CIS 121

Introduction to Trees
Tree ADT

- Tree definition
  - A tree is a set of nodes which may be empty
  - If not empty, then there is a distinguished node $r$, called root and zero or more non-empty subtrees $T_1, T_2, \ldots T_k$, each of whose roots are connected by a directed edge from $r$.

- This recursive definition leads to recursive tree algorithms and tree properties being proved by induction.

- Every node in a tree is the root of a subtree.
A Generic Tree

- $T_1$
- $T_2$
- $T_3$
- $T_4$
- $T_{10}$
Tree Terminology

- **Root** of a subtree is a child of $r$. $r$ is the **parent**.
- All children of a given node are called **siblings**.
- A **leaf** (or external node) has no children.
- An **internal node** is a node with one or more children.
- A **path** from node $V_1$ to node $V_k$ is a sequence of nodes s.t. $V_i$ is the parent of $V_{i+1}$ for $1 \leq i \leq k$.
  - If there is a path from $V_1$ to $V_2$, then $V_1$ is an **ancestor** of $V_2$ and $V_2$ is a **descendant** of $V_1$. 
More Tree Terminology

- The *length* of this path is the number of edges.
  - The length of the path is one less than the number of nodes on the path (\( k - 1 \) in this example)
- The *depth* (also called *level*) of any node in a tree is the length of the path from root to the node.
- The *height* of a tree is the length of the path from the root to the deepest node in the tree.
  - A tree with only one node (the root) has height 0.
A tree node contains:
- Data Element
- Links to other nodes

Any tree can be represented with the “first-child, next-sibling” implementation.

```java
class TreeNode
{
    AnyType element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```
Printing a Child/Sibling Tree

// depth equals the number of tabs to indent name
private void listAll( int depth )
{
    printName( depth ); // Print the name of the object
    if( isDirectory() )
        for each file c in this directory
            (i.e. for each child)
            c.listAll( depth + 1 );
}

public void listAll( )
{
    listAll( 0 );
}

- What is the output when listAll( ) is used for the Unix directory tree?
K-ary Tree

- If we know the maximum number of children each node will have, $K$, we can use an array of children references in each node.

