Chapter 20

Binary Trees

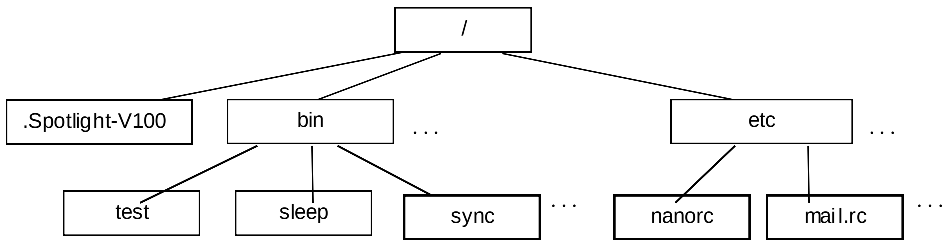
The data structures presented so far are predominantly linear. Every element has one unique predecessor and one unique successor (except the first and last elements). Arrays, and singly linked structures used to implement lists, stacks, and queues all have this linear characteristic. The **tree** structure presented in this chapter is a hierarchical in that nodes may have more than one successor.

Goals

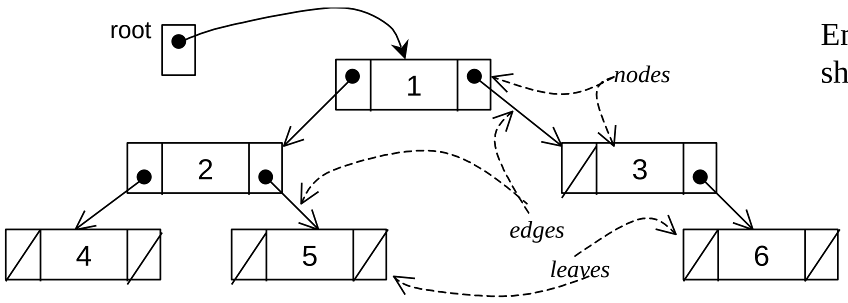
- Become familiar with tree terminology and some uses of trees
- Store data in a hierarchical data structure as a Java Collection class
- Implement binary tree algorithms
- Implement algorithms for a Binary Search Tree

20.1 Trees

Trees are often used to store large collections of data in a hierarchical manner where elements are arranged in successive levels. For example, file systems are implemented as a tree structure with the root directory at the highest level. The collection of files and directories are stored as a tree where a directory may have files and other directories. Trees are hierarchical in nature as illustrated in this view of a very small part of a file system (the root directory is signified as /).



Each node in a tree has exactly one parent except for the distinctive node known as the **root**. Whereas the root of a real tree is usually located in the ground and the leaves are above the root, computer scientists draw trees upside down. This convention allows us to grow trees down from the root since most people find it more natural to write from top to bottom. You are more likely to see the root at the 'top' with the leaves at the 'bottom' of trees. Trees implemented with a linked structure can also be pictured like this:



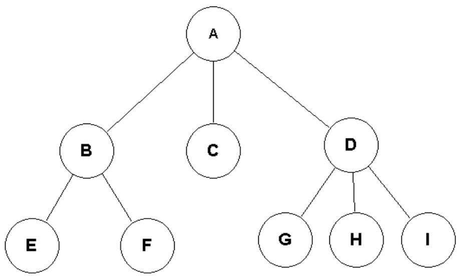
Empty trees are shown here as /

A nonempty tree is a collection of nodes with one node designated as the **root**. Each **node** contains a reference to an element and has **edges** connecting it to other nodes, which are also trees. These other nodes are called children. A tree can be empty — have no nodes. Trees may have nodes with two or more children.

A leaf is a node with no children. In the tree above, the nodes with 4, 5, and 6 are leafs. All nodes that are not leaves are called the internal nodes of a tree, which are 1, 2, and 3 above. A leaf node could later grow a nonempty tree as a child. That leaf node would then become an internal node. Also, an internal node might later have its children become empty trees. That internal node would become a leaf.

