Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 18
Trees

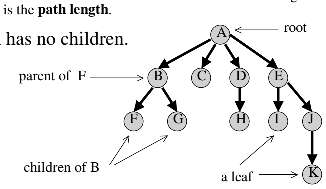
Contents

We will learn:

- What is a tree and how is it used in a file system;
- What is the binary tree and how can one examine the binary tree;
- How to implement the tree operations, using recursions; and
- How can one visit all nodes of a tree.

General Trees

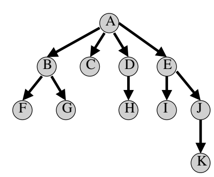
- A **tree** consists of a **set of nodes** and a **set of directed edges** that connect pairs of nodes.
- We will consider **rooted trees**. A rooted tree has the following properties:
  - One node is distinguished as the **root.**
  - Every node *c,* except the root, is connected by an edge from exactly one other node *p. Node p* is *c*'s **parent**, and *c* is one of *p*'s **children**.
  - A unique path traverses from the root to each node. The number of edges that must be followed is the **path length**.
- A leaf is a node which has no children.
- A tree with N nodes must have N-1 edges.



A → K path length is 3

General Trees: definitions

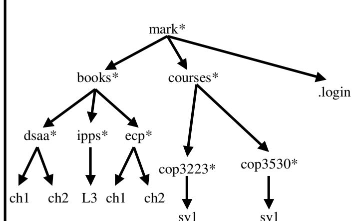
- The **depth** of a node is the length of the path from the root to the node.
- The **height** of a node is the length of the path from the node to the deepest leaf.
- Nodes with the same parent are called **siblings.**
- If there is a path from *u* to *v* then *u* is an **ancestor** of *v* and *v* is **descendant** of *u*. (**Proper ancestor** and **proper descendant** if u ≠ v.)
- The **size** of a node is the number of descendants the node has (including the node itself).



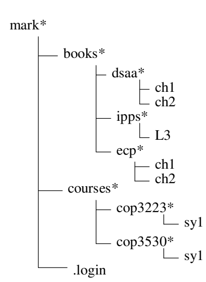
Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

An Application: File Systems

- File systems use treelike structures.



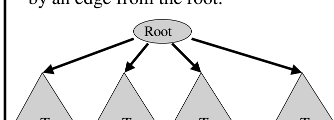
A Unix Directory.



The directory listing for that tree.

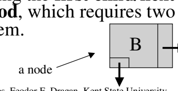
Recursive Definition of a Tree

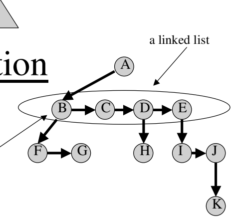
- Either a tree is empty or it consists of a root and zero or more nonempty subtrees each of whose roots are connected by an edge from the root.



Presentation

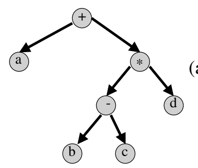
- General trees can be implemented by using the **first child/next sibling method**, which requires two pointer per item.





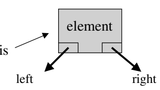
Binary Trees

- A **binary tree** is a tree in which no node can have more than two children.
- We name them *left* and *right*.
- An expression tree is one example of the use of binary trees.
- Such trees are central structures in compiler design.
- An important use of binary trees is in other data structures, notably the binary search tree and the priority queue.



(a+((b-c)*d))

a node in a binary tree is



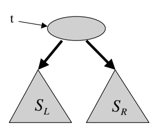

```
// BinaryNode class; stores a node in a tree.
//
// CONSTRUCTION: with (a) no parameters, or (b) an Object,
//     or (c) an Object, left pointer, and right pointer.
// *******************PUBLIC OPERATIONS**********
// int size( )            --> Return size of subtree at node
// int height( )          --> Return height of subtree at node
// void printPostOrder( ) --> Print a postorder tree traversal
// void printInOrder( )   --> Print an inorder tree traversal
// void printPreOrder( )  --> Print a preorder tree traversal
// BinaryNode * duplicate( )   --> Return a duplicate tree
//
*******************ERRORS************************
// None.

template <class Object>
class BinaryNode
{public:
   BinaryNode( const Object & theElement = Object( ),
       BinaryNode *lt = NULL, BinaryNode *rt = NULL );

   static int size( BinaryNode *t );
   static int height( BinaryNode *t );

   void printPreOrder( ) const;
   void printPostOrder( ) const;
   void printInOrder( ) const;
```

