Logistics

Reading

- Weiss Ch. 7 Recursion
- Weiss Ch 18 General Trees
- Weiss Ch 19 BSTs

Today

- Tree Traversals
- Recursive traversals
- Recursion practice for tree properties
Ordering

List property
There is a well defined ordering of first, next, last objects in the data structure,
- Wide ranging uses
- Supported in List data structure (Linked List, ArrayList)
- Supported structurally in Lists
- A property of the Data Structure

Sorting property
There is a well defined ordering relation over all possible data of a type
- "bigger than" "less than" "equal to" are well defined
- A property of the Data
- A data structure can try to mirror the data ordering structurally
- Useful for searching, walking through stored data in order
Definition is straight-forward
▶ ”Smallest” things are structurally ”first”, ”Biggest” last
▶ Ordering on elements (Comparable/Comparator)
▶ add/insert put elements in proper place

Question: For a sorted List L, what is the complexity of L.insert(x) which preserves sorting?

L is an ArrayList
How long to
▶ find insertion location?
▶ complete insertion?
▶ traverse elements in order (e.g. for printing)?

L is a LinkedList
How long to
▶ find insertion location?
▶ complete insertion?
▶ traverse elements in order (e.g. for printing)?
Alternatives to the Linear Data Structures

Hash Tables
- Abandon list property
- Abandon sorting property
- $O(1)$ insertion/retrieval
- $O(N)$ traversal, not ordered

Trees
- Abandon list property
- Preserve sorting property
- $O(\log N)$ insertion/retrieval
- $O(N)$ traversal, ordered
- Commonly Binary Trees
- Other variants
Next few sessions we’ll talk about roots

For simplicity, we’ll call them trees
Mutated Nodes

Node structures should be familiar for linked lists

- **Singly linked**: next/data
- **Doubly linked**: next/previous/data

Trees use Nodes as well

- **children**, data, possibly parent
- **Arbitrary Trees**: List<Node> of children
- **Binary Trees**: left and right children
Tree Properties of Interest

- Root of tree
- Leaves
- Data at nodes
- Size (number of nodes)
- Height of tree
- Depth of a node
class Node<T>{
    T data;
    Node<T> left, right;
}

void main(){
    Node root = new Node();
    root.data = 8;
    root.left = new Node();
    root.right = new Node();
    root.left.data = 3;
    root.right.data = 10;
    root.left.left = new Node();
    ...
}
Recursive Example: Binary Tree Size Method

```java
int size(Node<T> t)
Number of nodes in tree t

public Tree<T>{
    Node<T> root;

    // Entry point
    public int size(){
        return size(this.root);
    }

    // Recursive helper
    public static <T>
        int size(Node<T> t){
            if(t == null){
                return 0;
            }
            int sL = size(t.left);
            int sR = size(t.right);
            return 1 + sL + sR;
        }
}
```

Usage

```java
Tree<Integer> myTree = new Tree();
// add some stuff to myTree
int s = myTree.size();
```
Recursive Example: Binary Tree Height Method

Exercise

- Define a recursive `t.height()`
- `t.height()` is the longest path from root to leaf
- Empty tree has height=0

```java
int height(Node<T> t)
// Depth of deepest node in t

public Tree<T>{
    Node<T> root;
    public int height(){
        return height(this.root);
    }
    // Depth of deepest node
    public static <T>
    int height( Node<T> t ){
        // Recursive version?
    }
}
```
Recursive Implementation of `height()`

Slight difference of definitions from textbook

- Empty tree has size=0 and height=0
- 1-node tree has size=1 and height=1

```java
public Tree<T>{
    Node<T> root;
    public int height(){
        return height(this.root);
    }

    public static <T>
    int height( Node<T> t ){
        if(t == null){
            return 0;
        }
        int hL = height(t.left);
        int hR = height(t.right);
        int bigger = Math.max(hL,hR);
        return 1+bigger;
    }
}
```
The Many Ways to Walk

No linear property: several orders to traverse tree, mostly starting from the root

- (a) Pre-order traversal (this, left, right)
- (b) Post-order traversal (left, right, this)
- (c) In-order traversal (left, this, right)

Picture shows the order nodes will be visited in each type of traversal
The Many Ways to Walk

No linear property: several orders to traverse tree

**Pre-order traversal**
this, left, right

**Post-order traversal**
left, right, this

**In-order traversal**
left, this, right
Walk This Tree

Show

- (a) Pre-order traversal (this, left, right)
- (b) Post-order traversal (left, right, this)
- (c) In-order traversal (left, this, right)

Which one "sorts" the numbers?
Implementing Traversals for Binary Trees

class Tree<T>{
    private Node<T> root;

    public void printPreOrder(){
        preOrder(this.root);
    }
    private static void preOrder(Node<T> t){
        ... print(t.data) ...
    }

    public void printInOrder(){ } 
    private static void inOrder(Node<T> t){ } 

    public void printPostOrder(){ } 
    private static void postOrder(Node<T> t){ } 
}

class Node<T> {
    T data;
    Node<T> left, right;
}

Implement Print Traversals

▶ preOrder(this.root)
▶ postOrder(this.root)
▶ inOrder(this.root)

2 Ways

▶ Recursively (first)
▶ Iteratively (good luck...)
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?

TRAVERE TREE WITHOUT RECURSION?

CHALLENGE ACCEPTED
// Pseudo-code for post order print
void postOrder(root){
    Stack s = new Stack();
s.push( {root, 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?
Weiss’s Traversals

Implemented as iterators

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

```java
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 a debugger if you want more practice
General Notes

Iterative Traversal Implementation Notes

- Can augment tree nodes to have a parent pointer
  
  ```java
  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
Level-order Traversal

Level Order Traversal: 1 2 3 4 5 6 7
- Top level first (depth 1: 1)
- Then next level (depth 2: 2 3)
- etc.

This is a bit trickier
- Need an auxiliary data structure: Queue
- Does recursion help?