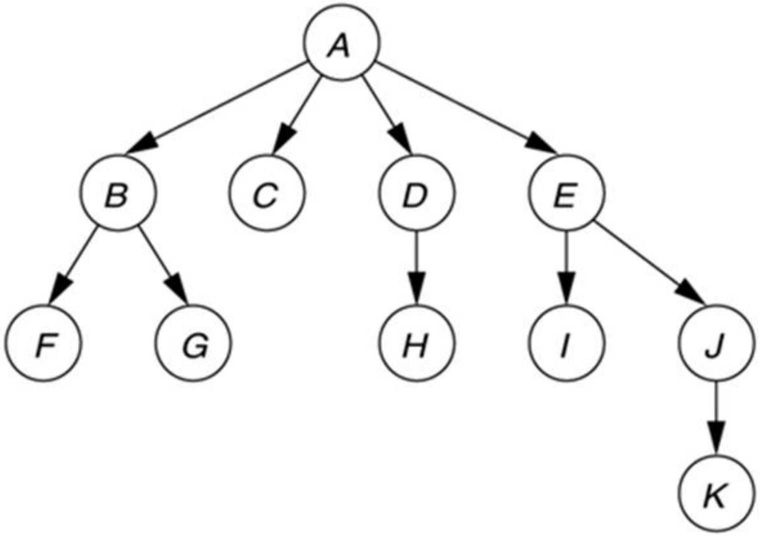


Q1

CSSE 230 Day 7

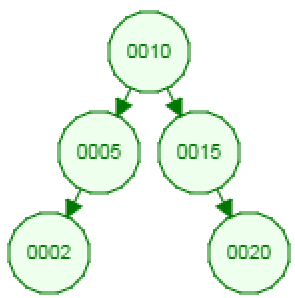
More BinaryTree methods
Tree Traversals and Iterators

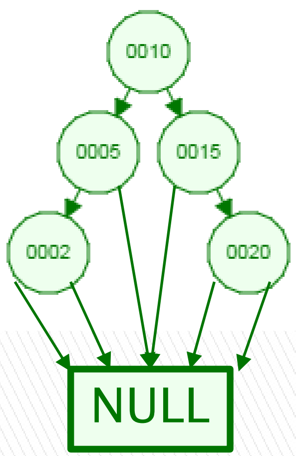
After today, you should be able to…
… traverse trees on paper & in code
… implement a simple iterator for trees

Observation about Stacks and Queues Infix → Postfix problem

- Why must I use a StringBuilder?
  - Strings are immutable. If you build your string character-by-character by using
    - s += "*"
    - It is like growing an array using the +1 scheme
  - StringBuilders have internal capacity. If you build your string character-by-character using a StringBuilder,
    - sb.append("*")
    - It is like growing an array using …
  - Not again?! ☺

A dummy NULL_NODE lets you recurse to a simpler base case while avoiding null pointer exceptions





4 possibilities for children (leaf, Left only, Right only, Both)

1 possibility for children: Both (which could be NULL_NODE)

A dummy NULL_NODE lets you recurse to a simpler base case while avoiding null pointer exceptions

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    public BinarySearchTree() {
        root = null;
    }

    public int size() {
        if (root == null) {
            return 0;
        }
        return root.size();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public int size() {
            if (left == null && right == null) {
                return 1;
            } else if (left == null) {
                return right.size() + 1;
            } else if (right == null) {
                return left.size() + 1;
            } else {
                return left.size() + right.size() + 1;
            }
        }
```

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    private final BinaryNode NULL_NODE = new BinaryNode();

    public BinarySearchTree() {
        root = NULL_NODE;
    }

    public int size() {
        return root.size();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public BinaryNode(T element) {
            this.data = element;
            this.left = NULL_NODE;
            this.right = NULL_NODE;
        }

        public int size() {
            if (this == NULL_NODE) {
                return 0;
            }
            return left.size() + right.size() + 1;
        }
```

Simpler

Simpler

More Trees

Comment out unused tests and uncomment as you go

Write `containsNonBST(T item)` now.

