CS 310: Binary Search Trees

Chris Kauffman

Week 11-1
Logistics

Reading
- Weiss Ch. 7 Recursion
- Weiss Ch. 19 BSTs

HW 3

Today
- Wrap traversals
- Mention Tree Iterators
- BSTs and comparisons
- BST methods
Recursive Implementation of Traversals

```java
inOrder(Node t) {
    if(t != null) {
        inOrder(t.left);
        print(t.data);
        inOrder(t.right);
    }
}

preOrder(Node t) {
    if(t != null) {
        print(t.data);
        preOrder(t.left);
        preOrder(t.right);
    }
}

postOrder(Node t) {
    if(t != null) {
        postOrder(t.left);
        postOrder(t.right);
        print(t.data);
    }
}
```

Evaluate

- Correct?
- Time complexity?
- Space complexity?
- What makes this so easy?
Iterative Implementation?

TRAVERSE A TREE WITHOUT USING ITERATION?

CHALLENGE ACCEPTED
Compare to Iterative Implementation of Traversals

// Pseudo-code for post order print
void postOrder(t){
    Stack s = new Stack();
    s.push( {t, DOLEFT });
    while(!s.empty()){
        {tree, action} = s.popTop();
        if(tree == null){
            // do nothing;
        }
        else if(action == DOLEFT){
            s.push({tree, DORIGHT});
            s.push({tree.left, DOLEFT});
        }
        else if(action == DORIGHT){
            s.push({tree, DOTHIS});
            s.push({tree.right, DOLEFT});
        }
        else if(action == DOTHIS){
            print(tree.data);
        }
        else{
            throw new YouScrewedUpException();
        }
    }
}

- No call stack
- Use an explicit stack
- Auxilliary data action
  - DOLEFT work on left subtree
  - DORIGHT work on right subtree
  - DOTHIS process data for current

Evaluate
- Correct?
- Time complexity?
- Space complexity?
General Notes

Iterative Traversal Implementation Notes

- Can augment tree nodes to have a parent pointer
  ```
  class Node<T>{
    T data; Node left, right, parent;
  }
  ```
- Enables stackless, iterative traversals with great cleverness

Iterative vs Recursive Tree Methods

- Multiple types of traversals of T
- Other Tree methods: T.find(x), T.add(x), T.remove(x)
- Recursive implementations are simpler to code but will cost more memory
- Iterative methods are possible and save memory at the expense of tricky code
Weiss’s Traversals

Implemented as iterators

▶ See TestTreeIterators.java
▶ Uses BinaryTree.java and BinaryNode.java
▶ Must preserve state across advance() calls

```
BinaryTree<Integer> t = new BinaryTree<Integer>( );
... // fill tree

TreeIterator<AnyType> itr = new PreOrder<Integer>( t );
for( itr.first( ); itr.isValid( ); itr.advance( ) ){
    System.out.print( " " + itr.retrieve( ) );
}
```

▶ Much more complex to understand but good for you
▶ **Play** with some of these in JGrasp debugger if you want more practice
The Forest So Far

Traversals, heights, sizes, recursion

- Very useful
- Applicable to general trees
- Not why we started down this path

Why were we doing this again?

- What is our motivations for looking at trees again?
- e.g. Why not just stick to ArrayList/LinkedList/HashTable?
Binary Search Tree Property

A binary tree where every node \( N \) in the BST

- Any data in the tree rooted at \( N\text{.left} \) sort before \( N\text{.data} \)
- Any data in the tree rooted at \( N\text{.right} \) sort after \( N\text{.data} \)
Comparisons

How does Java guarantee comparability?