A tree with no nodes is called an empty tree. A single node by itself can be considered a tree. A structure formed by taking a node N and one or more separate trees and making N the parent of all roots of the trees is also a tree. This recursive definition enables us to construct trees from existing trees. After the construction, the new tree would contain the old trees as subtrees. A subtree is a tree by itself. By definition, the empty tree can also be considered a subtree of every tree.

All nodes with the same parent are called siblings. The level of a node is the number of edges it takes to reach that particular node from the root. For example, the node in the tree above containing J is at level 2. The height of a tree is the level of the node furthest away from its root. These definitions are summarized with a different tree where the letters A through I represent the elements.

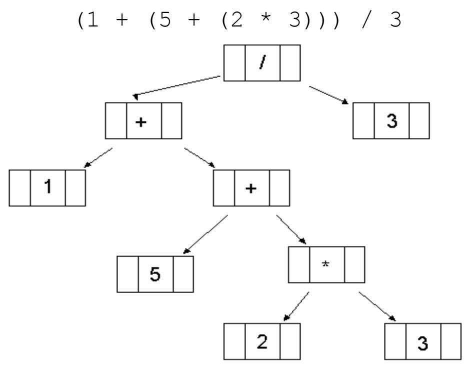


root	A
internal nodes	A B D
leaves	E F C G H I
height	2
level of root	0
level of node with B	1
level of node with E	2
nodes at level 1	3
parent of G, H and I	D
children of B	E F

A **binary tree** is a tree where each node has exactly two binary trees, commonly referred to as the left child and right child. Both the left or right trees are also binary trees. They could be empty trees. When both children are empty trees, the node is considered a leaf. Under good circumstances, binary trees have the property that you can reach any node in the tree within $\log_2 n$ steps, where n is the number of nodes in the tree.

Expression Tree

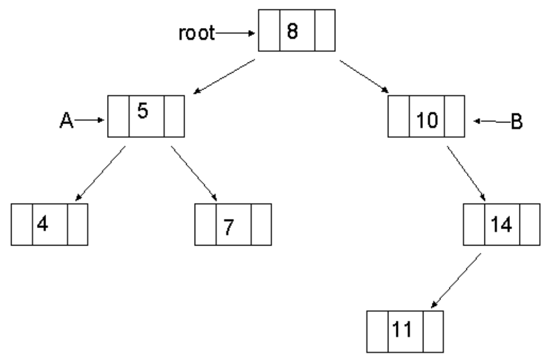
An **expression tree** is a binary tree that stores an arithmetic expression. The tree can then be traversed to evaluate the expression. The following expression is represented as a binary tree with operands as the leaves and operators as internal nodes.



Depending on how you want to traverse this tree — visit each node once — you could come up with different orderings of the same expression: infix, prefix, or postfix. These tree traversal algorithms are presented later in this chapter.

Binary Search Tree

Binary Search Trees are binary trees with the nodes arranged according to a specific ordering property. For example, consider a binary search tree that stores `Integer` elements. At each node, the value in the left child is less than the value of the parent. The right child has a value that is greater than the value of its parent. Also, since the left and right children of every node are binary search trees, the same ordering holds for all nodes. For example, all values in the left subtree will be less than the value in the parent. All values in the right subtree will be greater than the value of the parent.



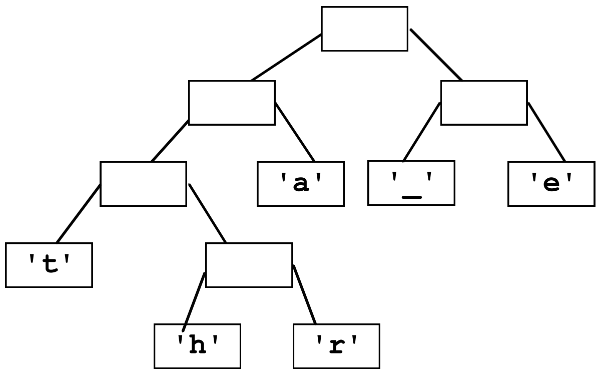
The left child of the root (referenced by **A**) has a value (5) that is less than the value of the root (8). Likewise, the value of the right child of the root has a value (10) that is greater than the root's value (8). Also, all the values in the subtree referenced by **A** (4, 5, 7), are less than the value in the root (8).

