Prelim Updates

- Regrades are live until next Thursday @ 11:59PM
- A few rubric changes are happening
  - Recursion question: -0pts if you continued to print
  - Exception handling "write the output of execution of that statement" – rubrics change in place

Data Structures

- There are different ways of storing data, called data structures
- Each data structure has operations that it is good at and operations that it is bad at
- For any application, you want to choose a data structure that is good at the things you do often

Example Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(v)</th>
<th>get(i)</th>
<th>contains(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked List</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

add(v): append v to this list
get(i): return element at position i in this list
contains(v): return true if this list contains v

AKA add, lookup, search

Tree Overview

Tree: data structure with nodes, similar to linked list
- Each node may have zero or more successors (children)
- Each node has exactly one predecessor (parent) except the root, which has none
- All nodes are reachable from root

A tree or not a tree?

A tree

Not a tree

Not a tree

A tree

Singly linked list:

Today: trees!
Tree Terminology (1)

- The root of the tree (no parents)
- Child of M
- The leaves of the tree (no children)

Tree Terminology (2)

- Ancestors of B
- Descendants of W

Tree Terminology (3)

- Subtree of M

Tree Terminology (4)

- A node’s depth is the length of the path to the root.
- A tree’s (or subtree’s) height is the length of the longest path from the root to a leaf.

Tree Terminology (5)

- Multiple trees: a forest

Class for general tree nodes

```java
public class GTreeNode<T> {
    private T value;
    private List<GTreeNode<T>> children;
    // appropriate constructors, getters, setters, etc.
}
```

<T> means user picks a type when they create one (later lecture)

Parent contains a list of its children
Class for general tree nodes

class GTreeNode<T> {
    private T value;
    private List<GTreeNode<T>> children;
    //appropriate constructors, getters, setters, etc.
}

Java.util.List is an interface!
It defines the methods that all implementations must implement.
Whoever writes this class gets to decide what implementation to use — ArrayList? LinkedList? Etc.?

Binary Trees

A binary tree is a particularly important kind of tree in which every node has at most two children.

In a binary tree, the two children are called the left and right children.

Class for binary tree node

class TreeNode<T> {
    private T datum;
    private TreeNode<T> left, right;
    /** Constructor: one-node tree with datum d */
    public TreeNode(T d) {datum = d; left = null; right = null;}
    /** Constr: Tree with root datum d, left tree l, right tree r */
    public TreeNode(T d, TreeNode<T> l, TreeNode<T> r) {
        datum = d; left = l; right = r;
    }
    // more methods: getValue, setValue, getLeft, setLeft, etc.
}

Either might be null if the subtree is empty.

Binary versus general tree

In a binary tree, each node has up to two pointers to the left subtree and to the right subtree:
- One or both could be null, meaning the subtree is empty
  (remember, a tree is a set of nodes)

In a general tree, a node can have any number of child nodes (and they need not be ordered)
- Very useful in some situations ...
- ... one of which may be in an assignment!
A Tree is a Recursive Thing

A binary tree is either null or an object consisting of a value, a left binary tree, and a right binary tree.

Looking at trees recursively

A Recipe for Recursive Functions

Base case:
If the input is “easy,” just solve the problem directly.

Recursive case:
Get a smaller part of the input (or several parts).
Call the function on the smaller value(s).
Use the recursive result to build a solution for the full input.
A Recipe for Recursive Functions on Binary Trees

Base case: an empty tree (null), or possibly a leaf
If the input is "easy," just solve the problem directly.

Recursive case:
Get a smaller part of the input (or several parts).
Call the function on the smaller value(s), each subtree.
Use the recursive result to build a solution for the full input.

Searching in a Binary Tree

```java
/** Return true iff x is the datum in a node of tree t */
public static boolean treeSearch(T x, TreeNode<T> t) {
    if (t == null) return false;
    if (x.equals(t.datum)) return true;
    return treeSearch(x, t.left) || treeSearch(x, t.right);
}
```

• Analog of linear search in lists: given tree and an object, find out if object is stored in tree
• Easy to write recursively, harder to write iteratively

We sometimes talk of the root of the tree, t. But we also use t to denote the whole tree.

Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add (val v)</th>
<th>get (int i)</th>
<th>contains (val v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

Index set by pre-determined traversal order (see slide 36); have to go through the whole tree (no short cut like array indexing).

Node you seek could be anywhere in the tree; have to search the whole thing.

Binary Search Tree (BST)

A binary search tree is a binary tree that is ordered and has no duplicate values. In other words, for every node:

• All nodes in the left subtree have values that are less than the value in that node, and
• All values in the right subtree are greater.

A BST is the key to making search way faster.

Building a BST

To insert a new item:
- Pretend to look for the item
- Put the new node in the place where you fall off the tree

Note: Inserting them chronologically, (January, then February…) but the BST places them alphabetically (Feb comes before Jan, etc.)
Building a BST

- Insert: February
  - January

Building a BST

- Insert: March
  - January

Building a BST

- Insert: April
  - January
  - January
  - February

Printing contents of BST

```java
/** Print BST t in alpha order */
private static void print(TreeNode<T> t) {
    if (t == null) return;
    print(t.left);
    System.out.print(t.value);
    print(t.right);
}
```

Because of ordering rules for BST, easy to print alphabetically

- Recursively print left subtree
- Print the root
- Recursively print right subtree

Tree traversals

- "Walking" over the whole tree is a tree traversal
- Done often enough that there are standard names
- Previous example: inorder traversal
- Process left subtree
- Process root
- Process right subtree
- Note: Can do other processing besides printing

Other standard kinds of traversals
- preorder traversal
  - Process root
  - Process left subtree
  - Process right subtree
- postorder traversal
  - Process left subtree
  - Process right subtree
  - Process root
- level-order traversal
  - Not recursive: uses a queue
    (we’ll cover this later)
**Binary Search Tree (BST)**

Boolean `searchBST(n, v):`
- if n == null, return false
- if n.v == v, return true
- if v < n.v, return `searchBST(n.left, v)`
- else return `searchBST(n.right, v)`

**Comparing Data Structures**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>get(int i)</th>
<th>contains(val x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked List</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>BST</td>
<td>O(depth)</td>
<td>O(depth)</td>
<td>O(depth)</td>
</tr>
</tbody>
</table>

**Inserting in Alphabetical Order**

- **April**
- **August**
- **December**
- **February**
- **January**

**Insertion Order Matters**
- A balanced binary tree is one where the two subtrees of any node are about the same size.
- Searching a binary search tree takes \( O(h) \) time, where \( h \) is the height of the tree.
- In a balanced binary search tree, this is \( O(\log n) \).
- But if you insert data in sorted order, the tree becomes imbalanced, so searching is \( O(n) \).
Things to think about

What if we want to delete data from a BST?

A BST works great as long as it's balanced. There are kinds of trees that can automatically keep themselves balanced as things are inserted!