Class Notes CS 3137

1 The Tree Data Model: Overview

- A tree is a collection of *nodes* and *edges* (also called *links*) with certain properties:
  - There is a distinguished node called the *root*
  - Every node C (the child) other than the root is connected to one other node P (the parent). We can think of this as parent-child relationship. A parent may have many children, but a child has only 1 parent. Nodes who share a parent are called *siblings*.
  - The tree is connected in such a way that there is a *path* from every node to the root. This path is found by simply following successive parent links to the root.
- We can also define trees recursively (you can think of trees as recursive data structures):
  - Basis: a single node N is a tree. We call N the root of the tree.
  - Induction: Given trees $T_1, T_2, ..., T_k$, and a node N, we can form a new tree by creating an edge from N to the root of every other tree $T_1, T_2, ..., T_k$. N is now the parent node of $T_1, T_2, ..., T_k$.
- Besides the parent-child relationship, we can think of tree nodes having *ancestors* and *descendants*. An ancestor of a node is any other node on the path from the node to the root.
- A descendant is the inverse relationship of ancestor: A node p is a descendant of a node q if and only if q is an ancestor of p.
- We can talk about a *path* from one node to another. Given the sequence of nodes $N_1, N_2, ..., N_k$ where N_i is the parent of N_{i+1}, we can say that $N_1, N_2, ..., N_k$ is the *path* from N_1 to N_k.
- The *length* of the path is $k - 1$, one less than the number of nodes on the path. We can think of every node having a path of length 0 to itself.
- Trees also have *subtrees*. A subtree of a tree T is any node K of T (other than the root) and all of its descendants. We can also define a tree recursively as a root node with links to subtrees.
- Trees have *interior* nodes and *leaf* or *external nodes*. A leaf or external node has no children. An interior node has at least 1 child. Every node of a tree is either a leaf or an interior node. A root node is usually an interior node, but in a tree with 1 node, the root is a leaf.
- The *height* of a node N is the length of the LONGEST path from N to a leaf node. We usually talk about the height of a tree, which is the height of the root node - the longest path from the root to a leaf. All leaf nodes have height 0.
- We also use the terms *depth* and *level* of a node. The depth or level of a node N is the length of the path from the root to N. The depth or level of the root is 0.

2 General Trees

- A tree of any branching factor can be stored in a data structure that looks like this:

```
public class TreeNode {
  Object element;
  TreeNode firstChild;
  TreeNode nextSibling;
}
```

- We can encode any general tree as a special kind of binary tree: the 2 child pointers are FirstChild and NextSibling. Each sibling is part of a linked list which can be of arbitrary length.
- Because the root node will NEVER have a right child (roots have no siblings), we can use this empty pointer to create a *forest* or *orchard* of trees.
- Trees are linked by having their root nodes right child point to the next tree in the orchard.

3 Binary Trees

- A *binary* tree is a tree with nodes that have at most 2 children.
- A *full* binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node has zero or two children.
- A *perfect* binary tree (sometimes *complete* binary tree) is a binary tree in which all leaves are at the same depth or level, and all internal nodes have 2 children.
-

Height of Tree	Number of Nodes	Interior Nodes	Leaf Nodes
0	1	0	1
1	3	1	2
2	7	3	4
3	15	7	8
4	31	15	16
.	.	.	.

- Prove that the total number of nodes in a perfect binary tree of height K is $2^{K+1} - 1$. We will prove this by induction. Induction is a powerful tool that we can use to analyze recursive objects such as trees.

  1. Basis: For a tree of height $K = 0$, we have 1 node (the root). Thus, $2^{K+1} - 1 = 2^1 - 1 = 1$, and we have shown that the base case is true.
  2. Induction: Assume that it is true that a perfect binary tree of height K has $2^{K+1} - 1$ nodes. We need to show that a perfect binary tree of height $K + 1$ has $2^{K+2} - 1$ nodes.

     A perfect binary tree of height $K + 1$ has a root node, and 2 subtrees, each of height K. Since we know that the number of nodes in a subtree of height K is $2^{K+1} - 1$, this new tree of height $K + 1$ has the following number of nodes

$$2^{K+1} - 1 \; + \; 1 \; + 2^{K+1} - 1$$

$$2 \cdot 2^{K+1} - 1 \; + \; 1 \; - 1$$

$$2^{K+2} - 1$$

- We can invert this last relationship to find the height of a perfect binary tree given the number of nodes.