```java
class KTreeNode {
    AnyType element;
    KTreeNode children[ K ];
}
```
Pseudocode for Printing a K-ary Tree

// depth equals the number of tabs to indent name
private void listAll( int depth )
{
    printElement( depth ); // Print the object
    if( children != null )
        for each child c in children array
            c.listAll( depth + 1 );
}

public void listAll( )
{
    listAll( 0 );
}
Binary Trees

- A special case of K-ary tree is a tree whose nodes have exactly two child references -- binary trees.

- A *binary tree* is a rooted tree in which no node can have more than two children AND the children are distinguished as *left* and *right*. 
private class BinaryNode<AnyType>
{
    // Constructors
    BinaryNode( AnyType theElement )
    {
       this( theElement, null, null );
    }

    BinaryNode( AnyType theElement,
                BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
    {
       element = theElement; left = lt; right = rt;
    }

    AnyType element;            // The data in the node
    BinaryNode<AnyType> left;   // Left child reference
    BinaryNode<AnyType> right;  // Right child reference
}
Full Binary Tree

A full binary tree is a binary tree in which every node is a leaf or has exactly two children.
FBT Theorem

- Theorem: A FBT with \( n \) internal nodes has \( n + 1 \) leaves (external nodes).

- Proof by strong induction on the number of internal nodes, \( n \):

  - Base case:
    - Binary Tree of one node (the root) has:
      - zero internal nodes
      - one external node (the root)

  - Inductive Assumption:
    - Assume all FBTs with \( n \) internal nodes have \( n + 1 \) external nodes.
**FBT Proof (cont’d)**

- **Inductive Step** - prove true for a tree with $n + 1$ internal nodes (i.e. a tree with $n + 1$ internal nodes has $(n + 1) + 1 = n + 2$ leaves)
  - Let $T$ be a FBT of $n$ internal nodes.
  - Therefore $T$ has $n + 1$ leaf nodes. (Inductive Assumption)
  - Enlarge $T$ so it has $n+1$ internal nodes by adding two nodes to some leaf. These new nodes are therefore leaf nodes.
  - Number of leaf nodes increases by 2, but the former leaf becomes internal.
  - So,
    - # internal nodes becomes $n + 1$,
    - # leaves becomes $(n + 1) + 2 - 1 = n + 2$
Perfect Binary Tree

- A *Perfect Binary Tree* is a Full Binary Tree in which all leaves have the same depth.
PBT Theorem

- **Theorem:** The number of nodes in a PBT is $2^{h+1} - 1$, where $h$ is height.

- **Proof by strong induction on $h$, the height of the PBT:**
  - Notice that the number of nodes at each level is $2^l$. (Proof of this is a simple induction - left to student as exercise). Recall that the height of the root is 0.
  - **Base Case:**
    The tree has one node; then $h = 0$ and $n = 1$ and $2^{(h + 1)} - 1 = 2^{(0 + 1)} - 1 = 2^1 - 1 = 2 - 1 = 1 = n$.
  - **Inductive Assumption:**
    Assume true for all PBTs with height $h \leq H$. 

17
Proof of PBT Theorem (cont)

- Prove true for PBT with height H+1:
  - Consider a PBT with height H + 1. It consists of a root and two subtrees of height <= H. Since the theorem is true for the subtrees (by the inductive assumption since they have height <= H) the PBT with height H+1 has
    - \((2^{(H+1)} - 1)\) nodes for the left subtree
    + \((2^{(H+1)} - 1)\) nodes for the right subtree
    + 1 node for the root
  - Thus, \(n = 2 \times (2^{(H+1)} - 1) + 1\)
    \[= 2^{((H+1)+1)} - 2 + 1 = 2^{((H+1)+1)} - 1\]
A Complete Binary Tree is a binary tree in which every level is completed filled, except possibly the bottom level which is filled from left to right.
Tree Traversals

Depth-First Traversals
- Preorder – root, left subtree, right subtree
- Inorder – left subtree, root, right subtree
- Postorder – left subtree, right subtree, root

Breadth-First Traversal
- Level-order – each level is printed in turn
Tree Traversals

**Depth-first**
Inorder: A, B, C, D, E, F, G, H, I (left, root, right) ← Notice the sorting!
Postorder: A, C, E, D, B, H, I, G, F (left, right, root)

**Breadth-first**
Level-order: F, B, G, A, D, I, C, E, H
Is it possible to reconstruct a Binary Tree from just one of its pre-order, inorder, or post-order sequences?
Given two sequences (say pre-order and inorder) is the tree unique?
Finding an element in a Binary Tree?

Return a reference to node containing \( x \), return null if \( x \) is not found

```java
public BinaryNode<AnyType> find(AnyType x) {
    return find(root, x);
}

private BinaryNode<AnyType> find(BinaryNode<AnyType> node, AnyType x) {
    BinaryNode<AnyType> t = null; // in case we don’t find it
    if (node.element.equals(x)) // found it here??
        return node;

    // not here, look in the left subtree
    if (node.left != null)
        t = find(node.left, x);

    // if not in the left subtree, look in the right subtree
    if (t == null && node.right != null)
        t = find(node.right, x);

    // return reference, null if not found
    return t;
}
```
A Binary Tree can have many properties

- Number of leaves
- Number of interior nodes
- Is it a full binary tree?
- Is it a perfect binary tree?
- Height of the tree

Each of these properties can be determined using a recursive function.
Recursive Binary Tree Function

return-type function (BinaryNode<AnyType> t)
{
    // base case - usually empty tree
    if (t == null) return xxxx;

    // determine if the node referred to by t has the property
    // traverse down the tree by recursively “asking” left/right
    // children if their subtree has the property

    return theResult;
}
Is this a full binary tree?

boolean isFBT (BinaryNode<AnyType> t)
{
    // base case - an empty tree is a FBT
    if (t == null) return true;

    // determine if this node is “full”
    // if just one child, return - the tree is not full
    if ((t.left == null && t.right != null)
        ||  (t.right == null && t.left != null))
        return false;

    // if this node is full, “ask” its subtrees if they are full
    // if both are FBTs, then the entire tree is an FBT
    // if either of the subtrees is not FBT, then the tree is not
    return isFBT( t.right ) && isFBT( t.left );
}
Other Recursive Binary Tree Functions

- Count number of interior nodes
  ```c
  int countInteriorNodes( BinaryNode<AnyType> t);
  ```

- Determine the height of a binary tree. By convention (and for ease of coding) the height of an empty tree is -1
  ```c
  int height( BinaryNode<AnyType> t);
  ```

- Many others
Other Binary Tree Operations

- How do we insert a new element into a binary tree?

- How do we remove an element from a binary tree?