMe);
        } else
            root = new Node (insertMe);
    }
}
```

How To Search?

- Just start at the root, and recurse down the tree

Searching a Simple BST

```
class Node {  
    ...  
    public boolean isThere (T findMe) {  
        if (myData.compareTo (findMe) == 0) {  
            return true;  
        } else if (myData.compareTo (findMe) > 0) {  
            return (leftSubtree != null) &&  
                leftSubtree.isThere (findMe);  
        } else {  
            return (rightSubtree != null) &&  
                rightSubtree.isThere (findMe);  
        }  
    }  
}
```

- Then “isThere” for ChrisBST just returns “root != null && root.isThere ()”
- Note that this does not crash ‘cause of “short circuiting”
- That is, if the first part of an “and” evals to false, second part is ignored

So, How To Keep a BST Balanced?

- Our simple BST will only be balanced w. random insert order
- What if the insert order is not random?
 - Well, can “hash” the inserted objects then compare on hashed vals instead
 - Pros and cons?
- Many classic BST variants are “self balancing”
 - AVL trees
 - Red/black trees
 - All have intricate, challenging algorithms!
 - Will cover one in couple of lectures

Questions?