LINKED STRUCTURES IN JAVA (#2)

Prof. Chris Jermaine
cmj4@cs.rice.edu

The Code We've Written is Not Very Nice

- Notice all of the null checks in our code... very error prone

```
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);
  }
}
```

- Why does our code look so nasty?

The Code We've Written is Not Very Nice

- Notice all of the null checks in our code... very error prone

```
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);
  }
}
```

- Why does our code look so nasty?
  - — We are using `null` as a proxy for a special type of node...
  - — That is empty, and sits at the end of the list...
  - — So we always need to check for the presence of that special node type

What's the Obvious Fix?

- Rather than using "if" statements to check the type of the node...
  - — And then handling the different node types accordingly
- We should have several different types of nodes
  - — And then ask them to do their work via polymorphism
  - — Same as with IDoubleVector, ISparseArray, IDoubleMatrix...

Polymorphism and Linked Structures

- Basic idea:

  — Have a hierarchy of nodes. Two concrete types... empty and occupied

```
interface IBSTNode <T extends Comparable <T>> {
  public boolean isThere (T findMe);
  public IBSTNode <T> insert (T insertMe);
}

class EmptyNode <T implements Comparable <T>> implements IBSTNode <T> {
...
}

class OccupiedNode <T implements Comparable <T>> implements IBSTNode <T> {
...
}
```

  — I know, I don't have an abstract class here (bad!)

Empty Node is Easy

— Then the code for an EmptyNode is really short and sweet

```
class EmptyNode <T implements Comparable <T>> implements IBSTNode <T> {

  public boolean isThere (T findMe) {
    return false;
  }

  public IBSTNode <T> insert (T insertMe) {
    return new OccupiedNode <T> (insertMe);
  }
}
```

isThere () for an Occupied Node

— And the OccupiedNode code is not much more complex

```
class OccupiedNode <T implements Comparable <T>> implements IBSTNode <T> {

  private T myData;
  private IBSTNode <T> left = new <T> EmptyNode ();
  private IBSTNode <T> right = new <T> EmptyNode ();

  public Occupied (T initWithMe) {
    myData = initWithMe;
  }

  public boolean isThere (T findMe) {
    if (myData.compareTo (findMe) > 0)
      return left.isThere (findMe);
    else
      return myData.compareTo (findMe) == 0 || right.isThere (findMe);
  }
  ...
}
```

insert() for an Occupied Node

— And the OccupiedNode code is not much more complex

```
class OccupiedNode <T implements Comparable <T>> implements IBSTNode <T> {

  private T myData;
  private IBSTNode <T> left = new <T> EmptyNode ();
  private IBSTNode <T> right = new <T> EmptyNode ();

  ...

  public IBSTNode <T> insert (T addMe) {
    if (myData.compareTo (findMe) >= 0)
      left = left.insert (addMe);
    else
      right = right.insert (addMe);
    return this;
  }
}
```

Now, How Does the ChrisBST Code Look?

```
class ChrisBST <.> implements <.> {

  IBSTNode <T> root = new EmptyNode <T> ();

  public void insert (T insertMe) {
    root = root.insert (insertMe);
  }

  public boolean isThere (T findMe) {
    return root.isThere (findMe);
  }
}
```

Take-Home Message

- When you are building a linked structure
  - — Do **NOT** use the value `null`
- Why?
  - — `null` is really just a placeholder for the end of the list
  - — Instead, have a node type that marks the end of the list
  - — Much less error prone. Ask an object to do its work, don't do it for the object!
- In general: polymorphism is your friend
  - — When you have nodes of different types
  - — Use polymorphism to *ask* the nodes to do their work!
  - — Never hard code the various cases

Questions?