To find the node with the value 10 in a binary search tree, the search begins at the root. If the search value (10) is greater than the element in the root node, search the binary search tree to the right. Since the right tree has the value you are looking for, the search is successful. If the key is further down the tree, the search keeps going left or right until the key is found or the subtree is empty indicating the key was not in the BST. Searching a binary search tree can be O(log n) since half the nodes are removed from the search at each comparison. Binary search trees store large amounts of real world data because of their fast searching, insertions, and removal capabilities. The binary search tree will be explored later in this chapter.

Huffman Tree

David Huffman designed one of the first compression algorithms in 1952. In general, the more frequently occurring symbols have the shorter encodings. Huffman coding is an integral part of the standards for high definition television (HDTV). The same approach to have the most frequently occurring characters in a text file be represented by shorter codes, allows a file to be compressed to consume less disk space and to take less time to arrive over the Internet.

Part of the compression algorithm involves creation of a Huffman tree that stores all characters in the file as leaves in a tree. The most frequently occurring letters will have the shortest paths in the binary tree. The least occurring characters will have longer paths. For example, assuming a text file contains only the characters 'a', 'e', 'h', 'r', 't', and '_', the Huffman tree could look like this assuming that 'a', 'e', and '_' occur more frequently than 'h' and 'r'.



With the convention that 0 means go left and 1 right, the 6 letters have the following codes:

'a'	01
'_'	10
'e'	11
't'	000
'h'	0010
'r'	0011

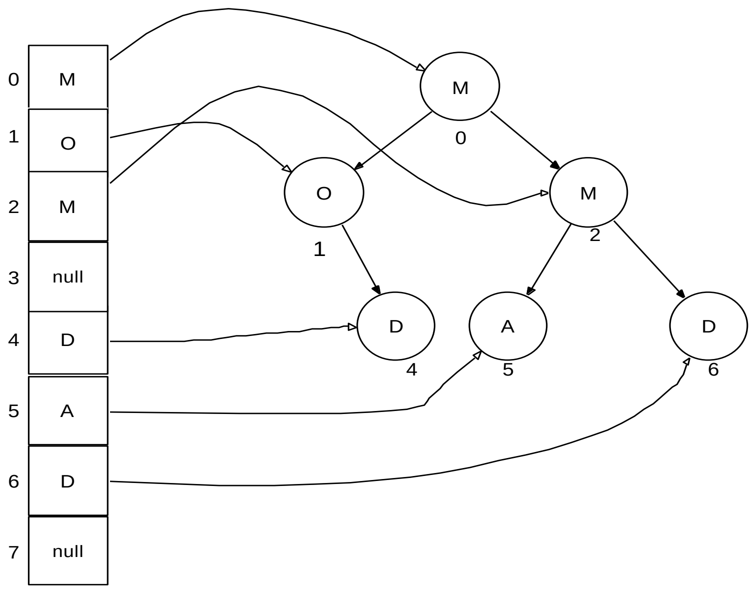
Instead of storing 8 bits for each character, the most frequently occurring letters in this example use only 2 or 3 bits. Some of the characters in a typical file would have codes for some characters that are much longer than 8 bits. These 31 bits represent a text file containing the text "tea_at_three".

```
0001101100100010000001000111111
  | | | | |  | |  |   |   | | |
  t e a _ a  t _  t   h   r e e
```

Assuming 8 bit ASCII characters, these 31 bits would require 12*8 or 96 bits.

20.2 Implementing Binary Trees

A binary tree can be represented in an array. With an array-based implementation, the root node will always be positioned at array index 0. The root's left child will be positioned at index 1, and the right child will be positioned at array index 2. This basic scheme can be carried out for each successive node counting up by one, and spanning the tree from left to right on a level-wise basis.



Notice that some nodes are not used. These unused array locations show the "holes" in the tree. For example, nodes at indexes 3 and 7 do not appear in the tree and thus have the `null` value in the array. In order to find any left or right child for a node, all that is needed is the node's index. For instance to find node 2's left and right children, use the following formula:

```
Left Child's Index   =  2 * Parent's Index + 1
Right Child's Index  =  2 * Parent's Index + 2
```

So in this case, node 2's left and right children have indexes of 5 and 6 respectively. Another benefit of using an array is that you can quickly find a node's parent with this formula:

```
Parent's Index = (Child's Index – 1) / 2
```

For example, (5-1)/2 and (6-1)/2 both have the same parent in index 2. This works, because with integer division, 4/2 equals 5/2.