  If the number of nodes $N = 2^{K+1} - 1$, then the height $K = Log(N + 1) - 1$. So, we can say that the height K of a perfect binary tree of N nodes is:

$$K \approx Log(N)$$

4 Binary Tree Implementation

- We can define a JAVA class MyBinaryNode that is defined as: follows:

```
class MyBinaryNode  {

 MyBinaryNode( Object theElement ,MyBinaryNode lt, MyBinaryNode rt )
      {

         element  = theElement;
          left     = lt;
          right    = rt;

      }

   Object element;            // The data in the node
   MyBinaryNode left;         // Left child
   MyBinaryNode right;        // Right child
 }
```

5 Useful Tree Methods

- We can count the number of nodes in a binary tree using a recursive program:

```
public static int countnodes(MyBinaryNode t)
{
 if(t==null)return 0;
    else return(1+countnodes(t.left)+countnodes(t.right));
}
```

- We can also calculate the height of the tree:

```
public static int treeHeight(MyBinaryNode t)
{
  if(t==null) return -1; else return(1 + max(treeHeight(t.left),
                                        treeHeight(t.right)));
}
public static int max(int a,int b)
{
if(a>b) return a; else return b;
}
```

- To print a tree out so it looks like a tree on a terminal, we can use an inorder traversal that indents on the page depending upon what level of the tree we are at. Try it, you'll like it! We call this program with a pointer to the root and indent of 0: printTree(T,0)

```
//prints the tree out, indenting for depth

public static void printTree( MyBinaryNode t , int indent)
 {
            if( t != null )
            {
                printTree( t.right, indent + 3 );
                for(int i=0;i<indent;i++)
                   System.out.print(" ");
                System.out.println( t.element );
                printTree( t.left , indent + 3 );
            }
 }
```

- A measure of the cost of searching a tree is called IPL: Internal Path Length. This simply measures the total cost of traversing the tree from the root to each node. It is the sum of *all* the path lengths from the root to each node in the tree.

$$IPL = \sum_j depth(node\, j)$$

For a perfect binary tree, we can calculate the IPL as:

$$IPL \;=\; \sum_{i=0}^{H} i \cdot 2^i,\; H = Height\,of\,tree$$

Trees that are out of balance will have a larger IPL for the same number of nodes. We can also get a useful number if we divide the sum of the path lengths by the number of nodes, which gives us an "average" traversal time of the tree.

$$Average\,Cost\,of\,a\,Find \;=\; IPL\,/\,N$$

How do we compute the IPL on a given tree? We can use a recursive program that simply adds 1 to the path length each time we go to a new level of the tree on a left or right link. When the recursion hits the leaf nodes (who have NULL link children), we end the recursion.

```
// calculates Internal Path Length

public static int ipl(MyBinaryNode t, int level)
{
    if(t==null) return 0;
    else {
         return(ipl(t.left, level+1) + level+ ipl(t.right,level+1));
         }
}
```

6 Binary Tree Traversals

- We can traverse a binary tree in a number of different ways. Three of the most common traversals are *Preorder, Inorder and Postorder*
- Preorder: Visit the Root, then visit the Left Subtree, then visit the Right Subtree
- Inorder: Visit the Left Subtree, then visit the Root, then visit the Right Subtree
- Postorder: Visit the Left Subtree, then visit the Right Subtree, then Visit the Root.
- Here is an outline of the code for these traversals:

```
public static void inOrder(MyBinaryNode t){

           if (t!=null){
               inOrder(t.left);
               System.out.print(t.element + " ");
               inOrder(t.right);
           }
}
```

```
public static void preOrder(MyBinaryNode t){
            if (t!=null){
                System.out.print(t.element + " ");
                preOrder(t.left);
                preOrder(t.right);
            }
}
public static void postOrder(MyBinaryNode t){
            if (t!=null){
                postOrder(t.left);
                postOrder(t.right);
                System.out.print(t.element + " ");
            }
}
```

7 Level Order Traversal

Another traversal is called *Level Order*. This will visit the tree in a top down fashion, first visitng the root, then the root's children, then the root's grandchildren and so on. It is also called Breadth First Search

The trick in *Level Order* traversal is to postpone processing of children until the parents are visited. We do this by queueing up requests for visits to chldren each time we viist a new node.

```
void LevelOrderTraversal(BinaryNode t) {
   EnQueue node t into a Queue Q - check to make sure t not null
   while (Q not empty) {
      nextnode= DeQueue(Q);
      if (nextnode != null) {
         Visit(nextnode);
         EnQueue nextnode.left in Q; // insert left link at rear
         EnQueue nextnode.right in Q; // insert right link on rear of Q
}}}
```

8 Expression Trees

Expression trees are a very useful application for binary trees. We know that storing a normal infix expression is wasteful, since we need to store the parentheses in the expression to properly evaluate the expression.

A compiler will read a mathematical expression in a language like JAVA, and perform a transformation to put the expression into an unambiguous (i.e no parentheses) form. Expression trees can be very useful for:

- Evaluation of the expression: assigning a numeric value to the expression
- Generating correct compiler code to actually compute the expression's value at execution time
- Performing symbolic mathematical operations on the expression such as differentiation

Expression trees invoke a hierarchy on the operations in the expression. Terms deeper in the tree (at greater depth) get evaluated first, and this allows us to establish the correct precedence of operations.

