CMSC 341

Introduction to Trees

Tree ADT

Tree definition

- A tree is a set of nodes.
- The set 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, … T_k, each of whose roots are connected by a directed edge from r.

Basic 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

More Tree Terminology

A *path* from node V_1 to node V_k is a sequence of nodes such that V_i is the parent of V_{i+1} for $1 \le i \le k$.

The *length* of this path is the number of edges encountered. The length of the path is one less than the number of nodes on the path (k – 1 in this example)

The *depth* of any node in a tree is the length of the path from root to the node.

All nodes of the same depth are at the same *level*.

The *depth of a tree* is the depth of its deepest leaf.

The *height* of any node in a tree is the length of the longest path from the node to a leaf.

The *height of a tree* is the height of its root.

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 *descendent* of V_1.

Tree Storage

A tree node contains:

- Element
- Links
  - to each child
  - to sibling and first child

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*.

A *full BT* is a BT in which every node either has two children or is a leaf (every interior node has two children).

FBT Theorem

Theorem: A FBT with n internal nodes has n + 1 leaf nodes.

Proof by strong induction on the number of internal nodes, n:

Base case: BT of one node (the root) has:

zero internal nodes

one external node (the root)

Inductive Assumption:

Assume all FBTs with up to and including n internal nodes have n + 1 external nodes.

Proof (cont)

Inductive Step (prove true for 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.
- It therefore has $n + 1$ external 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 $(n + 1) + 1 = n + 2$

Proof (more rigorous)

Inductive Step (prove for n+1):

- Let T be any FBT with n + 1 internal nodes.
- Pick any leaf node of T, remove it and its sibling.
- Call the resulting tree T1, which is a FBT
- One of the internal nodes in T is changed to a external node in T1
  - T has one more internal node than T1
  - T has one more external node than T1
- T1 has n internal nodes and n + 1 external nodes (by inductive assumption)
  - Therefore T has (n + 1) + 1 external nodes.

Perfect Binary Tree

A *perfect BT* is a full BT 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 $2^{(h+1)} = 2^{(0+1)} - 1 = 2^1 - 1 = 2 - 1 = 1 = n$

Proof of PBT Theorem(cont)

Inductive Assumption:

Assume true for all trees with height $h \leq H$

Prove true for tree with height H+1:

Consider a PBT with height H + 1. It consists of a root and two subtrees of height H. Therefore, since the theorem is true for the subtrees (by the inductive assumption since they have height = H)

$n = (2^{(H+1)} - 1)$ for the left subtree

$+ (2^{(H+1)} - 1)$ for the right subtree

$+ 1$ for the root

$= 2 * (2^{(H+1)} - 1) + 1$

$= 2^{((H+1)+1)} - 2 + 1 = 2^{((H+1)+1)} - 1.$ QED

Other Binary Trees

Complete Binary Tree

A *complete BT* is a perfect BT except that the lowest level may not be full. If not, it is filled from left to right.

Augmented Binary Tree

An *augmented binary tree* is a BT in which every unoccupied child position is filled by an additional “augmenting” node.

Path Lengths

The ***internal path length*** (IPL) of a rooted tree is the sum of the depths of all of its internal nodes.

The ***external path length*** (EPL) of a rooted tree is the sum of the depths of all the external nodes.

There is a relationship between the IPL and EPL of Full Binary Trees.

If n_i is the number of internal nodes in a FBT, then

$$EPL(n_i) = IPL(n_i) + 2n_i$$

Example:

$n_i =$

$EPL(n_i) =$

$IPL(n_i) =$

$2\ n_i =$

Proof of Path Lengths

Prove: $EPL(n_i) = IPL(n_i) + 2\,n_i$ by induction on number of internal nodes

Base: $n_i = 0$ (single node, the root)

$EPL(n_i) = 0$

$IPL(n_i) = 0$; $2\,n_i = 0$ $0 = 0 + 0$

IH: Assume true for all FBT with $n_i < N$

Prove for $n_i = N$.

Proof: Let T be a FBT with $n_i = N$ internal nodes.