Linked Implementation

Binary trees are often implemented as a linked structure. Whereas nodes in a singly linked structure had one reference field to refer to the successor element, a `TreeNode` will have two references — one to the left child and one to the right child. A tree is a collection of nodes with a particular node chosen as the root. Assume the `TreeNode` class will be an inner class with private instance variables that store these three fields

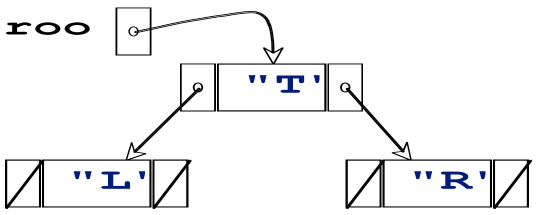
- a reference to the element
- a reference to a left tree (another `TreeNode`),
- a reference to a right tree (another `TreeNode`).

To keep things simple, the `TreeNode` class begins like this so it can store only strings. There are no generics (Chapter 12 will show a generic binary tree).

```
  // A type to store an element and a reference to two other TreeNode objects
  private class TreeNode {

    private String data;
    private TreeNode left;
    private TreeNode right;

    public TreeNode(String elementReference) {
      data = elementReference;
      left = null;
      right = null;
    }
  }
```

The following three lines of code (if in the same class as this inner node class) will generate the binary tree structure shown:

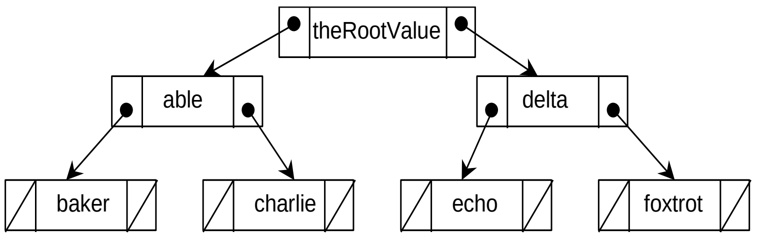
```
TreeNode root = new TreeNode("T");
root.left = new TreeNode("L");
root.right = new TreeNode("R");
```



Self-Check

20-1 Using the tree shown below, identify

a) the root	c) the leaves	e) the children of delta
b) size	d) the internal nodes	f) the number of nodes on level 4



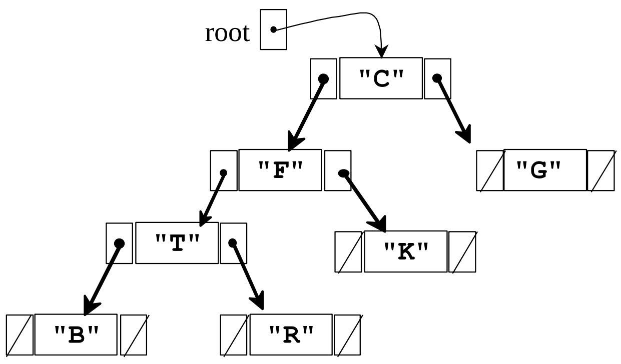
20-2 Using the `TreeNode` class above, write the code that generates the tree above.

Node as an Inner Class

Like the node classes of previous collections, this `TreeNode` class can also be placed inside another. However, instead of a collection class with an insert method, `hardCodeATree` will be used here to create a small binary tree. This will be the tree used to present several binary tree algorithms such as tree traversals in the section that follows.

