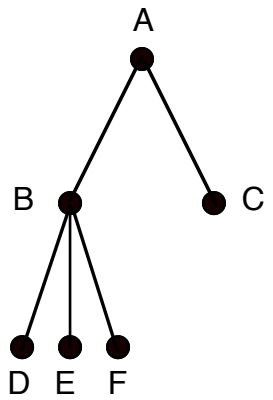
Chapter 10-1, Lecture notes

Terminology

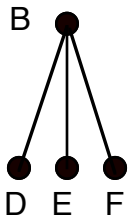
1. We had informally used tree diagrams to represent the calls of a recursive algorithm – like with the rabbit algorithm in chapter 2.
   1.1 In rabbit, each call was represented by a box or node, or "vertex" in the tree.
   1.2 The lines between the nodes are called "edges".
   1.3 For the rabbit tree, the edges indicated recursive calls.

   1.4 All trees are "hierarchical" in nature.

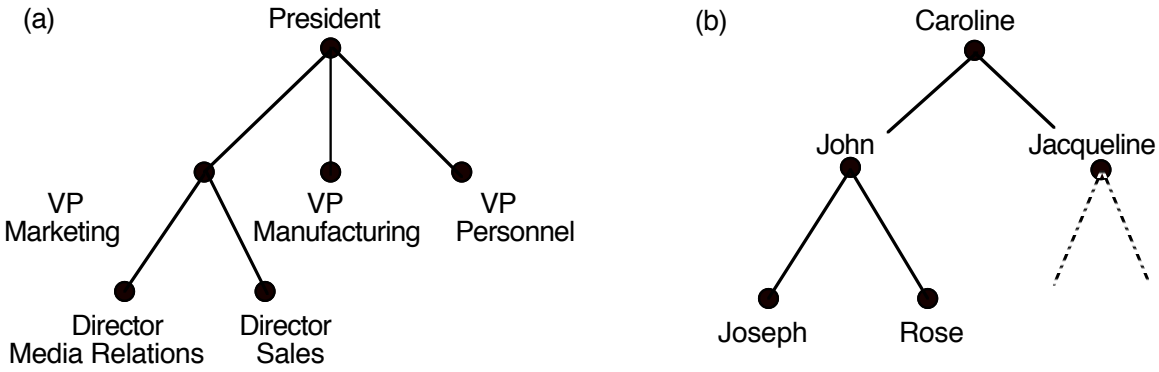
      1.4.1 That means a "parent-child" relationship exists between the nodes in the tree.
      1.4.2 If an edge is between node n and m then n is the parent of m and m is the child of n.
      1.4.3 In figure 10-1 has nodes B and C as the children of node A. B and C are "siblings".
      1.4.4 Each node in a tree has, at most, one parent and exactly one node that is the "root" of the tree.



   1.5 You can also call the parent-child relationship as the general "ancestor" and "descendant".
      1.5.1 In 10-1, A is an ancestor of D and D is a descendant of A.
      1.5.2 Though nodes B and C are not related by the ancestor and descendant relationship.
      1.5.3 The root is the ancestor of all nodes in the tree.
      1.5.4 A "subtree" would be any node in a tree with its descendants.
      1.5.5 So with node n, its subtree would be node n and its descendants, like with Figure 10-2 that shows node B with descendants D, E, and F:



2. Because trees are hierarchical in nature, you can use them to represent information that itself is hierarchical in nature – like with organization charts and family trees.
   2.1 Figure 10-3 has examples of hierarchical trees:



   2.2 With figure 10-3b, the family tree is upside down: i.e. Caroline's parents are John and Jacqueline – the "parents" are in "child" nodes (CS can be confusing).

   2.3 Formally, a "general tree T" is a set of one or more nodes such that T is partitioned into disjoint subsets:

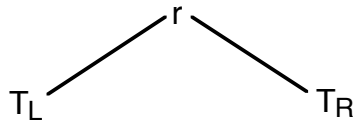
   - A single node r, the root
   - Sets that are general trees, called subtrees of r

   2.4 Figure 10-1 and 10-3a are general trees.
      2.4.1 However, we will concentrate on a "binary tree" is a set T of nodes such that either:

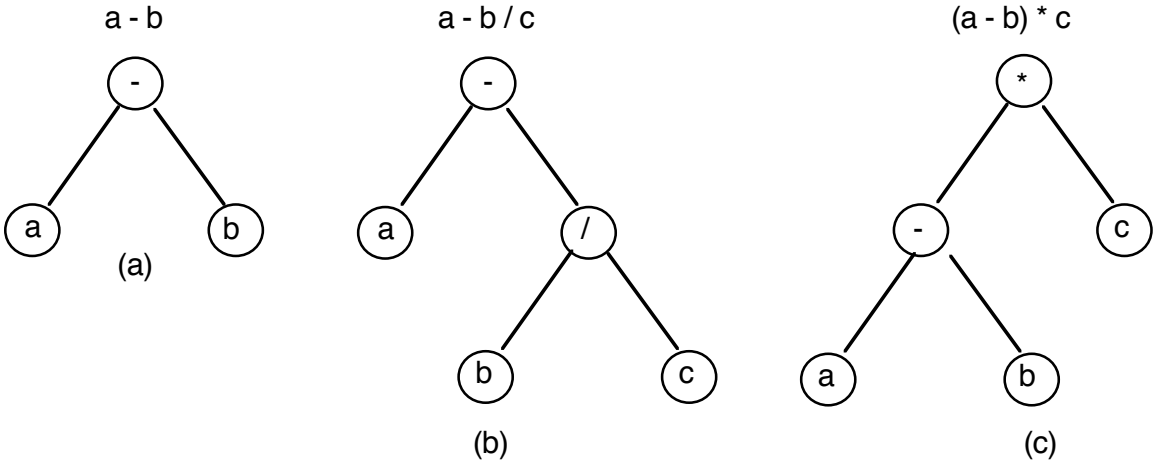
      - T is empty, or
      - T is partitioned into three disjoint subsets:
      - A single node r, the root
      - Two possibly empty sets that are binary trees, called "left" and "right subtrees" of r

