Important Announcements

- A4 is out now and due two weeks from today. Have fun, and start early!

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(val x)</th>
<th>lookup(int i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Linked List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

The Problem of Search

Example Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(val x)</th>
<th>lookup(int i)</th>
<th>search(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>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
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

Tree Terminology
- the root of the tree
  (no parents)
- child of M

Tree Terminology
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
Singly linked list:
Node object
int value
Today: trees!

Tree Terminology
- the leaves of the tree
  (no children)

Tree Terminology
- ancestors of B
- descendants of W

Tree Terminology
- subtree of M

Tree Terminology
- Depth 1, height 2.
- Depth 3, height 0.
Tree Terminology

Multiple trees: a forest.

Class for general tree nodes

```java
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 implementation must implement. Whoever writes this class gets to decide what implementation to use — ArrayList? LinkedList? Etc.?

Class for binary tree node

```java
class TreeNode<T> {
    private T value;
    private TreeNode<T> left, right;
    /** Constructor: one-node tree with datum x */
    public TreeNode (T d) { datum= d; left= null; right= null; }
    /** Constr: Tree with root value x, left tree l, right tree r */
    public TreeNode (T d, TreeNode<T> l, TreeNode<T> r) {
        datum= d; left= l; right= r; }
    }
```

Either might be null if the subtree is empty.

Class for binary tree node

```java
class TreeNode<T> {
    private T value;
    private TreeNode<T> left, right;
    /** Constructor: one-node tree with datum x */
    public TreeNode (T d) { datum= d; left= null; right= null; }
    /** Constr: Tree with root value x, left tree l, right tree r */
    public TreeNode (T d, TreeNode<T> l, TreeNode<T> r) {
        datum= d; left= l; right= r; }
        }
```

Either might be null if the subtree is empty.

Binary Trees

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

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

Class for general tree nodes

```java
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 implementation 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 where 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

```java
class TreeNode<T> {
    private T value;
    private TreeNode<T> left, right;
    /** Constructor: one-node tree with datum x */
    public TreeNode (T d) { datum= d; left= null; right= null; }
    /** Constr: Tree with root value x, left tree l, right tree r */
    public TreeNode (T d, TreeNode<T> l, TreeNode<T> r) {
        datum= d; left= l; right= r; }
    }
```

Either might be null if the subtree is empty.

Class for binary tree node

```java
class TreeNode<T> {
    private T value;
    private TreeNode<T> left, right;
    /** Constructor: one-node tree with datum x */
    public TreeNode (T d) { datum= d; left= null; right= null; }
    /** Constr: Tree with root value x, left tree l, right tree r */
    public TreeNode (T d, TreeNode<T> l, TreeNode<T> r) {
        datum= d; left= l; right= r; }
    }
```

Either might be null if the subtree is empty.

Binary Trees

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

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

Class for general tree nodes

```java
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 implementation must implement. Whoever writes this class gets to decide what implementation to use — ArrayList? LinkedList? Etc.?
**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 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 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 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 binary tree is either null or an object consisting of a value, a left binary tree, and a right binary tree.
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.

Recursive Functions on Binary Trees

**Base case:**
empty tree (null)
or, possibly, a leaf

**Recursive case:**
Call the function on 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

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);
}
```

VERY IMPORTANT!
We sometimes talk of t as the root of the tree.
But we also use t to denote the whole tree.

Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>add(ceil x)</th>
<th>lookup(ceil x)</th>
<th>search(ceil 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(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>

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

```
january
```

```
january
```

```
january

  february
```

```
january

  february
```

```
january
```

```
january

  february
```
Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february

Building a BST

january  march
february
Building a BST

```
january
february
march
april
may
june
july
august
september
october
november
```

Printing contents of BST

```
Because of ordering rules for a BST, it’s easy to print the items in alphabetical order:
- Recursively print left subtree
- Print the node
- Recursively print right subtree

```java
private static void print(TreeNode<T> t) {
    if (t == null) return;
    print(t.left);
    System.out.print(t.value);
    print(t.right);
}
```

Tree traversals

"Walking" over the whole tree is a tree traversal:
- Done often enough that there are standard names
- Previous example: in-order 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)

Compare binary tree to binary search tree:
```java
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)
```

```
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>search(n)</th>
<th>lookup(i)</th>
<th>search(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(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>
<tr>
<td>BST</td>
<td>(O(depth))</td>
<td>(O(depth))</td>
<td>(O(depth))</td>
</tr>
</tbody>
</table>
```

Inserting in Alphabetical Order

```
``
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 you insert things!

Useful facts about binary trees

Max # of nodes at depth $d$: $2^d$

If height of tree is $h$:
- min # of nodes: $h + 1$
- max # of nodes in tree:
  $n_0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree
- All levels of tree down to a certain depth are completely filled

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Height 2, minimum number of nodes

Height 2, maximum number of nodes