The *BinaryNode* class interface.

```
BinaryNode *duplicate( ) const;

 public:   // To keep things simple
   Object      element;
   BinaryNode *left;
   BinaryNode *right;
};
```



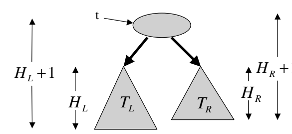
```
// Return size of subtree rooted at t.
template <class Object>
int BinaryNode<Object>::size( BinaryNode<Object> * t )
{  if( t == NULL )
       return 0;
   else
       return 1 + size( t->left ) + size( t->right );
}


// Return a pointer to a node that is the root of a
// duplicate of the subtree rooted at the current node.
template <class Object>
BinaryNode<Object> * BinaryNode<Object>::duplicate( ) const
{ BinaryNode<Object> *root =
                 new BinaryNode<Object>( element );

   if( left != NULL )           // If there's a left subtree
      root->left = left->duplicate( );   // Duplicate; attach
   if( right != NULL )          // If there's a right subtree
      root->right = right->duplicate( ); // Duplicate; attach

   return root;                 // Return resulting tree
}
```



$$S_T = S_L + S_R + 1$$

Recursive routines for size and duplicate.


```
#define max MAX

template <class Comparable>
Comparable max( const Comparable & a,
                const Comparable & b )
{
   return a > b ? a : b;
}

// Return height of tree rooted at t.
template <class Object>
int BinaryNode<Object>::height( BinaryNode<Object> * t )
{
   if( t == NULL )
      return -1;
   else
      return 1 + max( height( t->left ), height( t->right ) );
}

#undef max
```

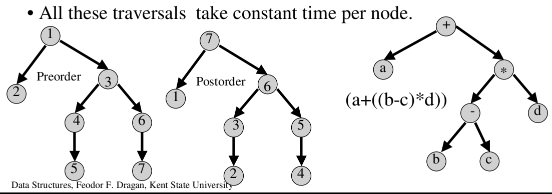


$$H_T = Max(H_L + 1,\ H_R + 1)$$

Recursive routine for height.

Binary Tree Traversal

- A **preorder tree traversal:**
  - In a preorder tree traversal, the node is processed and then its children are processed recursively. (e.a., duplicate routine)
- A **postorder tree traversal:**
  - In a postorder tree traversal, the node is processed after both children are processed recursively. (e.a., size, height routines)
- An **inorder tree traversal:**
  - In an inorder tree traversal, the left child is recursively processed, the node is processed, and the right child is recursively processed. (e.a., an expression tree)
- All these traversals take constant time per node.




```
// Print the tree rooted at current node using preorder traversal.
template <class Object>
void BinaryNode<Object>::printPreOrder( ) const
{
   cout << element << endl;      // Node
   if( left != NULL )
      left->printPreOrder( );     // Left
   if( right != NULL )
      right->printPreOrder( );    // Right
}

// Print the tree rooted at current node using postorder traversal.
template <class Object>
void BinaryNode<Object>::printPostOrder( ) const
{
   if( left != NULL )            // Left
      left->printPostOrder( );
   if( right != NULL )           // Right
      right->printPostOrder( );
   cout << element << endl;      // Node
}
```

Recursive routines for print functions.

```
// Print the tree rooted at current node using
// inorder traversal.
template <class Object>
void BinaryNode<Object>::printInOrder( ) const
{
   if( left != NULL )            // Left
      left->printInOrder( );
   cout << element << endl;      // Node
   if( right != NULL )
      right->printInOrder( );    // Right
}
```