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method. Instead a tree must be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinaryTreeOfStrings {

 private class TreeNode {

   private String data;
   private TreeNode left;
   private TreeNode right;

   public TreeNode(String elementReference) {
     data = elementReference;
      left = null;
      right = null;
    }
  }

  // The entry point into the tree
  private TreeNode root;

  // Construct and empty tree
  public BinaryTreeOfStrings() {
    root = null;
  }

  // Hard code a tree of size 6 on 4 levels
  public void hardCodeATree() {
    root = new TreeNode("C");
    root.left = new TreeNode("F");
    root.left.left = new TreeNode("T");
    root.left.left.left = new TreeNode("B");
    root.left.left.right = new TreeNode("R");
    root.left.right = new TreeNode("K");
    root.right = new TreeNode("G");
  }
}
```

The tree built in `hardCodeATree()`



20.3 Binary Tree Traversals

Code that traverses a linked list would likely visit the nodes in sequence, from the first element to the last. Thus, if the list were sorted in a natural ordering, nodes would be visited in from smallest to largest. With binary trees, the traversal is quite different. We need to stack trees of parents before visiting children. Common tree traversal algorithms include three of a possible six:

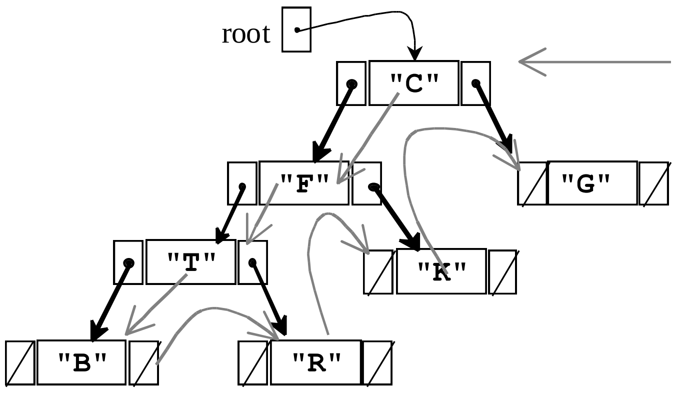
- Preorder traversal: Visit the root, preorder traverse the left tree, preorder traverse the right subtree
- Inorder traversal: Inorder traverse the left subtree, visit the root, inorder traverse the right subtree
- Postorder traversal: Postorder traverse the left subtree, postorder traverse the right subtree, visit the root

When a tree is traversed in a preorder fashion, the parent is processed *before* its children — the left and right subtrees.

Algorithm: Preorder Traversal of a Binary Tree

- Visit the root
- Visit the nodes in the left subtree in preorder
- Visit the nodes in the right subtree preorder

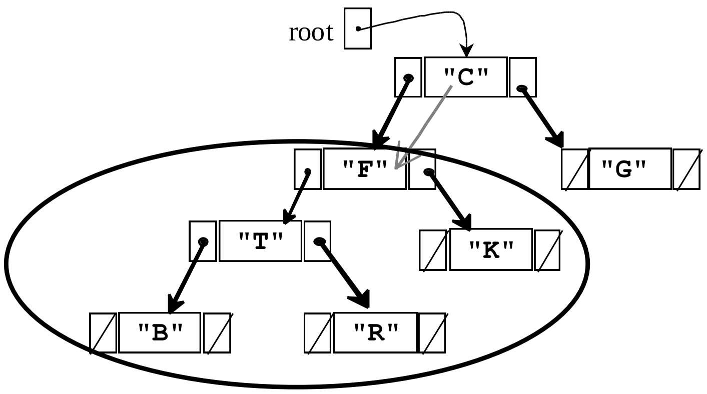
When a binary tree is traversed in a preorder fashion, the root of the tree is "visited" *before* its children — its left and right subtrees. For example, when `preorderPrint` is called with the argument `root`, the element **C** would first be visited. Then a call is made to do a preorder traversal beginning at the left subtree. After the left subtree has been traversed, the algorithm traverses the right subtree of the root node making the element **G** the last one visited during this preorder traversal.



The following method performs a preorder traversal over the tree with "C" in the root node. Writing a solution to this method without recursion would require a stack and a loop. This algorithm is simpler to write with recursion.

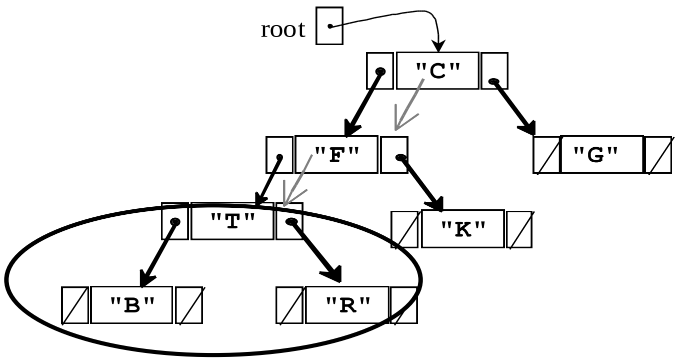
```
public void preOrderPrint() {
  preOrderPrint(root);
}

