Linked Structures in Java

Prof. Chris Jermaine
cmj4@cs.rice.edu

Prof. Scott Rixner
rixner@cs.rice.edu

Say You Want To Design a Container

- That can should be able to hold a variable amount of data
  - — Where “variable” means not known at compile time
- Three ways
- First, you could use an existing container
- Or, you could use an array (since can be sized at runtime)
- Or, you can build a linked structure
  - — This is what we’ll consider today!

Linked Structures

- Are always declared like this:

```
public class LinkedList <T extends Comparable <T>> {

  private T myData;
  private LinkedList <T> next;

  public LinkedList () {
    next = null;
    myData = null;
  }
}
```

— 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 an “insert (T insertMe)” method?

How They Grow

```
public class LinkedList <T extends Comparable <T>> {

  // convention: myData in root is null
  // myData in everyone else is valid
  // makes it possible to do “direct” removals
  private T myData;
  private LinkedList <T> next;

  private LinkedList (T data, LinkedList <T> nextOne) {
    myData = data;
    next = nextOne;
  }

  public void insert (T insertMe) {
    next = new LinkedList <T> (insertMe, next);
  }
}
```

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 a “remove (T removeMe)” method...

Why Does This Follow Recursion in the Syl?

```
public class LinkedList <T extends Comparable <T>> {

  private T myData;
  private LinkedList <T> next;

  public T remove (T removeMe) {
    if (next == null) {
      return null;
    } else if (next.myData.compareTo (removeMe) == 0) {
      T returnVal = next.myData;
      next = next.next;
      return returnVal
    } else {
      return next.remove (removeMe);
    }
  }
}
```

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 LinkedList <T extends Comparable <T>> {

  private T myData;
  private LinkedList <T> next;

  // since myData is null at root, we always print
  // the list starting at “next”
  public void print () {
    if (next == null) {
      return;
    } else {
      next.print ();
      System.out.println (next.myData);
    }
  }
}
```

Easy, Right?

- Well, linked structures can be of arbitrary complexity
- A common (more complex) linked structure: 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

Insertion Into a Simple BST

```
public class BST <T extends Comparable <T>> {

  private T myData;
  private BST <T> leftSubtree;
  private BST <T> rightSubtree;

  public BST () {
    myData = null;
    leftSubtree = rightSubtree = null;
  }

  public void insert (T insertMe) {
    if (myData == null) {
      myData = insertMe;
    } else {...}
  }
}
```

Insertion Into a Simple BST

```
public class BST <T extends Comparable <T>> {

  public void insert (T insertMe) {
    if (myData == null) {
      myData = insertMe;
    } else if (myData.compareTo (insertMe) >= 0) {
      if (leftSubtree == null) {
        leftSubtree = new BST <T> ();
      }
      leftSubtree.insert (insertMe);
    } else {
      if (rightSubtree == null) {
        rightSubtree = new BST <T> ();
      }
      rightSubtree.insert (insertMe);
    }
  }
}
```

How To Search?

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

Searching a Simple BST

```
public boolean isThere (T findMe) {
  if (myData == null) {
    return false;
  } else 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);
  }
}
```

— 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! Take a look!
- Won’t cover in this class, since not that useful for our doc system
  - — Instead, we’ll cover another type of linked tree structure in depth
  - — Called a “B-Tree”
  - — Starting next time!

Questions?