The *BinaryTree* class interface.

```
// BinaryTree class; stores a binary tree.
// CONSTRUCTION: with (a) no parameters or (b) an object to
//    be placed in the root of a one-element tree.
//// *******************PUBLIC OPERATIONS*******
// Various tree traversals, size, height, isEmpty, makeEmpty.
// Also, the following tricky method:
// void merge( Object root, BinaryTree t1, BinaryTree t2 )
//                        --> Construct a new tree
//*******************ERRORS*********************
// Error message printed for illegal merges.

template <class Object>
class BinaryTree
{public:
    BinaryTree( ) : root( NULL ) { }
    BinaryTree( const Object & rootItem ) : root( new Node( rootItem ) ) { }
    BinaryTree( const BinaryTree & rhs )  : root( NULL ) { *this = rhs; }
    ~BinaryTree( )   { makeEmpty( ); }
    const BinaryTree & operator= ( const BinaryTree & rhs );
 // Recursive traversals, with printing
    void printPreOrder( ) const { if( root != NULL ) root->printPreOrder( ); }
    void printInOrder( ) const { if( root != NULL ) root->printInOrder( ); }
    void printPostOrder( ) const { if( root != NULL ) root->printPostOrder( ); }

    void makeEmpty( ) { makeEmpty( root ); }
    bool isEmpty( ) const { return root == NULL; }
```

- The *BinaryTree* methods use the *BinaryNode* routines on the *root*.
- The only data member in the *BinaryTree* is a pointer to the *root* node.

The *BinaryTree* class interface (cont.).

```
  // Combine t1 and t2
     void merge( const Object & rootItem, BinaryTree & t1, BinaryTree & t2 );

     int size( ) const { return Node::size( root ); }
     int height( ) const { return Node::height( root ); }

   private:
     typedef BinaryNode<Object> Node;
     Node *root;

     friend class TreeIterator<Object>;
     void makeEmpty( BinaryNode<Object> * & t );
  };
```

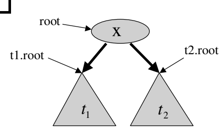
MakeEmpty private function.

```
// Make tree rooted at t empty, freeing nodes, and setting t to NULL.
template <class Object>
void BinaryTree<Object>::makeEmpty( BinaryNode<Object> * & t )
{  if( t != NULL )
    {  makeEmpty( t->left );
       makeEmpty( t->right );

       delete t;

       t = NULL;  }
}
```


The *Merge* routine.

- The merge routine is a one-liner in principle.

```
root = new BinaryNode<Object>( rootItem, t1.root, t2.root);
```



- However, we must also handle aliasing, avoid memory leaks, ensure that a node is not in two trees, and check for errors.
- Different calls like

t3.merge(x, t1, t2); — We set the original trees' *root* to *NULL* so that each node is in one tree.

t1.merge(x, t1, t2);
t2.merge(x, t1, t2); — If the input tree is aliased to the output tree, we must avoid having the resultant *root* point to *NULL*.

t1.merge(x, t3, t3); — If the two input trees are aliases, we should disallow the operation unless the trees are empty.

should work correctly.

- Hence, see the next slide.

The *Merge* routine (cont).

```
// Merge routine for BinaryTree class.
//
// Forms a new tree from rootItem, t1 and t2.
// Does not allow t1 and t2 to be the same.
// Correctly handles other aliasing conditions.

template <class Object>
void BinaryTree<Object>::merge( const Object &  rootItem, BinaryTree<Object> & t1,
                                                                         BinaryTree<Object> & t2 )
{
   if( t1.root == t2.root && t1.root != NULL )
   { cerr << "Cannot merge a tree with itself; merge aborted." << endl; return; }

   Node *oldRoot = root;   // Save old root

   // Allocate new node
   root = new Node( rootItem, t1.root, t2.root );

   // Deallocate nodes in the original tree
   if( this != &t1 && this != &t2 ) makeEmpty( oldRoot );

   // Ensure that every node is in one tree
   if( this != &t1 ) t1.root = NULL;
   if( this != &t2 ) t2.root = NULL;
}
```


Binary Tree Traversal: Iterator Classes

- preorder tree traversal
- postorder tree traversal
- inorder tree traversal

All these traversals take constant time per node.

- Each iterator allows us (like the Linked List Iterator)
  - to go to the first node (root),
  - advance to the next node,
  - test whether we have gone past the last node, and
  - access the current node.
- The order in which nodes are visited is determined by the type of traversal.
- We decide to implement all traversals (preorder, postorder, inorder, level-order) non-recursively.
- Implementations for preorder, postorder and inorder traversals use a stack (to imitate the recursion). The level-order traversal implementation uses a queue.