Notice the pattern: contains

- If (node is null)
  - Return something simple

- Recurse to the left
- Recurse to the right
- Combine results with this node

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    private final BinaryNode NULL_NODE = new BinaryNode();

    public BinarySearchTree() {
        root = NULL_NODE;
    }

    public boolean containsNonBST(T item) {
        return root.containsNonBST(item);
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public BinaryNode() {
            this.data = null;
            this.left = null;
            this.right = null;
        }

        public boolean containsNonBST(T item) {
            if (this == NULL_NODE) {
                return false;
            }
            return this.data.equals(item) ||
                   left.containsNonBST(item) ||
                   right.containsNonBST(item);
        }
    }
}
```

Notice the pattern: size

- If (node is null)
  - Return something simple
- Recurse to the left
- Recurse to the right
- Combine results with this node

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    private final BinaryNode NULL_NODE = new BinaryNode();

    public BinarySearchTree() {
        root = NULL_NODE;
    }

    public int size() {
        return root.size();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public BinaryNode() {
            this.data = null;
            this.left = null;
            this.right = null;
        }

        public int size() {
            if (this == NULL_NODE) {
                return 0;
            }
            return left.size() + right.size() + 1;
        }
    }
}
```

Notice the pattern: height

- If (node is null)
  - Return something simple

- Recurse to the left
- Recurse to the right
- Combine results with this node

```java
public class BinarySearchTree<T> {
    private BinaryNode root;

    private final BinaryNode NULL_NODE = new BinaryNode();

    public BinarySearchTree() {
        root = NULL_NODE;
    }

    public int height() {
        return root.height();
    }

    class BinaryNode {
        private T data;
        private BinaryNode left;
        private BinaryNode right;

        public BinaryNode() {
            this.data = null;
            this.left = null;
            this.right = null;
        }

        public int height() {
            if (this == NULL_NODE) {
                return -1;
            }
            return Math.max(left.height(), right.height()) + 1;
        }
    }
```

What else could you do with this recursive pattern?

- If (node is null)
  - Return something simple

- Recurse to the left
- Recurse to the right
- Combine results with this node

- Print the tree contents
- Sum the values of the nodes
- Dump the contents to an array list
- Lots more

- In what order should we print nodes?

Binary tree traversals

- InOrder (left-to-right, if tree is spread out)
  - Left, root, right
- PreOrder (top-down, depth-first)
  - root, left, right
- PostOrder (bottom-up)
  - left, right, root
- LevelOrder (breadth-first)
  - Level-by-level, left-to-right within each level

If the tree has N nodes, what's the (worst-case) big-Oh run-time of each traversal?

```java
// Print tree rooted at current node using preorder
public void printPreOrder( )  {
    System.out.println( element );          // Node
    if( left != null )
        left.printPreOrder( );              // Left
    if( right != null )
        right.printPreOrder( );             // Right
}

// Print tree rooted at current node using postorde
public void printPostOrder( ) {
    if( left != null )
        left.printPostOrder( );             // Left
    if( right != null )
        right.printPostOrder( );            // Right
    System.out.println( element );          // Node
}

// Print tree rooted at current node using inorder t
public void printInOrder( ) {
    if( left != null )
        left.printInOrder( );               // Left
    System.out.println( element );          // Node
    if( right != null )
        right.printInOrder( );              // Right
}
```

Converting the tree to an ArrayList gives an elegant solution for toString()

- Brainstorm how to write:

```
public ArrayList<T> toArrayList()
```

- Then BST toString() will simply be:

```
return toArrayList().toString();
```

Use the recursive pattern when you want to process the whole tree at once

Otherwise, you'll need a loop. Examples:

- Lazy iterators (next class):
  - use a stack too.
- AVL trees (week 4):
  - use pointer to parents to move up tree and "rebalance"
- Threaded trees:
  - A tree that uses pointer to next in-order nodes

What’s an iterator?

- In Java, specified by **`java.util.Iterator<E>`**

`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by the iterator (optional operation).

Q7-8

Binary Tree Iterators

What if we want to iterate over the elements in the nodes of the tree one-at-a-time instead of just printing all of them?