

Chapter 18

Trees

Terminology (mixture of horticulture and genealogy)

- Tree – Collection of nodes and edges such that:
 1. All nodes, except the root, have exactly one predecessor. The root has no predecessor.
 2. All nodes are reachable by following paths from the root.
- Parent – predecessor of a node.
- Child – successor of a node.
- Siblings – nodes having the same parent.
- Degree – number of children of a node.
- Edges – branches.
- Leaves – nodes with out degree 0 (no children).
- Level – vertical displacement (root has level 1).
- Depth of tree – level of leaves most distant from the root (MAX LEVEL).
- Descendant – all nodes you can reach from a given node.
- Ancestor – node on path from root to node.
- Subtree – a node and all its descendants.
- Binary tree – every node has no more than two children.
- Similar trees – trees that have the same shape.
- Strictly binary – no one child nodes (termed 2-trees in text).
- Full – binary tree of height h which contains exactly $2^h - 1$ elements.
- Complete – full tree with possibly partial last level (pushed to left).

Binary Trees

- Binary trees should have been covered in CS1720, but we will quickly review them here.
- Here is some code that represents the way I like to code binary trees. It represents the best of the ideas I've gleaned from others.

```
// ONE APPROACH TO CODING BINARY TREES
// The fact that all fields are public appears to be a violation of what you
// have been taught about protecting the access.
// However, public fields are defensible in the following context.
// The TreeNode class is only used in conjunction with the BinaryTree.
// BinaryTree does not allow hostile classes to access any of its data.
// In particular, no other class can get at the root
// (unless it is a descendant class). Thus, the TreeNode variables
// are inaccessible.
// This is analogous to protecting your valuables by locking the house
// rather than by locking the drawer containing them. Everything
// is locked up either way.
```

```

// TreeNode: Contains an individual node of an expression tree
class TreeNode
{
public:
    int value; // integer value of node, if a leaf
    char operator; // operator (+-*/) if internal node
    TreeNode *left; // left child
    TreeNode *right; // right child
    TreeNode( int Val, char Op, TreeNode *L = NULL,
              TreeNode *R = NULL ) : value(Val), operator(Op),
              left( L ), right( R ) {}
};

class BinaryTree
{ protected:
    void makeEmpty( TreeNode * & T );
    void printTree( TreeNode * T ,int indent) ;
    TreeNode *root; // Root of expression tree
    void getSubtree(TreeNode *& root, string exp,int& beg);
public:
    BinaryTree( ) : root( NULL ) { }
    BinaryTree(string exp);
    virtual ~BinaryTree( ) { makeEmpty( root ); }
    void makeEmpty( ) { makeEmpty( root ); }
    void printTree( ) { printTree( root,1 ); }
};

// Print tree rooted at T, indented by indent positions
// Print the tree "prettily" as outlined in assignment 3
void BinaryTree:: printTree( TreeNode *T,int indent )
{ // You Provide

}

// Recursively free the space occupied by the subtree rooted at T
void BinaryTree:: makeEmpty( TreeNode * & T )
{ if( T != NULL )
    { makeEmpty( T->left );
      makeEmpty( T->right );
      delete T;
      T = NULL;
    }
}

// Create a tree from an expression stored in exp
// using only the characters beginning at beg
// Store the tree in the subtree rooted at root
// We are assuming that nodes are a single digit.
// The tree represented by exp is fully parenthesized: ((3+5)/(2-6))
void BinaryTree::getSubtree(TreeNode *& root, string exp,int& beg)

```

```

{ TreeNode * left, *right;
  if (exp[beg]=='('){ // You fill in

  }
  else{ // expecting a single digit operand
    root = new TreeNode(exp[beg]-'0', ' ');
    beg++;
  }
}

// Construct an expression tree using all of the of exp
BinaryTree::BinaryTree(string exp)
{ int loc = 0;
  root = NULL;
  getSubtree(root,exp,loc);
}

```

- Try to code the following:
 - Write the code to evaluate an expression tree (of the variety seen in the example above).
 - Write the method to determine the number of nodes in a tree.
 - Write the code to find the largest element in a tree.
 - Write the code to sum all the leaf node values.
 - Write the code to flip a tree (exchange left and right subtrees throughout).
 - Write the code to determine the maximum level of a tree.
 - Write the code to find the number of nodes that have only one child.

Expression Trees

- Tree whose interior nodes represent an operation (e.g., addition, multiply) and leaf nodes represent operands.
- For example, see Figure 1 below.

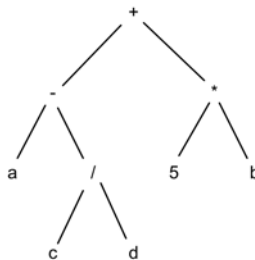


Figure 1 Expression Tree

Traversal

- Sometimes we need to visit each node exactly once (imposed linearity).

- Typically, we visit the left branch before the right one.
- There are three traversal methods:
 1. Preorder
 - Order of visit – node, left, right
 - For an example, consider nodes are chapters or subsections and arcs are containment relationship, preorder gives table of contents.
 - This is the prefix form of an expression
 2. Inorder
 - Order of visit – left, node, right
 - If tree represents expression tree, inorder gives algebraic expression.
 - Pay attention to precedence. The higher the precedence, the lower in the tree.
 - A-C/D + 5*D
 3. Postorder
 - Order of visit – left, right, node
 - Postfix form of an expression
- The procedure to print the contents of a tree in preorder is shown in Figure 2.
- Discuss by level traversal (breadth first). Use queue as an auxiliary structure.

```

template <class Etype>
void BinarySearchTree<Etype>::
Preorder( TreeNode<Etype> *T) const
{ if (T==NULL) return;
  cout << T->Element << endl;
  Preorder(T->Left);
  Preorder(T->Right);
}

```

Figure 2 Preorder Traversal

Formula Based Representation

- Arrays (thus being able to use formulas) work well to represent complete binary trees.
- The root node is stored at node 1.
- Given a node in the tree at subscript j , we get the locations of other nodes as follows:
 - Left child – $2j$.
 - Right child – $2j+1$.
 - Parent – $j/2$, using integer division.

Efficiency Issues

Auxiliary Stack

- Since recursion is expensive, it is sometimes better to code iteratively. Then a stack is used as an auxiliary data structure.
- See the code in Figure 3.

```

inorder(node * root)
{ node * p;
  STACKCLASS Stack;
  for (p = root; p != NULL; p = p->Left) Stack.push(p);
  while (!Stack.isempty())
  { p = Stack.pop();
    cout << p->Element;
    for(p = p->Right; p != NULL; p = p->nleft)
      Stack.push(p);
  }
}

```

Figure 3 Using a Stack with a Tree

General Trees

- Fixed number of children for a given parent.
- If there are a fixed number of children (say MAX), the definition in Figure 4 works well.

Left Most Child Next Right Sibling

- Instead of having a pointer to all children, each node stores a pointer to its left most child and its next right sibling.
- In effect, it becomes a binary tree.
- Example

```
#define MAX 4
Class node
{ char Name[20];
  /* Don't have to allocate space for name*/
  node * Child[MAX];
};
```

Figure 4 Fixed Number of Children

Parent Pointers

- Sometimes it is helpful to maintain a parent pointer for each node.
- A declaration for such a scheme is shown in Figure 5.

```
class node
{ char Name[20];
  node * Parent;
  node * Left;
  node * Right;
};
```

Figure 5 Parent Pointer