      2.4.2 Trees in figures 2-11 and 10-3b are binary trees.
      2.4.3 A binary tree is NOT a special kind of general tree because a binary tree can be empty where a general tree cannot be.
      2.4.4 The binary tree T can be restated as:

- T has no nodes, or
- T is the form



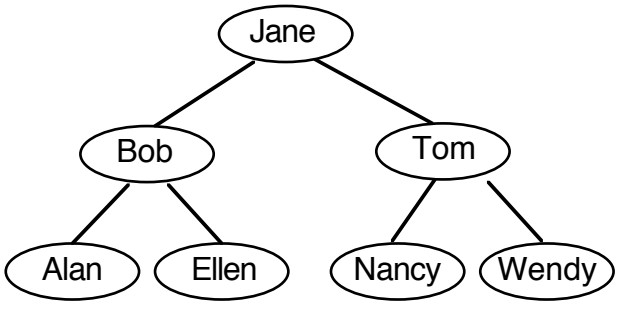
2.5 You can also use the binary tree to represent algebraic equations – that are hierarchical in nature.
  2.5.1 The parent node has higher priority than the child nodes.
  2.5.2 The figure 10-4 shows some examples:



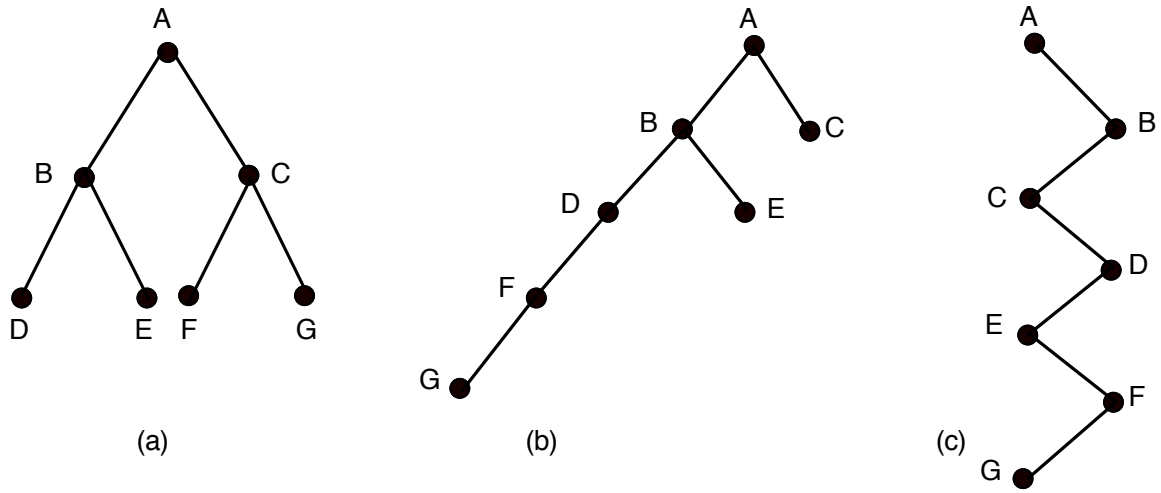
3. The nodes of a tree typically contain values like with a "binary search tree".
  3.1 For each node n, a binary search tree satisfies the following 3 properties:

- n's value is greater than all values in its left subtree T_L
- n's value is less than all values in its right subtree T_R
- Both T_L and T_R are binary search trees

  3.2 Figure 10-5 is an example of binary search tree:



4. Trees come in many shapes.
  4.1 Figure 10-6 has trees that have the same nodes but different structures like so:



4.2 The height of a tree is the number of nodes on the longest path from the root to a leaf node.

4.2.1 Figure 10-6 tree heights are (from a to c): 3, 5, and 7.

4.2.2 Some books would state the height as 2, 4, and 6 – use the number of edges instead of nodes.

4.2.3 The text book will use nodes.

4.2.4 Also another way to define height would be to define "level of a node n", like so:

- If n is the root of T, it is at level 1
- If n is not root of T, its level is 1 greater than the level of its parent

4.3 For example figure 10-6a, would have node A at level 1, node B at level 2, and node D at level 3.

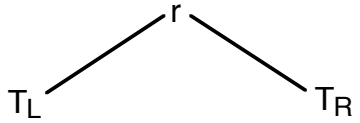
4.4 So we can define the height of a tree T as:

- If T is empty, its height is 0
- If T is not empty, its height is equal to the maximum level of its nodes

4.5 That definition allows the heights of 3, 5, and 7 for figure 10-6.

5. You can have a recursive definition of a binary tree like so:

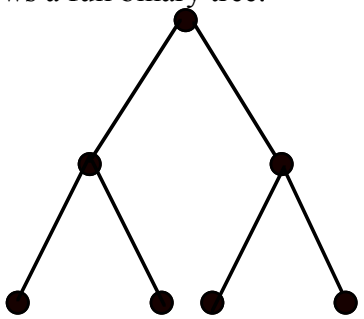
- If T is empty, its height is 0
- If T is a nonempty binary tree, then because T is of the form



5.1 the height of T is 1 greater than the height of its root's taller subtree; that is:

$$height(T) = 1 + \max\{height(T_L), height(T_R)\}$$

5.2 A “full binary tree” of height h, all nodes that are at a level less than h have two children each and all the subtrees have the same height.
  5.2.1 A full binary tree would have the maximum number of possible nodes.
  5.2.2 Figure 10-7 shows a full binary tree:

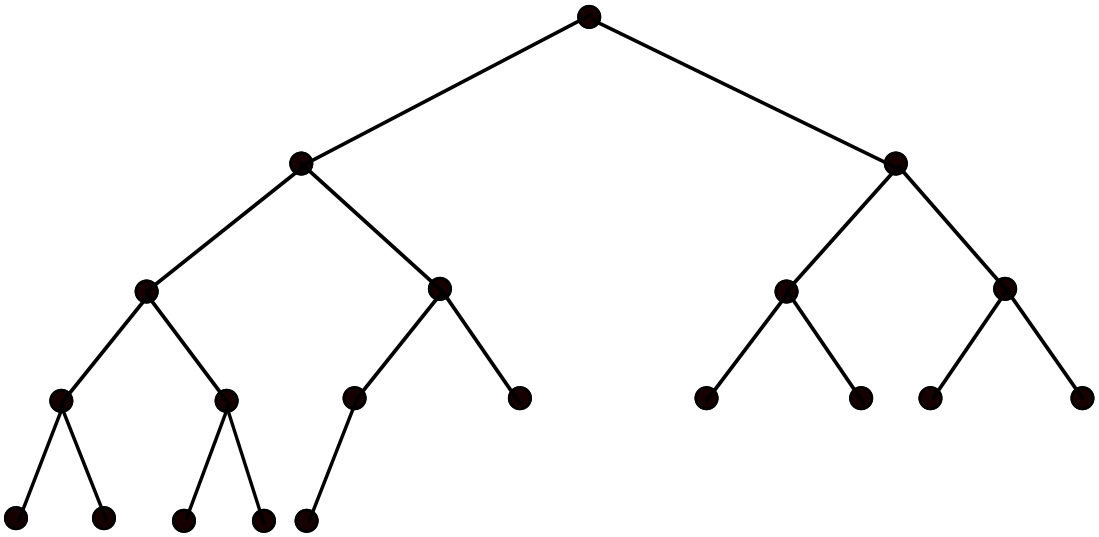


5.3 The following recursive definition of a full binary tree is:

- If T is empty, T is a full binary tree of height 0
- If T is not empty and has height > 0, T is a full binary tree if its root’s subtrees are both full binary trees of height h-1

5.4 The binary tree is recursive in nature.

5.5 A “complete binary tree” of height h is a binary tree that is full down to level h – 1, with level h filled in from left to right as Figure 10-8 shows:



6. More formal definition of a binary tree T of height h is complete if:

6.1 All nodes at level h – 2 and above have two children each, and
6.2 When a node at level h – 1 had children, all nodes to its left at the same level have two children each, and
6.3 When a node at level h – 1 has one child, it is a left child

6.4 A binary tree is “height balanced” or just “balanced” if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.

6.5 The binary trees in figure 10-8 and 10-6a are balanced but the figures 10-6b and 10-6c are not.

Chapter 10-2, The ADT Binary Tree

1. Let’s create an ADT for the binary tree.

1.1 Now with traversals of a binary tree, it would “visit” each node. The ADT binary tree has the following operations

1.1.1 Create an empty binary tree
1.1.2 Create a one-node binary tree, given an item
1.1.3 Create a binary tree, given an item for its root and two binary trees for the root’s subtrees
1.1.4 Destroy a binary tree
1.1.5 Determine whether a binary tree is empty
1.1.6 Determine or change the data in the binary tree’s root
1.1.7 Attach a left or right child to the binary tree’s root
1.1.8 Attach a left or right subtree of the binary tree’s root
1.1.9 Detach the left or right subtree of the binary tree’s root
1.1.10 Return a copy of the left or right subtree of the binary tree’s root
1.1.11 Traverse the nodes in a binary tree in preorder, inorder, or postorder

1.2 The operational contract for the ADT binary tree is:

TreeItemType is the type of the items stored in the binary tree

createBinaryTree(in rootItem:TreeItemType) throw TreeException
Creates a one-node binary tree whose root contains rootItem. Throws TreeException if allocation fails

createBinaryTree(in rootItem:TreeItemType, inout leftTree:BinaryTree,
inout rightTree:BinaryTree) throw TreeException
Creates a binary tree whose root contains rootItem and has leftTree and rightTree, respectively, as its left and right subtrees. Makes leftTree and rightTree empty so they cannot be used to gain access to the new tree. Throws TreeException if allocation fails

isEmpty():boolean{query}
Determines whether this binary tree is empty

getRootData():TreeItemType throw TreeException

Returns the data item in the root of a nonempty binary tree. Throws TreeException if the tree is empty

setRootData(in newItem:TreeItemType) throw TreeException

Replaces the data item in the root of this binary tree with newItem, if the tree is not empty. However, if the tree is empty, creates a root node whose data item is newItem and inserts the new node into the tree. If a new node cannot be created, TreeException is thrown

attachLeft(in newItem:TreeItem) throw TreeException

Attaches a left child containing newItem to the root of this binary tree. Throws TreeException if a new child node cannot be allocated. Also throws TreeException if the binary tree is empty (no root to attach to) or a left subtree already exists (should explicitly detach it first)

attachRight(in newItem:TreeItemType) throw TreeException

Attaches a right child containing newItem to the root of this binary tree. Throws TreeException if a new child node cannot be allocated. Also throws TreeException if the binary tree is empty (no root to attach to) or a right subtree already exists (should explicitly detach it first)

attachLeftSubtree(in leftTree:BinaryTree) throw TreeException

Attaches leftTree as the left subtree of the root of this binary tree and makes leftTree empty so that it cannot be used to access this tree. Throws TreeException if the binary tree is empty (no root to attach to) or a left subtree already exists (should explicitly detach it first)

attachRightSubtree(in righttTree:BinaryTree) throw TreeException

Attaches rightTree as the right subtree of the root of this binary tree and makes rightTree empty so that it cannot be used to access this tree. Throws TreeException if the binary tree is empty (no root to attach to) or a right subtree already exists (should explicitly detach it first)

detachLeftSubtree(out leftTree:BinaryTree) throw TreeException

Detaches the left subtree of this binary tree's root and retains it in leftTree. Throws TreeException if the binary tree is empty (no root to detach from)

detachRighttSubtree(out rightTree:BinaryTree) throw TreeException

Detaches the right subtree of this binary tree's root and retains it in rightTree. Throws TreeException if the binary tree is empty (no root to detach from)

getLeftSubtree():BinaryTree

Returns a copy of the left subtree of this binary tree's root without detaching the subtree. Returns an empty tree if the binary tree is empty (no root node to copy from).