private void preOrderPrint(TreeNode tree) {
  if (tree != null) {
    // Visit the root
    System.out.print(tree.data + " ");
    // Traverse the left subtree
    preOrderPrint(tree.left);
    // Traverse the right subtree
    preOrderPrint(tree.right);
  }
}
```

When the public method calls `preOrderPrint` passing the reference to the root of the tree, the node with **C** is first visited. Next, a recursive call passes a reference to the left subtree with **F** at the root. Since this TreeNode argument it is not null, **F** is visited next and is printed.



Preorder Traversal so far: **C F**

Next, a recursive call is made with a reference to the left subtree of **F** with **T** at the root, which is visited before the left and right subtrees.

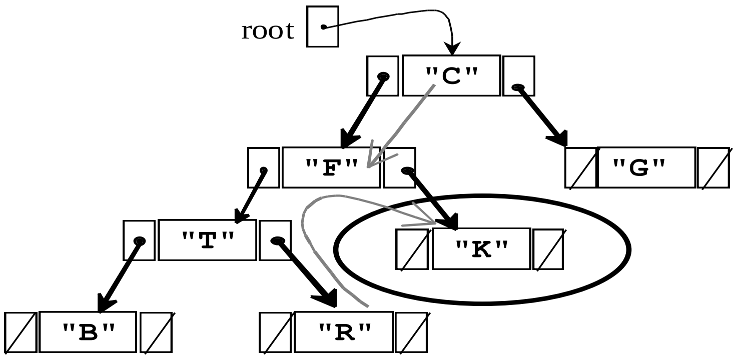


Preorder Traversal so far: C F T

After the root is visited, another recursive call is made with a reference to the left subtree **B** and it is printed. Recursive calls are made with both the left and right subtrees of B. Since they are both null, the if statement is false and the block of three statement is skipped. Control returns to the method with **T** at the root where the right subtree is passed as the argument.

Preorder Traversal so far: C F T B R

The flow of control returns to visiting the right subtree of **F**, which is **K**. The recursive calls are then made for both of K's children (empty trees). Again, in both calls, the block of three statements is skipped since `t.left` and `t.right` are both `null`.



Preorder Traversal so far: C F T B R K

Finally, control returns to visit the right subtree in the first call with the root as the parameter to visit the right subtree in preorder fashion when **G** is printed.

Inorder Traversal

During an inorder traversal, each parent gets processed *between* the processing of its left and right children. The algorithm changes slightly.

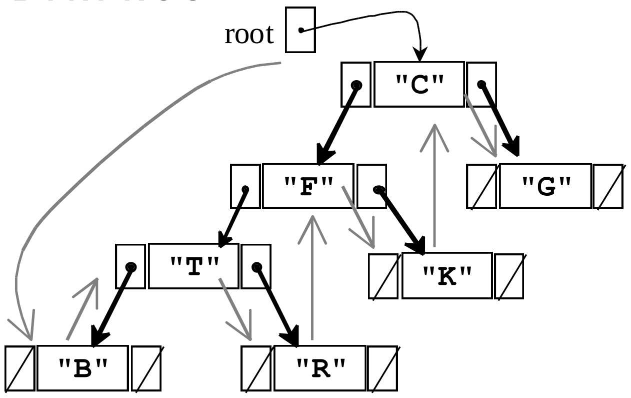
- Traverse the nodes in the left subtree inorder
- Process the root
- Traverse the nodes in the right subtree inorder

Inorder traversal visits the root of each tree only after its left subtree has been traversed inorder. The right subtree is traversed inorder after the root.