**Comparable**
Data can implement Comparable

```java
int c = x.compareTo(y);
// neg for x < y, right order
// 0 for x = y, don't care
// pos for x > y, wrong order
```

**Comparator**
Use a Comparator object to do comparisons

```java
Comparator<Thing> cmp = new ...;
int c = cmp.compare(x,y);
// neg for x < y, right order
// 0 for x = y, don't care
// pos for x > y, wrong order
```

- HW 3 assume data are Comparable (common)
- HW 4 will implement several Comparators for sorting
- Hints at a fundamental problem
Define `bst.find()`

- `find(T x)` is publicly accessible
  
  `tree.find("Mario");`

- Define
  
  `find(T x, Node<T> t)` which works on a given start node

- Recursive Version

- Iterative Version

```java
public class BinarySearchTree
<T extends Comparable<T>>
{
    protected Node<T> root;
    // Return x if in tree, null otherwise
    public T find(T x) {
        Node<T> result =
            find(x, this.root);
        if(result == null) { return null; }
        else { return result.data; }
    }

    // Find node containing x
    // starting at node t
    // Return null if not found
    private static Node<T> find(T x, Node<T> t){
        // DEFINE ME
    }
}
```
Recursive find(x,node)

Use key of data to search through tree

- Left for less than
- Right for greater than

// pseudocode
T find(x,t){
    if(t == null){
        return null;
    }
    int diff = x.compareTo(t.data);
    if(diff < 0){ // x < t
        return find(x,t.left);
    } else if(diff > 0){ // x > t
        return find(x,t.right);
    } else { // x==t
        return t; // found
    }
}
Iterative find(x, node)

See weiss/nonstandard/BinarySearchTree.java

```java
private BinaryNode<T> find(T x, BinaryNode<T> t){
    while( t != null ) {
        if( x.compareTo( t.element ) < 0 )
            t = t.left;
        else if( x.compareTo( t.element ) > 0 )
            t = t.right;
        else
            return t; // Match
    }
    return null; // Not found
}
```
Quick Note on Generics

From Weiss

```java
public class BinarySearchTree<T extends Comparable<? super T>>
```

The type

T extends Comparable<? super T>

means *descends from something Comparable* vs.

T extends Comparable<T>

comparable to self only
Break it

- What is the worst-case complexity of $\text{find}(x)$ in terms of tree properties?
- Construct a tree with this this worst case complexity
Examples of Insert

Play with `MyBST.java` in JGrasp and look at the pretty pictures after multiple `insert(x)` calls
Insertion: Similar to \texttt{find(x)}

- May need to change a left or right pointers, redefine root
- No duplication, define a TreeSet, each item unique

Define Recursive Insert

```java
class BinarySearchTree<T> {
    Node<T> root=null; int size=0;
    public void insert( T x ){
        root = insert( x, root );
    }
    protected Node<T> insert( T x, Node<T> t ){
        // DEFINE ME
    }
}
```

Define Iterative Insert

```java
public void insert( T x ){
    // DEFINE ME
}
```
Recursive insert(x,t)

From weiss/nonstandard/BinarySearchTree.java

class BinarySearchTree<T> {
    Node<T> root;
    public void insert( T x ){
        root = insert( x, root );
    }
    Node<T> insert( T x, Node<T> t )
    {
        if( t == null )
            t = new Node<T>( x );
        else if( x.compareTo( t.data ) < 0 )
            t.left = insert( x, t.left );
        else if( x.compareTo( t.data ) > 0 )
            t.right = insert( x, t.right );
        else
            throw new DuplicateItemException( x.toString( ) );
        return t;
    }
}
Iterative insert(x, t)

class BinarySearchTree<T> {
    Node<T> root;
    public void insert( T x ){
        if(this.root == null){
            this.root = new Node<T>(x);
            return;
        }
        Node<T> t = this.root;
        while(true){
            int diff = x.compareTo(t.data);
            if(diff == 0){
                throw new DuplicateItemException( x.toString() );
            }
            if(diff < 0){
                if(t.left == null){
                    t.left = new Node<T>(x);
                    return;
                }
                else{
                    t = t.left;
                }
            }
            else{
                if(t.right == null){
                    t.right = new Node<T>(x);
                    return;
                }
                else{
                    t = t.right;
                }
            }
        }
    }
}