getRightSubtree():BinaryTree

Returns a copy of the right subtree of this binary tree's root without detaching the subtree. Returns an empty tree if the binary tree is empty (no root node to copy from).

preorderTraverse(in visit:FunctionType)

Traverses this binary tree in preorder and calls the function visit() once for each node.

inorderTraverse(in visit:FunctionType)

Traverses this binary tree in inorder and calls the function visit() once for each node.

postorderTraverse(in visit:FunctionType)

Traverses this binary tree in postorder and calls the function visit() once for each node.

2. To build the tree in figure 10-6b where the nodes represent character data, do the following:

```
tree1.setRootData('F)
tree1.attachLeft('G')

tree2.setRootData('D')
tree2.attachLeftSubtree(tree1)

tree3.setRootData('B')
tree3.attachLeftSubtree(tree2)
tree3.attachRight('E')

tree4.setRootData('C)
// tree in Fig 10-6b
binTree.createBinaryTree('A', tree3, tree4)
```

2.1 Note that

```
binTree.getLeftSubtree()
```

2.2 is a copy of the subtree tree3 (rooted at 'B') and that:

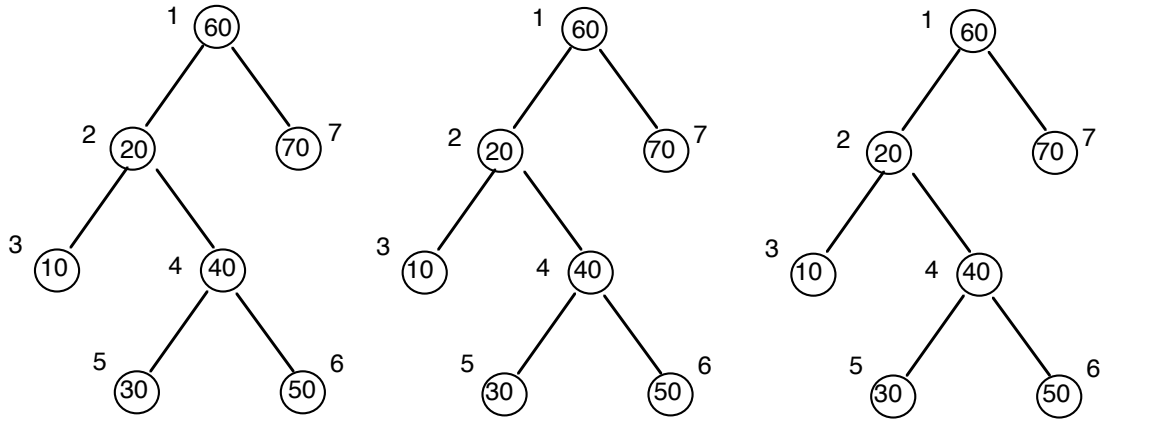
```
binTree.detachLeftSubtree(leftTree)
```

2.3 detaches that same subtree from binTree and names it leftTree.

3. The traversal algorithm is recursive in nature.
    3.1 When you "visit" a node, you may either change the value, print the value, or use the value in some way.
        3.1.1 If the tree T is empty, nothing is done.
        3.1.2 Otherwise the traversal algorithm looks like:

```
Traverse(in binTree:BinaryTree)
// Traverses the binary tree binTree
        if (binTree is not empty)
        {       traverse(Left subtree of binTree's root)
                traverse(Right subtree of binTree's root)
        }
```

    3.2 The above algorithm is not complete: it does not display the root value of the current subtree.
        3.2.1 It depends on where in the algorithm the root value is displayed.
        3.2.2 There is three basic traversals of: "preorder", "inorder", and "postorder".
        3.2.3 With preorder, the root of the subtree is visited first then left subtree and right subtree.
        3.2.4 With inorder, the left subtree is visited first, the root, then the right subtree.
        3.2.5 With postorder, the left subtree is visited first, the right subtree is visited, and then the root.
        3.2.6 Figure 10-10 shows the traversals:



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

    3.3 So the preorder is:

```
preorder(in binTree:BinaryTree)
// Traverses the binary tree binTree in preorder.
// Assumes that "visit a node" means to display the node's data item
        if (binTree is not empty)
        {       Display the data in the root of binTree
```

```
            preorder(Left subtree of binTree's root)
            preorder(Right subtree of binTree's root)
        } // end if
```

3.4 Figure 10-10 (a) has the values displayed in preorder.

3.5 The inorder traversal algorithm is:

```
inorder(in binTree:BinaryTree)
// Traverses the binary tree binTree in inorder.
// Assumes that "visit a node" means to display the node's data item
        if (binTree is not empty)
        {       inorder(Left subtree of binTree's root)
                Display the data in the root of binTree
                inorder(Right subtree of binTree's root)
        } // end if
```

3.6 Figure 10-10 (b) has the values displayed in inorder.

3.7 The postorder traversal algorithm is:

```
postorder(in binTree:BinaryTree)
// Traverses the binary tree binTree in postorder.
// Assumes that "visit a node" means to display the node's data item
        if (binTree is not empty)
        {       postorder(Left subtree of binTree's root)
                postorder(Right subtree of binTree's root)
                Display the data in the root of binTree

        } // end if
```