Let n_{iL}, n_{iR} be # of internal nodes in L, R subtrees of T

then $N = n_i = n_{iL} + n_{iR} + 1 \implies n_{iL} < N; n_{iR} < N$

So by IH:

$$EPL(n_{iL}) = IPL(n_{iL}) + 2\, n_{iL}$$

and $EPL\,(n_{iR}) = IPL(n_{iR}) + 2\, n_{iR}$

For T,

$$EPL(n_i) = EPL(n_{iL}) + n_{iL} + 1 + EPL(n_{iR}) + n_{iR} + 1$$

By substitution

$$EPL(n_i) = IPL(n_{iL}) + 2\, n_{iL} + n_{iL} + 1 + IPL(n_{iR}) + 2\, n_{iR} + n_{iR} + 1$$

Notice that $IPL(n_i) = IPL(n_{iL}) + IPL(n_{iR}) + n_{iL} + n_{iR}$

By combining terms

$$EPL(n_i) = IPL(n_i) + 2\,(n_{iR} + n_{iL} + 1)$$

But $n_{iR} + n_{iL} + 1 = n_i$, therefore

$$EPL(n_i) = IPL(n_i) + 2\, n_i \qquad \text{QED}$$

Traversal

Inorder

Preorder

Postorder

Levelorder

Constructing Trees

Is it possible to reconstruct a BT from just one of its pre-order, inorder, or post-order sequences?

Constructing Trees (cont)

Given two sequences (say pre-order and inorder) is the tree unique?

Tree Implementations

What should methods of a tree class be?

Tree class

```
template <class Object>
class Tree {
  public:
  Tree(const Object& notFnd);
  Tree (const Tree& rhs);
  ~Tree();

  const Object &find(const Object& x) const;
  bool isEmpty() const;
  void printTree() const;
  void makeEmpty();
  void insert (const Object& x);
  void remove (const Object& x);
  const Tree& operator=(const Tree &rhs);
```

Tree class (cont)

```
private:
  TreeNode<Object> *root;
  const Object ITEM_NOT_FOUND;
  const Object& elementAt(TreeNode<Object> *t) const;
  void insert (const Object& x, TreeNode<Object> *& t)
  const;
  void remove (const Object& x, TreeNode<Object> *& t)
  const;
  TreeNode<Object> *find(const Object& x,
                         TreeNode<Object> * t) const;
  void makeEmpty(TreeNode<Object> *& t) const;
  void printTree(TreeNode<Object * t) const;
  TreeNode<Object> * clone(TreeNode<Object> * t)const;
};
```

Tree Implementations

Fixed Binary

- element
- left pointer
- right pointer

Fixed K-ary

- element
- array of K child pointers

Linked Sibling/Child

- element
- firstChild pointer
- nextSibling pointer

TreeNode : Static Binary

```
template <class Object>
class BinaryNode {
   Object element;
   BinaryNode *left;
   BinaryNode *right;

   BinaryNode(const Object& theElement,
              BinaryNode* lt,
              BinaryNode* rt)
        : element (theElement), left(lt), right(rt) {}

   friend class Tree<Object>;
};
```

Find : Static Binary

```
template <class Object>
BinaryNode<Object> *Tree<Object> ::
find(const Object& x, BinaryNode<Object> * t) const {
   BinaryNode<Object> *ptr;

   if (t == NULL)
        return NULL;
   else if (x == t->element)
        return t;
   else if (ptr = find(x, t->left))
        return ptr;
   else
        return(ptr = find(x, t->right));
   }
```

Insert : Static Binary

Remove : Static Binary

TreeNode : Static K-ary

```
template <class Object>
class KaryNode {
   Object element;
   KaryNode * children[MAX_CHILDREN];

   KaryNode(const Object& theElement);

   friend class Tree<Object>;
};
```

Find : Static K-ary

```
template <class Object>
KaryNode<Object> *KaryTree<Object> ::
find(const Object& x, KaryNode<Object> *t) const
{
   KaryNode<Object> *ptr;

   if (t == NULL)
        return NULL;
   else if (x == t->element)
        return t;
   else {
        i =0;
        while ((i < MAX_CHILDREN)
        &&    !(ptr = find(x, t->children[i])) i++;
        return ptr;
   }
}
```

Insert : Static K-ary

Remove : Static K-ary

TreeNode : Sibling/Child

```
template <class Object>
class KTreeNode {
   Object element;
   KTreeNode *nextSibling;
   KTreeNode *firstChild;

   KTreeNode(const Object& theElement,
             KTreeNode *ns,
             KTreeNode *fc)
        : element (theElement),  nextSibling(ns),
          firstChild(fc) {}

   friend class Tree<Object>;
};
```

Find : Sibling/Child

```
template <class Object>
KTreeNode<Object> *Tree<Object> ::
find(const Object& x, KTreeNode<Object> *t) const
{
   KTreeNode<Object> *ptr;

   if (t == NULL)
        return NULL;
   else if (x == t->element)
        return t;
   else if (ptr = find(x, t->firstChild))
        return ptr;
   else
        return(ptr = find(x, t->nextSibling));
}
```

Insert : Sibling/Child

Remove : Sibling/Parent