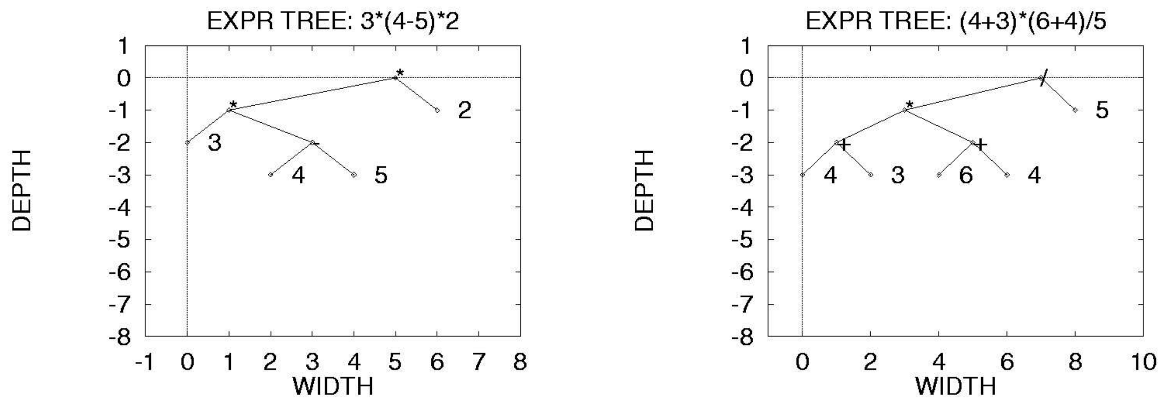

Figure 1: Expression trees

8.1 Creating an Expression Tree from a Postfix Expression

If we are given a postfix expression, how do we create an expression tree? It's really very easy. As we read the postifx from left to right, we place all operands on a stack. Once we read an operator, we form a binary tree with the operator as the root node, and the left and right children being the top two stack entries which are the operands. Then we push this tree onto the stack. As we continue to push and pop stack entries, we end up with the stack containing a single entry which is the root of the expression tree.

8.2 Evaluating an Expression Tree

How do we go about evaluting an Expression Tree (ET) that contains numbers and binary arithemetic operators $(+, -, *, /, \wedge)$? Its quite simple. We can recursively define an Expresison Tree as:

- An ET is a single node containing a numeric value.
- An ET is a binary tree, whose root node is an operator, and whose left and right children are ET's.

We can then create an evaluation algorithm to evaluate the numeric expression using a tree traversal (the method returns the double value of the expression). We can keep an information field *nodetype* in each node that tells us whether the data stored at a node is a an operand or operator: We can extend the BinaryNode class by adding the nodetype field that tells whether the node is an operator or an operand.

```
double EvalTree( EBinaryNode T ){
 if( T == NULL) return 0;
   else if( T.nodetype.equals(‘‘OPERAND’’)) return ((double)T.element);
       else { // must be an operator...
              Operate = (char)T.element;
              Operand1 = EvalTree( T.left);
              Operand2 = EvalTree( T.right);
              return( ApplyOperator(Operand1, Operate,Operand2));
             }
}
```

When we actually store these expressions, we use a placehoder for a variable name, and include a pointer to a list where we can find the variable's name and a value if one was assigned. In figure 2 you can see what the compiler might store for a particular expression.

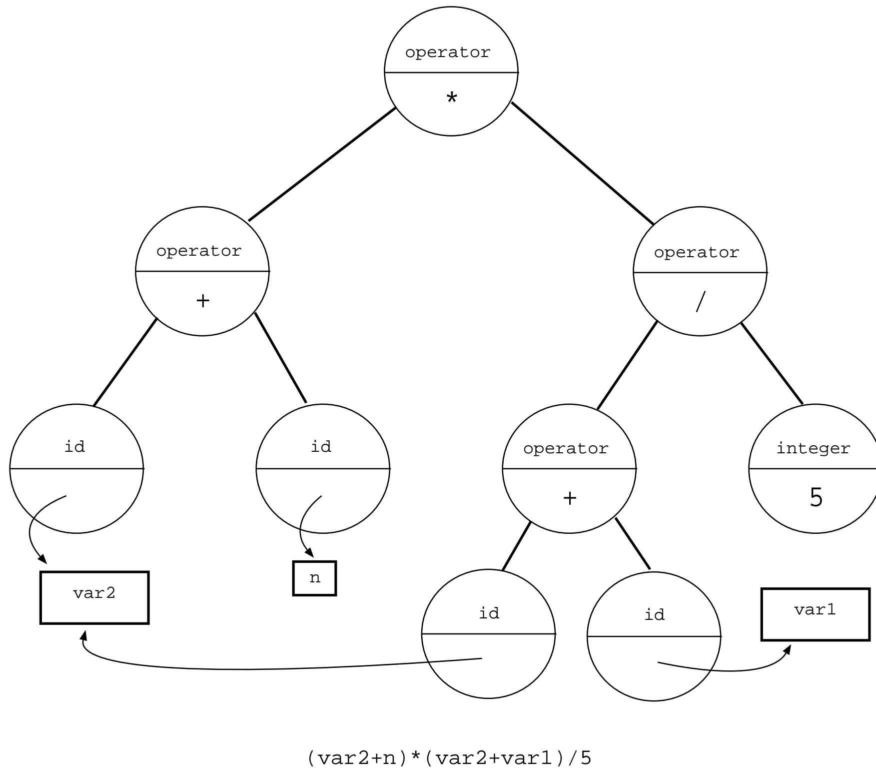

Figure 2: Expression trees with placeholders