3.8 Figure 10-10 (c) has the values displayed in postorder.

3.9 You can do different traversal operations for different tasks.

4. You can implement a binary tree three general ways.

4.1 Two of them use arrays but the most typical is to use pointers.

4.1.1 We'll look at the array-based representation first.

4.1.2 With array based would have a data portion and indexes to the left child and right child, like so:

```
const int MAX_NODES = 100;        // maximum number of nodes
typedef string TreeItemType;

class TreeNode                    // node in the tree
{
private:
        TreeNode();
        TreeNode(const TreeItemType& nodeItem, int left, int right);
```

```
        TreeItemType item;          // data portion
        int          leftChild;     // index to left child
        int          rightChild;    // index to right child

        // friend class – can access private data
        friend class BinaryTree;
}; // end TreeNode

TreeNode    tree[MAX_NODES]; // array of tree nodes
int         root;            // index of root
int         free;            // index of free list
```

4.2 The class Treenode has all its member private but declares the binary tree class BinaryTree is a friend class so that members of BinaryTree have direct access to all nodes of TreeNode.

4.2.1 Variable root is the index to the tree's root within the array "tree".

4.2.2 If the root is -1 then the tree is empty.

4.2.3 An index of -1 means NULL. Both leftChild and rightChild are indexes to the children of the node.

4.3 As the tree changes due to insertions and deletions, the nodes may not be contiguous – hence the list of available nodes called a "free list".

4.3.1 To insert a new node into the tree, you get an available node from the free list

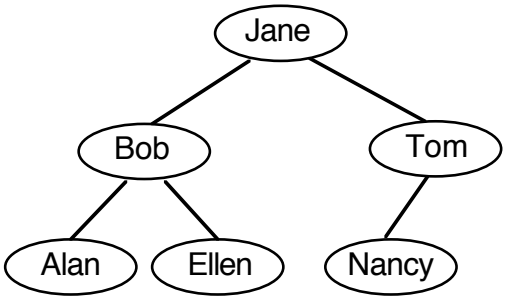
4.3.2 If you delete a node from the tree, you place it back into the free list so that it can be used again.

4.3.3 The variable "free" is the index to the first node in the free list and , arbitrarily, the rightChild of each node in the free list would point to the next node.

4.4 So if the root is the index of the root r of a binary tree, the tree[root].leftChild points to the left subtree and the tree[root].rightChild is the right subtree.

4.4.1 Figure 10-11 shows the binary tree and the implementation:

(a)



(b) tree

	item	leftChild	rightChild
0	Jane	1	2
1	Bob	3	4
2	Tom	5	-1
3	Alan	-1	-1
4	Ellen	-1	-1
5	Nancy	-1	-1
6	?	-1	7
7	?	-1	8
8	?	-1	9
.	.	.	.
.	.	.	.
.	.	.	.

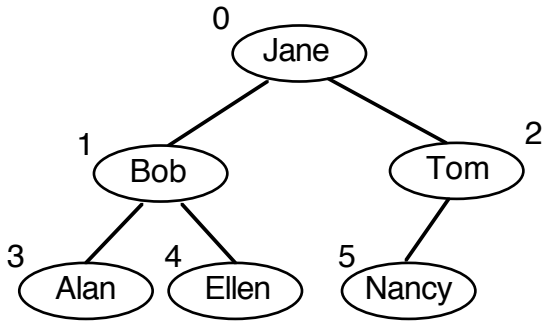
root: 0

free: 6

4.4.2 Now if you know you are going to implement a complete binary tree, you can use a simpler array-based implementation that saves memory.

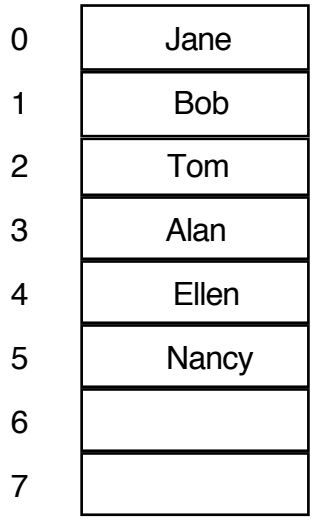
4.4.3 A complete binary tree of height h would have all the h-1 level completely filled from left to right.

4.4.4 Figure 10-12 shows the complete binary tree with the node numbered left to right:



4.5 If you place the nodes into the array tree in numeric order, you can find the left and right child (if they exist) by calculating the index.

4.5.1 The left child of node i would be tree[2*i + 1], the right child node i would be tree[2*i + 2] and the parent would be tree[(i -1) / 2] as shown in Figure 10-13:



