Data Structures

Trees II

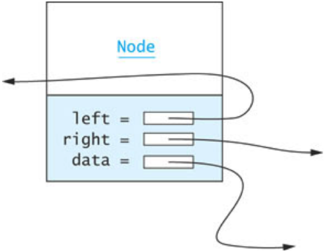
CS284

Implementing a `BinaryTree` Class

- The `Node` Class
- The `BinaryTree` Class

Node<E> Class

- Just as for a linked list, a node consists of a data part and links to successor nodes
- The data part is a reference to type E
- A binary tree node must have links to both its left and right subtrees



Node<E> Class (cont.)

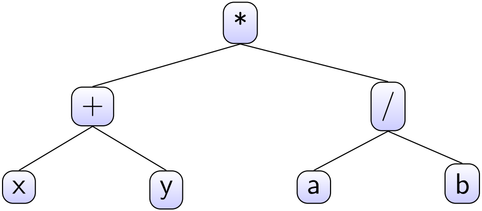
```java
protected static class Node<E> {
    protected E data;
    protected Node<E> left;
    protected Node<E> right;

    public Node(E data) {
      this.data = data;
      left = null;
      right = null;
    }
    public String toString()
      { return data.toString(); }
}
```

Node<E> Class (cont.)

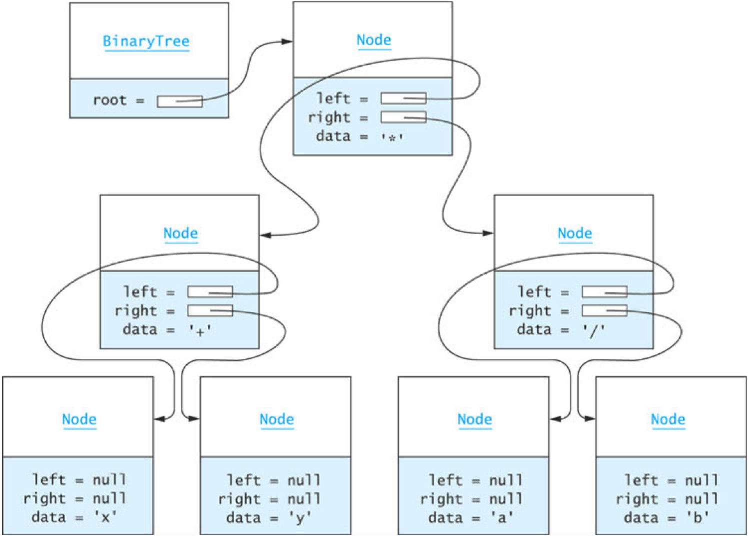
```java
protected static class Node<E> {
    protected E data;
    protected Node<E> left;
    protected Node<E> right;

    public Node(E data) {
      this.data = data;
      left = null;
      right = null;
    }
    public String toString()
      { return data.toString(); }
}
```

- The class and its data fields are declared **protected** because both `BinaryTree` and `Node` shall be subclassed later
- This way they can be accessed in the subclasses

Representation of a Binary Tree



Representation of a Binary Tree



- Implementing a `BinaryTree` Class
    - The `Node` Class
    - The `BinaryTree` Class

BinaryTree<E> Class

Data Field

```
protected Node<E> root
```

Constructor

```
public BinaryTree()
protected BinaryTree(Node<E> root)
public BinaryTree(E data, BinaryTree<E> left,
   BinaryTree<E> right)
```

Method

```
public BinaryTree<E> getLeftSubtree()
public BinaryTree<E> getRightSubtree()
public E getData()
public isLeaf()
public String toString()
private void preorderTraverse(Node<E> node, int depth,
   StringBuilder sb)
public static BinaryTree<E> readBinaryTree(Scanner scan)
```

BinaryTree<E> Class (cont.)

```
Class heading and data field declarations:

import java.io.*;

public class BinaryTree<E> implements {
// Insert inner class Node<E> here

    protected Node<E> root;

    ...
}
```

Constructors

The no-parameter constructor:

```java
public BinaryTree() {
  root = null;
}
```

The constructor that creates a tree with a given node at the root:

```java
protected BinaryTree(Node<E> root) {
  this.root = root;
}
```

- **`protected`** allows only methods in `BinaryTree` and its subclasses to use this constructor

Constructors (cont.)

