CS61B Summer 2006
Instructor: Erin Korber
Lecture 11, 13 July

Trees

1 What is a tree?

A *tree* consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the property that there is exactly one path (no more, no less) between any two nodes of the tree. A *path* is a connected sequence of zero or more edges.

In a *rooted* tree (which is what we will generally talk about, and we will use the word "tree" to mean "rooted tree", generally), one distinguished node is called the *root*. Every node c, except the root, has exactly one *parent* node p, which is the first node traversed on the path from c to the root. c is p's *child*. The root has no parent. A node can have any number of children.

Other definitions:

- A *leaf* is a node with no children.
- *Siblings* are nodes with the same parent.
- The *ancestors* of a node d are the nodes on the path from d to the root. These include d's parent, d's parent's parent, d's parent's parent's parent, and so forth up to the root. The ancestors of d also include d itself (since d is on the path). The root is an ancestor of every node in the tree.
- If a is an ancestor of d, then d is a *descendant* of a.
- The *length* of a path is the number of edges in the path.
- The *depth* of a node n is the length of the path from n to the root. (The depth of the root is zero.)
- The *height* of a node n is the length of the path from n to its deepest descendant. (The height of a leaf node is zero.)
- The height of a tree is the depth of its deepest node = height of the root.

- The *subtree* rooted at node n is the tree formed by n and its descendants.
- A *binary tree* is a tree in which no node has more than two children, and every child is distinguished as either a *left child* or a *right child*, even if it's the only child its parent has.

2 Representing Trees

There are several different ways that trees can be represented. Here, we will discuss three - one where each node keeps track of the list of its children, one where each node tracks its next sibling, and one for binary trees.

2.1 Child-list trees

In this representation, each node contains a piece of data and a reference to the list of its child nodes. The directory structure of a filesystem is typically stored this way (although the lists they use are quite different from the ones we have seen so far).

```
public class LTreeNode {
    Object item;    //The data for this node.
    SList children; //The list of its child nodes.
}
```

2.2 Sibling trees

This representation eliminates the separate linked lists and instead directly links the siblings. Instead of referencing a list of children, each node references its leftmost child, and the sibling to its right. Essentially, the sibling references join the children of a node in a singly-linked list, whose head is the node's first child.

```
public class SibTreeNode {
    Object item;    //The data for this node.
    SibTreeNode firstChild;
    SibTreeNode nextSibling;
}
```

2.3 Binary trees

A binary tree is one where each node has at most two children, and every child node is designated as being a left or right child. So a node will reference both of its children.

```
public class BinTreeNode {
   Object item;
   BinTreeNode leftChild;
   BinTreeNode rightChild;
}
```

3 Tree Traversals

A *traversal* is a manner of visiting each node in a tree once. What you do when visiting any particular node depends on the application; for instance, you might print a node's value, or perform some calculation upon it. There are several different traversals, each of which orders the nodes differently. Which one you will want to use will depend on the particular application.

3.1 Preorder traversal

In a preorder traversal, you visit each node before recursively visiting its children, which are visited from left to right. The root is visited first. (What "visit" is will depend on the application - we'll just use printing the value as an example.)

```
public class SibTreeNode {
  //previous stuff here

  public void preorder() {
    this.visit();
    if (firstChild != null) {
      firstChild.preorder();
    }
    if (nextSibling != null) {
      nextSibling.preorder();
    }
  }
```

```
  public void visit() {
     System.out.println(item.toString());
  }
}
```

A preorder traversal is a natural way to print a directory's structure:

```
~ekorber/61b
   hw
      hw1
      hw2
   index.html
   lab
      lab1
      lab2
   lec
      01
      02
      03
      04
      05
```

3.2 Postorder traversal

In a *postorder* traversal, you visit each node's children (in left-to-right order) before the node itself.

```
public class SibTreeNode {
  //previous stuff here

  public void postorder() {
    if (firstChild != null) {
      firstChild.postorder();
    }
    this.visit();
    if (nextSibling != null) {
      nextSibling.postorder();
    }
  }
}
```

The `postorder()` code is trickier than it looks. The best way to understand it is to draw a depth-two tree on paper, then pretend you're the computer and execute the algorithm carefully. Trust me on this.

A postorder traversal is the natural way to sum the total disk space used in the root directory and its descendants. In the example above, a postorder traversal would begin by summing the sizes of the files in hw1/ and hw2/; then it would visit hw/ and sum its two children. The file index.html would follow, then the directories lab1/, lab2/, and lab/, and so on.

3.3 Inorder traversal

Binary trees allow for an *inorder* traversal: visit a node's left child (left subtree), then the node itself, then the node's right child (right subtree). Note that the idea of an inorder traversal does not make sense for a general tree (not necessarily binary). The preorder, inorder, and postorder traversals of an expression tree will print a prefix, infix, or postfix expression, respectively.

```
public class BinTreeNode {
   //previous stuff here


   public void inOrder() {
      if (lefChild != null) {
         leftChild.inOrder();
      }
      this.visit();
      if (rightChild != null) {
         rightChild.inOrder();
      }
   }

  public void visit() {
      System.out.print(item.toString() + ‘‘ ‘‘);
  }

}
```

Inorder traversal is a natural way to print an expression parsing tree with infix notation. (Preorder would give us prefix notation (like Scheme uses!) and postorder would give us postfix (Polish) notation.)