Suppose you use trees to represent student schedules. For each student there would be a general tree with a root node containing student name and ID. The inner nodes in the tree represent courses, and the leaves represent the times/places where each course meets. Given two such trees, how could you determine whether and where the two students might run into one-another?

Tree Overview

- Tree: recursive data structure (similar to 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

Binary Trees were in A1!

You have seen a binary tree in A1.

A Bee object has a mom and pop. There is an ancestral tree!

Class for Binary Tree Node

```java
class TreeNode<T> {
    private T datum;
    private TreeNode<T> left, right;

    /** Constructor: one node tree with datum x */
    public TreeNode(T x) { datum= x; }

    /** Constr: Tree with root value x, left tree lift, right tree rgt */
    public TreeNode(T x, TreeNode<T> lift, TreeNode<T> rgt) {
        datum= x; left= lift; right= rgt;
    }
}
```

Tree Terminology

- M: root of this tree
- G: root of the left subtree of M
- B, H, J, N, S: leaves
- N: left child of P; S: right child
- 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)
- W is at height 2 (i.e. length of longest path to a leaf)
- A collection of several trees is called a ...?
Binary versus general tree

In a binary tree each node has exactly two pointers: to the left subtree and to the right subtree
- Of course one or both could be null

In a general tree, a node can have any number of child nodes
- Very useful in some situations ...
- ... one of which will be our assignments!

Class for General Tree nodes

class GTreeNode {
  private Object datum;
  private GTreeCell left;
  private GTreeCell sibling;
  appropriate getters/setters
}

Applications of 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

Example

Expression grammar:
- E → integer
- E → (E + E)

In textual representation
- Parentheses show hierarchical structure

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

Searching in a Binary Tree

// Return true iff x is the datum in a node of tree t
public static boolean treeSearch(Object x, TreeNode t) {
  if (t == null) return false;
  if (t.datum.equals(x)) 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

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

Important point about t. We can think of it either as
(1) One node of the tree OR
(2) The subtree that is rooted at t

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

Tree Traversals

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

What Can Go Wrong?

- A BST makes searches very fast, unless...
  - Nodes are inserted in alphabetical order
  - In this case, we're basically building a linked list (with some extra wasted space for the left fields that aren't being used)
- BST works great if data arrives in random order

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

Binary Search Tree (BST)

If the tree data are ordered, in every subtree,
All left descendents of node come before node
All right descendents of node come after node
Search is MUCH faster

** Return true if x is the datum in a node of tree t */
public static boolean treeSearch(Object x, TreeNode t) {
  if (t == null) return false;
  if (t.datum.equals(x)) return true;
  if (t.datum.compareTo(x) > 0)
    return treeSearch(x, t.left);
  else
    return treeSearch(x, t.right);
}
**Some Useful Methods**

```java
/** Return true iff node t is a leaf */
public static boolean isLeaf(TreeNode t) {
    return t != null && t.left == null && t.right == null;
}

/** Return height of node t using postorder traversal */
public static int height(TreeNode t) {
    if (t == null) return -1; // empty tree
    if (isLeaf(t)) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}

/** Return number of nodes in t using postorder traversal */
public static int nNodes(TreeNode t) {
    if (t == null) return 0;
    return 1 + nNodes(t.left) + nNodes(t.right);
}
```

**Useful Facts about Binary Trees**

- **Max number of nodes at depth** $d$: $2^d$
- If height of tree is $h$
  - **Min number of nodes in tree**: $h + 1$
  - **Max number of nodes in tree**: $2^h + \ldots + 2 = 2^{h+1} - 1$
- Complete binary tree
  - All levels of tree down to a certain depth are completely filled

**Tree with Parent Pointers**

- In some applications, it is useful to have trees in which nodes can reference their parents
- Analog of doubly-linked lists

**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

**Suffix Trees**

- Given a string $s$, a suffix tree for $s$ is a tree such that
  - each edge has a unique label, which is a nonnull substring of $s$
  - any two edges out of the same node have labels beginning with different characters
  - the labels along any path from the root to a leaf concatenate together to give a suffix of $s$
  - all suffixes are represented by some path
  - the leaf of the path is labeled with the index of the first character of the suffix in $s$

  - Suffix trees can be constructed in linear time
Suffix Trees

- Useful in string matching algorithms (e.g., longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)

Decision Trees

- Classification:
  - Attributes (e.g., is CC used more than 200 miles from home?)
  - Values (e.g., yes/no)
  - Follow branch of tree based on value of attribute.
  - Leaves provide decision.

Huffman Trees

- Fixed length encoding
  - $197^2 + 63^2 + 40^2 + 26^2 = 652$
- Huffman encoding
  - $197^1 + 63^2 + 40^3 + 26^3 = 521$

Huffman Compression of “Ulysses”

<table>
<thead>
<tr>
<th>Character</th>
<th>Decimal</th>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>242125</td>
<td>00100000</td>
<td>3</td>
</tr>
<tr>
<td>'e'</td>
<td>139496</td>
<td>01100100</td>
<td>4</td>
</tr>
<tr>
<td>'t'</td>
<td>89561</td>
<td>01110001</td>
<td>4</td>
</tr>
<tr>
<td>'a'</td>
<td>88584</td>
<td>01101111</td>
<td>4</td>
</tr>
<tr>
<td>'n'</td>
<td>78465</td>
<td>01101110</td>
<td>4</td>
</tr>
<tr>
<td>'i'</td>
<td>76505</td>
<td>01101101</td>
<td>4</td>
</tr>
<tr>
<td>'s'</td>
<td>73186</td>
<td>01101001</td>
<td>4</td>
</tr>
<tr>
<td>'h'</td>
<td>68625</td>
<td>01101000</td>
<td>5</td>
</tr>
<tr>
<td>'r'</td>
<td>68320</td>
<td>01101001</td>
<td>5</td>
</tr>
<tr>
<td>'l'</td>
<td>52657</td>
<td>01101100</td>
<td>5</td>
</tr>
<tr>
<td>'u'</td>
<td>32942</td>
<td>01101101</td>
<td>5</td>
</tr>
<tr>
<td>'g'</td>
<td>26201</td>
<td>01101110</td>
<td>6</td>
</tr>
<tr>
<td>'f'</td>
<td>25248</td>
<td>01101110</td>
<td>6</td>
</tr>
<tr>
<td>'.'</td>
<td>21361</td>
<td>01101110</td>
<td>6</td>
</tr>
<tr>
<td>'p'</td>
<td>20661</td>
<td>01110000</td>
<td>6</td>
</tr>
</tbody>
</table>

Huffman Compression of “Ulysses”

- Original size: 11904320
- Compressed size: 6822151
- 42.7% compression

BSP Trees

- BSP = Binary Space Partition (not related to BST!
- Used to render 3D images composed of polygons
- Each node n has one polygon p as data
- Left subtree of n contains all polygons on one side of p
- Right subtree of n contains all polygons on the other side of p
- Order of traversal determines occlusion (hiding)
A tree is a recursive data structure
- Each cell has 0 or more successors (children)
- Each cell except the root has at least one predecessor (parent)
- All cells are reachable from the root
- A cell with no children is called a leaf

Special case: Binary tree
- Binary tree cells have a left and a right child
- Either or both children can be null

Trees are useful for exposing the recursive structure of natural language and computer programs