The constructor that builds a tree from a data value and two trees:

```
public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E
  root = new Node<E>(data);
  if (leftTree != null) {
   root.left = leftTree.root;
  } else {
    root.left = null;
  }
  if (rightTree != null) {
    root.right = rightTree.root;
  } else {
    root.right = null;
  }
}
```

getLeftSubtree and getRightSubtree Methods

```java
public BinaryTree<E> getLeftSubtree() {
  if (root != null && root.left != null) {
    return new BinaryTree<E>(root.left);
  } else {
    return null;
  }
}
```

- getRightSubtree method is symmetric

isLeaf Method

```
public boolean isLeaf() {
   return (root == null || (root.left == null && root.right ==
}
```

- Tests whether there are any subtrees

toString Method

The toString method generates a string representing a preorder traversal in which each local root is indented a distance proportional to its depth

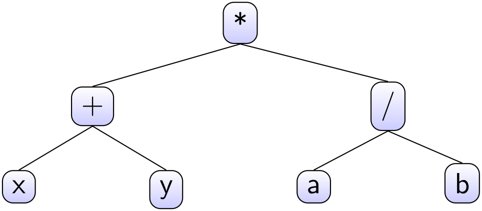
```
public String toString() {
  StringBuilder sb = new StringBuilder();
  preOrderTraverse(root, 1, sb);
  return sb.toString();
}
```

preOrderTraverse Method

```java
private void preOrderTraverse(Node<E> node, int depth, StringB

  for (int i = 1; i < depth; i++) {
    sb.append("  ");
  }
  if (node == null) {
    sb.append("null\n");
  } else {
    sb.append(node.toString());
    sb.append("\n");
    preOrderTraverse(node.left, depth + 1, sb);
    preOrderTraverse(node.right, depth + 1, sb);
  }
}
```

preOrderTraverse Method (cont.)

```
*
  +
    x
      null
      null
    y
      null
      null
  /
    a
      null
      null
    b
      null
      null
```



(x + y) * (a / b)

Reading a Binary Tree

Two step process:

- We use the class `FileReader` to open a text file
- We use the `Scanner` class to parse the text file
  - `Scanner` is a simple text scanner which can parse primitive types and strings using regular expressions.

Scanner – Example 1

```java
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Scanner – Example 2

```java
Scanner in = new Scanner(System.in);
int integer;

System.out.println("Enter an integer");

// Read in values
integer = in.nextInt();
```

Scanner – Example 3

```java
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);

while (src.hasNext()) {
  if (src.hasNextInt()) {
     i = src.nextInt();
     System.out.println("int: " + i);
  } else if (src.hasNextDouble()) {
     d = src.nextDouble();
     System.out.println("double: " + d);
  }
 else if (src.hasNextBoolean()) {
      b = src.nextBoolean();
      System.out.println("boolean: " + b);
    } else {
      str = src.next();
      System.out.println("String: " + str);
    }
  }

src.close();
```

Reading a Binary Tree

```java
public static BinaryTree<String> readBinaryTree(Scanner scan)
 String data = scan.next();
 if (data.equals("null")) {
   return null;
 } else {
    BinaryTree<String> leftTree = readBinaryTree(scan);
    BinaryTree<String> rightTree = readBinaryTree(scan);
    return new BinaryTree<String>(data, leftTree, rightTree);
 }
}
```

Text File Holding our Tree

```
*
+
x
null
null
y
null
null
/
a
null
null
b
null
null
```

Testing our Code

Place the file `Fig_6_12.txt` in your project folder (together with `bin` and `src`)

```
public class TestBinaryTree {

public static void main(String[] args) throws Exception {
  FileReader fin = new FileReader("Fig_6_12.txt");
  Scanner src = new Scanner(fin);
  BinaryTree<String> tree = BinaryTree.readBinaryTree(src);
  System.out.println(tree);
  }
}
```