4.5.2 If the tree was NOT a complete binary tree, it could become ambiguous.

5. The pointer-based representation would look like:

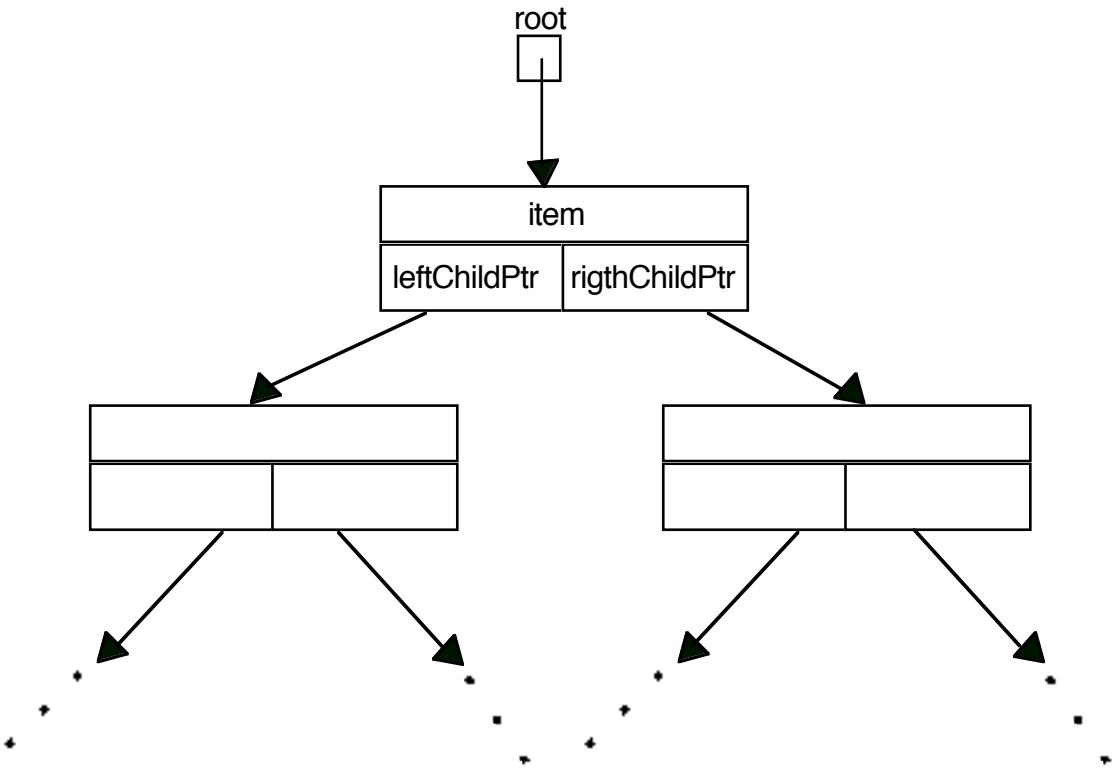
```
typedef string TreeItemType;

class TreeNode          // node in the tree
{
private:
        TreeNode() {}
        TreeNode(const TreeItemType& nodeItem,
                        TreeNode *left = NULL, TreeNode*right = NULL)
                : item(nodeItem), leftChildPtr(left), rightChildPtr(right) {}
        TreeItemType item;              // data portion
        TreeNode *leftChildPtr;         // pointer to left child
        TreeNode *rightChildPtr;        // pointer to right child
        Friend class BinaryTree;
}; // end TreeNode
```

5.1 The external pointer root points to the tree's root.

5.1.1 If the tree is empty, root is NULL.

5.1.2 Figure 10-14 shows pointer-based implementation of a binary tree.

5.1.3 The root for binary tree of r would have pointers to the left and right subtrees of root->leftChildPtr and root->rightChildPtr:



5.2 The pointer-based implementation of the ADT Binary Tree starts as:

```
/** @file TreeException.h */

#include <stdexcept>
#include <string>

using namespace std;

/** Exception class for the ADT binary tree */
class TreeException : public logic_error
{
public:
        TreeException(const string& message = " ") : logic_error(message.c_str())
        { } // end constructor
}; // end TreeException
```

Then the binary header file:

```
/** @file BinaryTree.h */

#include "TreeException.h"
#include "TreeNode.h"          // contains definitions for TreeNode and TreeItemType
```

```
typedef void (*FunctionType)(TreeItemType& anItem);

/** ADT binary tree */
class BinaryTree
{
public:
// constructors and destructor:
        BinaryTree();
        BinaryTree(const TreeItemType& rootItem) throw(TreeException);
        BinaryTree(const TreeItemType& rootItem, BinaryTree& leftTree,
                        BinaryTree& rightTree) throw(TreeException);
        BinaryTree(const BinaryTree& tree) throw(TreeException);
        virtual ~BinaryTree();

// binary tree operations:
        virtual bool isEmpty() const;

        virtual TreeItemType getRootData() const throw(TreeException);
        virtual void setRootData(const TreeItemType& newItem)
                throw(TreeException);

        virtual void attachLeft(const TreeItemType& newItem)
                throw(TreeException);
        virtual void attachRight(const TreeItemType& newItem)
                throw(TreeException);

        virtual void attachLeftSubtree(BinaryTree& leftTree)
                throw(TreeException);
        virtual void attachRightSubtree(BinaryTree& rightTree)
                throw(TreeException);

        virtual void detachLeftSubtree(BinaryTree& leftTree)
                throw(TreeException);
        virtual void detachRightSubtree(BinaryTree& rightTree)
                throw(TreeException);

        virtual BinaryTree getLeftSubTree() const;
        virtual BinaryTree getRightSubTree() const;

        virtual void preorderTraverse(FunctionType visit);
        virtual void inorderTraverse(FunctionType visit);
        virtual void postorderTraverse(FunctionType visit);

// overloaded operator:
        virtual BinaryTree& operator=(const BinaryTree& rhs)
                throw(TreeException);
```

```
protected:
        BinaryTree(TreeNode *nodePtr);      // constructor

        /** Copies the tree rooted at treePtr into a tree rooted at newTreePtr
        * @throw TreeException    If a copy of the tree cannot be allocated */
        void copyTree(TreeNode *treePtr), TreeNode *& newTyreePtr)
                        const throw(TreeException);

        /** Deallocates memory for a tree */
        void destroyTree(TreeNode *& treePtr);

        // The next two methods retrieve and set the value or the private data
        // member root

        TreeNode *rootPtr() const;
        void setRootPtr(TreeNode *newRoot);

        // Next two methods retrieve and set the values of left and right child
        // pointers of a tree node

        void getChildPtrs(TreeNode *nodePtr, TreeNode *& leftChildPtr,
                                TreeNode *& rightChildPtr) const;
        void setChildPtrs(TreeNode *nodePtr, TreeNode *& leftChildPtr,
                                TreeNode *& rightChildPtr);

        void preorder(TreeNode *treePtr, FunctionType visit);
        void inorder(TreeNode *treePtr, FunctionType visit);
        void postorder(TreeNode *treePtr, FunctionType visit);

private:
        TreeNode *root;         // Pointer to root of tree
}; // end BinaryTree
// End of header file
```

