TREES
Important Announcements

- A4 is out! Due two weeks from today. Follow the timetable and enjoy a stress-free A4 experience!

- Mid-semester TA evaluations are open; please participate!
  - Your feedback can help our staff improve YOUR experience for the rest of this semester.

- Next week’s recitation is canceled!
  - All Tuesday sections will be office hours instead (held in same room as recitation unless noted on Piazza)
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

**Binary tree**: tree in which each node can have at most two children: a left child and a right child

- General tree
- Binary tree
- Not a tree
- List-like tree
Binary trees were in A1!

You have seen a binary tree in A1.

A PhD object phd has one or two advisors. Here is an intellectual ancestral tree!

```
     phd
    /   \
   ad1   ad2
  /     /  \
ad1  ad2 ad1
```
Tree terminology

$M$: root of this tree
$G$: root of the left subtree of $M$
$B$, $H$, $J$, $N$, $S$: leaves (they have no children)
$N$: left child of $P$; $S$: right child of $P$
$P$: parent of $N$
$M$ and $G$: ancestors of $D$
$P$, $N$, $S$: descendants of $W$
$J$ is at depth 2 (i.e. length of path from root = no. of edges)
The subtree rooted at $W$ has height (i.e. length of longest path to a leaf) of 2
A collection of several trees is called a ...?
```java
class TreeNode<T> {
    private T datum;
    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;
    }
}
```

Points to left subtree (null if empty)

Points to right subtree (null if empty)

more methods: getValue, setValue, getLeft, setLeft, 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!
Class for general tree nodes

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

Parent contains a list of its children
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.?
Applications of Tree: Syntax Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is *implicit* in ordinary textual representation
- Recursive structure can be made *explicit* by representing sentences in the language as trees: *Abstract Syntax Trees* (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A *parser* converts textual representations to AST
Applications of Tree: Syntax Trees

In textual representation:
Parentheses show hierarchical structure

In tree representation:
Hierarchy is explicit in the structure of the tree

We’ll talk more about expressions and trees in next lecture
Got that?

\[(1 + (9 - 2)) \times 7\]

- \(7\) is the root.
- \(1\) is a leaf node.
- \(9\) is at depth 3.
- \(9\)'s parent is \(-\).
- \(1\), \(+\), and \(7\) are ancestors of \(1\).
- The tree's height is 4.
- The root is 7.
Recursion on trees

Trees are defined recursively:

A binary tree is either

(1) empty

or

(2) a value (called the root value),

a left binary tree, and a right binary tree
Recursion on trees

Trees are defined recursively, so recursive methods can be written to process trees in an obvious way.

Base case
- empty tree (null)
- leaf

Recursive case
- solve problem on each subtree
- put solutions together to get solution for full tree
Class for binary tree nodes

class BinTreeNode<T> {
    private T datum;
    private BinTreeNode<T> left;
    private BinTreeNode<T> right;
    //appropriate constructors, getters, setters, etc.
}

Binary tree
Looking at trees recursively

Binary tree

```
    2
   / \
  9   0
 / \ / \    
8  3 5  7
```
Looking at trees recursively

- value
  - left subtree
  - right subtree
/** 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
/** 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.
Some useful methods — what do they do?

/** Method A ??? */
public static boolean A(Node n) {
    return n != null && n.left == null && n.right == null;
}

/** Method B ??? */
public static int B(Node n) {
    if (n == null) return -1;
    return 1 + Math.max(B(n.left), B(n.right));
}

/** Method C ??? */
public static int C(Node n) {
    if (n == null) return 0;
    return 1 + C(n.left) + C(n.right);
}
/** Return true iff node n is a leaf */
public static boolean isLeaf(Node n) {
    return n != null && n.left == null && n.right == null;
}

/** Return height of node n (postorder traversal) */
public static int height(Node n) {
    if (n == null) return -1; //empty tree
    return 1 + Math.max(height(n.left), height(n.right));
}

/** Return number of nodes in n (postorder traversal) */
public static int numNodes(Node n) {
    if (n == null) return 0;
    return 1 + numNodes(n.left) + numNodes(n.right);
}
Binary Search Tree (BST)

If the tree data is ordered and has no duplicate values:
in every subtree,
  All left descendents of a node come before the node
  All right descendents of a node come after the node
Search can be made MUCH faster
Binary Search Tree (BST)

If the tree data is ordered and has no duplicate values:
- in every subtree,
  - All left descendants of a node come before the node
  - All right descendants of a node come after the node

Search can be made MUCH faster

Compare binary tree to binary search tree:

```java
boolean searchBST(n, v):
    if n==null, return false
    if n.v == v, return true
    return searchBST(n.left, v) || searchBST(n.right, v)
```

```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)
```

2 recursive calls 1 recursive call
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
- This can be done using either recursion or iteration
- Example
  - Tree uses alphabetical order
  - Months appear for insertion in calendar order
What can go wrong?

A BST makes searches very fast, unless...

- Nodes are inserted in increasing order
- In this case, we’re basically building a linked list (with some extra wasted space for the left fields, which aren’t being used)

BST works great if data arrives in random order
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
/** Print BST t in alpha order */
private static void print(TreeNode<T> t) {
    if (t == null) return;
    print(t.left);
    System.out.print(t.datum);
    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)
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:
- \(2^0 + \ldots + 2^h = 2^{h+1} - 1\)

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

\[
\text{Depth}
\begin{array}{c}
0 \quad \text{------} \\
1 \quad \text{------} \\
2 \quad \text{------} \\
\end{array}
\]

Height 2, maximum number of nodes

\[
\begin{array}{c}
5 \\
4 \quad 2 \\
7 \quad 8 \quad 0 \quad 4 \\
\end{array}
\]

Height 2, minimum number of nodes

\[
\begin{array}{c}
5 \\
2 \\
4 \\
\end{array}
\]
Things to think about

What if we want to delete data from a BST?

A BST works great as long as it’s balanced

How can we keep it balanced? This turns out to be hard enough to motivate us to create other kinds of trees
A tree is a recursive data structure

- Each node has 0 or more successors (children)
- Each node except the root has exactly one predecessor (parent)
- All nodes are reachable from the root
- A node with no children (or empty children) is called a leaf

**Special case: binary tree**

- Binary tree nodes have a left and a right child
- Either or both children can be empty (null)

Trees are useful in many situations, including exposing the recursive structure of natural language and computer programs