The *TreeIterator* *abstract base* class.

```
// TreeIterator class interface; maintains "current position".
//
// CONSTRUCTION: with a tree to which the iterator is bound.
// ******************PUBLIC OPERATIONS***********
//     First two are not virtual, last two are pure virtual
// bool isValid( )      --> True if at valid position in tree
// Object retrieve( )   --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance
// ******************ERRORS***********************
// BadIterator is thrown for illegal access or advance.
template <class Object>
class TreeIterator
{public:
    TreeIterator( const BinaryTree<Object> & theTree )  : root( theTree.root ), current( NULL ) { }
    virtual ~TreeIterator( ) { }

    virtual void first( ) = 0;
    bool isValid( ) const { return current != NULL; }
    const Object & retrieve( ) const;
    virtual void advance( ) = 0;

  protected:
    const BinaryNode<Object> *root;
    const BinaryNode<Object> *current;
};
```

Each type of traversal is represented by a derived class (from this abstract base class).

Will be implemented later.

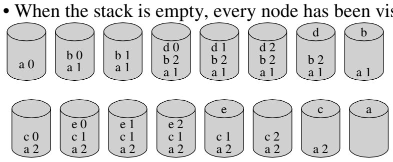
As before for Linked List.

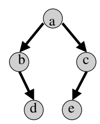
To allow the derived classes to access them.

A pointer to the tree root and an indication of the current node.

Postorder Traversal

- Postorder traversal maintains a stack that stores nodes that have been visited but those recursive calls are not yet complete.
  - In a postorder tree traversal, the node is processed after both children are processed recursively.
- Each node is placed on the stack three times.
  - About to make a recursive call to the left subtree,
  - About to make a recursive call to the right subtree,
  - about to process the current node.
- The third time off, the node is declared visited. The other times we simulate a recursive call.
- When the stack is empty, every node has been visited.





The *PostOrder* class interface.

An *StNode* stores a pointer to a node and a count that tells how many times it has already been popped.

```
// PostOrder class interface; maintains "current position".
//
// CONSTRUCTION: with a tree to which the iterator is bound.
// ******************PUBLIC OPERATIONS***********
// bool isValid( )      --> True if at valid position in tree
// Object retrieve( )   --> Return item in current position
// void first( )        --> Set current position to first
// void advance( )      --> Advance
// ******************ERRORS***********************
// BadIterator is thrown for illegal access or advance.
template <class Object>
struct StNode
{  const BinaryNode<Object> *node;
   int timesPopped;
   StNode( const BinaryNode<Object> *n = 0 ) : node( n ), timesPopped( 0 ) { }
};

template <class Object>
class PostOrder : public TreeIterator<Object>
{public:
   PostOrder( const BinaryTree<Object> & theTree );
   ~PostOrder( ) { }
   void first( );
   void advance( );
 protected:
   Stack< StNode<Object> > s;  };
```


PostOrder class constructor.

```
// Constructor.
template <class Object>
PostOrder<Object>::PostOrder( const BinaryTree<Object> & theTree )
  : TreeIterator<Object>( theTree )
{
    s.push( StNode<Object>( root ) );
}
```

First( ) function.

```
// Set the current position to the first.
template <class Object>
void PostOrder<Object>::first( )
{
    s.makeEmpty( );
    if( root != NULL )
    {
        s.push( StNode<Object>( root ) );
        advance( );
    }
}
```


***advance( ) routine for the PostOrder iterator* class.**

```
// Advance to the next position.
// Throw BadIterator exception if the iteration has been exhausted prior to the call.
template <class Object>
void PostOrder<Object>::advance( )
{if( s.isEmpty( ) )
   {  if( current == NULL )
         throw BadIterator( "Advance past end" );
      current = NULL;
      return;   }

   StNode <Object> cnode;

   for( ; ; )
   {  cnode = s.topAndPop( );
      if( ++cnode.timesPopped == 3 )
      { current = cnode.node;
         return;   }
      s.push( cnode );
      if( cnode.timesPopped == 1 )
      {  if( cnode.node->left != NULL )
            s.push( StNode<Object>( cnode.node->left ) );  }
      else  // cnode.timesPopped == 2
      {  if( cnode.node->right != NULL )
            s.push( StNode<Object>( cnode.node->right ) ); }
   }
}
```


Other Traversals

- Inorder traversal is similar to postorder, except that a node is declared visited when it is popped for the second time.
- Preorder is the same as postorder, except that a node is declared visited the first time it is popped. The right and then left children are pushed prior to the return.
- In a level-order traversal, nodes are visited top to bottom, left to right.
  - Level-order traversal is implemented via a queue.
  - This is a *breadth first search*.