The implementation file would be:

```
/** @file BinaryTree.cpp */

#include <cstddef>              // definition of NULL
#include <new>                  // for bad_alloc
#include "BinaryTree.h"         // header file

using namespace std;

BinaryTree::BinaryTree() : root(NULL)
```

```
{
} // end default constructor

BinaryTree::BinaryTree(const TreeItemType& rootItem) throw(TreeException)
{
	try
	{
		root = new TreeNode(rootItem, NULL, NULL);
	}
	catch (bad_alloc e)
	{
		delete root;
		throw TreeException(
			"TreeException: constructor cannot allocate memory");
	} // end try
} // end constructor

BinaryTree::BinaryTree(const TreeItemType& rootItem, BinaryTree& leftTree,
				BinaryTree& rightTree) throw(TreeException)
{
	try
	{
		root = new TreeNode(rootItem, NULL, NULL)

		attachLeftSubtree(leftTree);
		attachRightSubtree(rightTree);
	}
	catch (bad_alloc e)
	{
		delete root;
		throw TreeException(
			"TreeException: constructor cannot allocate memory");
	} // end try
} // end constructor

BinaryTree:BinaryTree(const BinaryTree& tree) throw(TreeException)
{
	try
	{
		copyTree(tree.root, root);
	}
	catch (bad_alloc e)
	{
		destroyTree(tree.root);
		throw TreeException(
			"TreeException: copy constructor cannot allocate memory");
```

```
        } // end try
} // end copy constructor

BinaryTree::BinaryTree(TreeNode *nodePtr) : root(nodePtr)
{
} // end protected constructor

bool BinaryTree::~BinaryTree()
{
        destroyTree(root);
}

bool BinaryTree::isEmpty() const
{
        return (root==NULL);
} // end isEmpty

TreeItemType BinaryTree::getRootData() const throw(TreeException)
{
        if (isEmpty())
                throw TreeException("TreeException: Empty tree");
        return root->tree;
} // end getRootData

void BinaryTree::setRootData(const TreeItemType& newITem)
                        throw(TreeException)
{
        if (!isEmpty())
                root->item = newItem;
        else
        {
                try
                {
                        root = new TreeNode(newItem, NULL, NULL);
                }
                catch (bad_alloc e)
                {
                        throw TreeException(
                                "TreeException: setRootData cannot allocate memory");
                } // end try
        } // end if
} // end setRootData

void BinaryTree::attachLeft(const TreeItemType& newItem) throw(TreeException)
{
        if (isEmpty())
```

```
            throw TreeException("TreeException: Empty tree");
        else if (root->leftChildPtr != NULL)
            throw TreeException(
                "TreeException: Cannot override left subtree");
        else // Assertion: nonempty tree; no left child
        {
            try
            {
                root->leftChildPtr = new TreeNode(newItem, NULL, NULL);
            }
            catch (bad_alloc e)
            {
                throw TreeException(
                    "TreeException: attachLeft cannot allocate memory");
            } // end try
        } // end if
} // end attachLeft

void BinaryTree::attachRight(const TreeItemType& newItem) throw(TreeException)
{
        if (isEmpty())
            throw TreeException("TreeException: Empty tree");
        else if (root->rightChildPtr != NULL)
            throw TreeException(
                "TreeException: Cannot override right subtree");
        else // Assertion: nonempty tree; no right child
        {
            try
            {
                root->rightChildPtr = new TreeNode(newItem, NULL, NULL);
            }
            catch (bad_alloc e)
            {
                throw TreeException(
                    "TreeException: attachRight cannot allocate memory");
            } // end try
        } // end if
} // end attachRight

void BinaryTree::attachLeftSubTree(BinaryTree& leftTree) throw(TreeException)
{
        if (isEmtpy())
            throw TreeException("TreeException: Empty tree");
        else if (root->leftChildPtr != NULL)
            throw TreeException(
                "TreeException: Cannot overwrite left subtree");
```

```
        else // Assertion: nonempty tree; no left child
        {       root->leftChildPtr = leftTree.root;
                leftTree.root = NULL;
        } // end if
} // end attachLeftSubtree

void BinaryTree::attachRightSubTree(BinaryTree& rightTree) throw(TreeException)
{
        if (isEmtpy())
                throw TreeException("TreeException: Empty tree");
        else if (root->rightChildPtr != NULL)
                throw TreeException(
                        "TreeException: Cannot overwrite left subtree");
        else // Assertion: nonempty tree; no left child
        {       root->rightChildPtr = rightTree.root;
                rightTree.root = NULL;
        } // end if
} // end attachRightSubtree

void BinaryTree::detachLeftSubtree(BinaryTree& leftTree) throw(TreeException)
{
        if (isEmpty())
                throw TreeException("TreeException: Emtpy tree");
        else
        {       leftTree = BinaryTree(root->leftChildPtr);
                root->leftChildPtr = NULL
        } // end if
} // end detachLeftSubtree

void BinaryTree::detachRightSubtree(BinaryTree& rightTree) throw(TreeException)
{
        if (isEmpty())
                throw TreeException("TreeException: Emtpy tree");
        else
        {       rightTree = BinaryTree(root->rightChildPtr);
                root->rightChildPtr = NULL
        } // end if
} // end detachRightSubtree

BinaryTree BinaryTree::getLeftSubtree() const throw(TreeException)
{
        TreeNode *subTreePtr;

        if (isEmpty())
                return BinaryTree();
        else
```

```
        {       copyTree(root->leftChildPtr, subTreePtr);
                return BinaryTree(subTreePtr);
        } // end if
} // end getLeftSubtree

BinaryTree BinaryTree::getRightSubtree() const throw(TreeException)
{
        TreeNode *subTreePtr;

        if (isEmpty())
                return BinaryTree();
        else
        {       copyTree(root->rightChildPtr, subTreePtr);
                return BinaryTree(subTreePtr);
        } // end if
} // end getRightSubtree

void BinaryTree::preorderTraverse(FunctionType visit)
{
        preorder(root, visit);
} // end preorderTraverse

void BinaryTree::inorderTraverse(FunctionType visit)
{
        inorder(root, visit);
} // end inorderTraverse

void BinaryTree::postorderTraverse(FunctionType visit)
{
        postorder(root, visit);
} // end postorderTraverse

BinaryTree& BinaryTree::operator=(const BinaryTree& rhs) throw(TreeException)
{
        if (this != &rhs)
        {       destroyTree(root);              // deallocate left-hand side
                copyTree(rhs.root, root);       // copy right-hand side
        } // end if
        return *this;
} // end operator==

void BinaryTree::copyTree(TreeNode *treePtr, TreeNode *& newTreePtr) const
                        throw(TreeException)
{
        // preorder traversal
        if (treePtr != NULL)
```

```
        {       // copy node
                try
                {
                        newTreePtr = new TreeNode(treePtr->item, NULL, NULL);
                        copyTree(treePtr->leftChildPtr, newTreePtr->leftChildPtr);
                        copyTree(treePtr->rightChildPtr, newTreePtr->rightChildPtr);
                }
                catch (bad_alloc e)
                {
                        throw TreeException(
                                "TreeException: copyTree cannot allocate memory");
                } // end try
        }
        else
                newTreePtr = NULL; // copy empty tree
} // end copyTree

void BinaryTree::destroyTree(TreeNode *& treePtr)
{
        // postorder traversal
        if (treePtr != NULL)
        {       destroyTree(treePtr->leftChildPtr);
                destroyTree(treePtr->rightChildPtr);
                delete treePtr;
                treePtr = NULL;
        } // end if
} // end destroyTree

TreeNode *BinaryTree::rootPtr() const
{
        return root;
} // end rootPtr

void BinaryTree::setRootPtr(TreeNode *newRoot)
{
        root = newRoot;
} // end setRootPtr

void BinaryTree::getChildPtrs(TreeNode *nodePtr,
                        TreeNode *& leftPtr, TreeNode *&rightPtr) const
{
        leftPtr = nodePtr->leftChildPtr;
        rightPtr = nodePtr->rightChildPtr;
} // end getChildPtrs

void BinaryTree::setChildPtrs(TreeNode *nodePtr,
```

```
                        TreeNode *&leftPtr, TreeNode *&rightPtr)
{
        nodePtr->leftChildPtr = leftPtr;
        nodePtr->rightChildPtr = rightPtr;
} // end setChildPtrs

void BinaryTree::preorder(TreeNode *treePtr, FunctionType visit)
{
        if (treePtr != NULL)
        {       visit(treePtr->item);
                preorder(treePtr->leftChildPtr, visit);
                preorder(treePtr->rightChildPtr, visit);
        } // end if
} // end preorder

void BinaryTree::inorder(TreeNode *treePtr, FunctionType visit)
{
        if (treePtr != NULL)
        {       inorder(treePtr->leftChildPtr, visit);
                visit(treePtr->item);
                inorder(treePtr->rightChildPtr, visit);
        } // end if
} // end inorder

void BinaryTree::postorder(TreeNode *treePtr, FunctionType visit)
{
        if (treePtr != NULL)
        {       postorder(treePtr->leftChildPtr, visit);
                postorder(treePtr->rightChildPtr, visit);
                visit(treePtr->item);
        } // end if
} // end postorder
```

6. The class BinaryTree has more constructors than usual.
   6.1 That allows you to define a binary tree:
   - That is empty
   - From data for its root, which is its only node
   - From data for its root and the root's two subtrees.

   6.2 For example:

```
BinaryTree tree1;
BinaryTree tree2(root2);
BinaryTree tree3(root3);
BinaryTree tree4(root4, tree2, tree3);
```

6.3 The tree1 is an empty binary tree; tree2 and tree3 have only root nodes (root2 and root3); and tree4 is a binary tree whose root contains root4 and has subtrees tree2 and tree3.

6.4 Note that tree2 and tree3 are BinaryTree and not pointers to trees.

6.5 There is also a constructor with a pointer like so:

```
BinaryTree tree5(nodePtr);
```

6.5.1 The methods of getLeftSubtree and getRightSubtree use this constructor but it should not be public because of the pointer.

6.5.2 It has been declared protected so that a derived class can use it.

6.6 Also, the following should not be public either because of pointers:

```
void inorder(TreeNode *treePtr, FunctionType visit);
```

6.6.1 The pubic function of "inorderTraverse" should then call the inorder function so that the ADT wall is not violated.

6.6.2 Now the type of:

```
typedef void (*FunctionType) (TreeItemType& anItem);
```

6.7 That defines a pointer to a function so that the traversal functions can have a function passed to it for the "visit" (makes the traversals more general).

6.7.1 The parameter of "anItem" is a reference to allow the visit to change it.

6.7.2 So if you define a visit function like so:

```
void display(TreeItemType& anItem);
```

6.7.3 You can use it like so:

```
#include <iostream>
#include "BinaryTree.h" // binary tree operations

using namespace std;

void display(TreeItemType& anItem);

int main()
{
        BinaryTree tree1, tree2, tree3, left;    // empty trees
        BinaryTree tree3(70);                    // tree with only a root 70
```

```
// build the tree in figure 10-10
        tree1.setRootData(40);
        tree1.attachLeft(30);
        tree1.attachRight(50)_;

        tree2,setRootData(20);
        tree2.attachLeft(10);
        tree2.attachRightSubtree(tree1);

        // tree in fig 10-10
        BinaryTree binTree(60, tree2, tree3);
        binTree.inorderTraverse(display);
        binTree.getLeftSubtree().inorderTraverse(display);
        binTree.detachLeftSubtree(left);
        left.inorderTraverse(display);
        binTree.inorderTraverse(display);
        return 0;
} // end main
```

6.8 And you get the tree in figure 10-10.

6.9 There is an optional section with non-recursive traversal. We'll skip that.