```
public void inOrderPrint() {
    inOrderPrint(root);
  }

  private void inOrderPrint(TreeNode t) {
    if (t != null) {
      inOrderPrint(t.left);
      System.out.print(t.data + " ");
      inOrderPrint(t.right);
    }
  }
```

Now a call to `inOrderPrint` would print out the values of the following tree as

B T R F K C G



The `inOrderPrint` method keeps calling `inOrderPrint` recursively with the left subtree. When the left subtree is finally empty, `t.left==null`, the block of three statements executed for **B**.

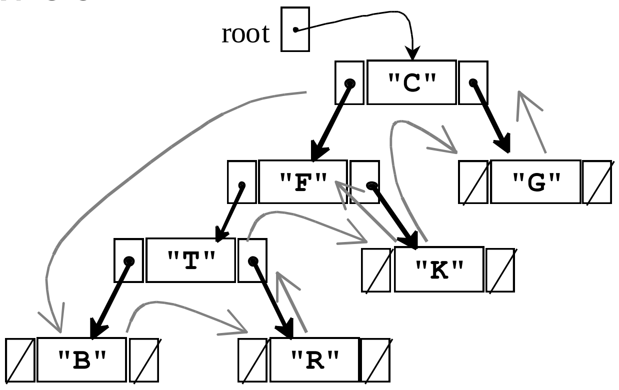
Postorder Traversal

In a postorder traversal, the root node is processed *after* the left and right subtrees. The algorithm shows the process step after the two recursive calls.

1. Traverse the nodes in the left subtree in a postorder manner
2. Traverse the nodes in the right subtree in a postorder manner
3. Process the root

A postorder order traversal would visit the nodes of the same tree in the following fashion:

B R T K F G C



The `toString` method of linear structures, such as lists, is straightforward. Create one big string from the first element to the last. A `toString` method of a tree could be implemented to return the elements concatenated in pre-, in-, or post-order fashion. A more insightful method would be to print the tree to show levels with the root at the leftmost (this only works on trees that are not too big). A tree can be printed sideways with a *reverse* inorder traversal. Visit the right, the root, and then the left.

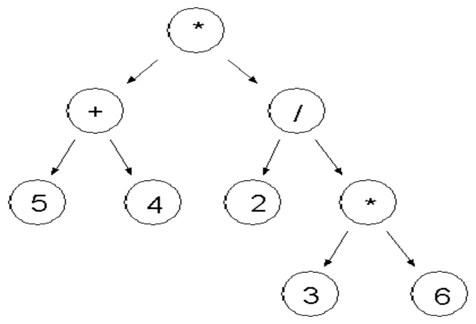
```
    G
C
        K
    F
            R
        T
            B
```

The `printSideways` method below does just this To show the different levels, the additional parameter `depth` begins at 0 to print a specific number of blank spaces depth times before each element is printed. When the root is to be printed depth is 0 and no blanks are printed.

```
  public void printSideways() {
    printSideways(root, 0);
  }

  private void printSideways(TreeNode t, int depth) {
    if (t != null) {
      printSideways(t.right, depth + 1);
      for (int j = 1; j <= depth; j++)
        System.out.print("    ");
      System.out.println(t.data);
      printSideways(t.left, depth + 1);
    }
  }
```

Self-Check

20-3 Write out the values of each node of this tree as the tree is traversed both
a. inOrder b. preorder c. postOrder



20-4 Implement the private helper method `postOrderPrint` that will print all elements separated by a space when this public method is called:

```
public void postOrderPrint() {
  postOrderPrint(root);
}
```

20.4 A few other methods

This section provides a few algorithms on binary trees, a few of which you may find useful.

height

The height of an empty tree is -1, the height of an tree with one node (the root node) is 0, and the height of a tree of size greater than 1 is the longest path found in the left tree from the root. The private `height` method first considers the base case to return -1 if the tree is empty.

```
// Return the longest path in this tree or -1 if this tree is empty.
public int height() {
  return height(root);
}

private int height(TreeNode t) {
  if (t == null)
    return -1;
  else
    return 1 + Math.max(height(t.left), height(t.right));
}
```

When there is one node, `height` returns 1 + the maximum height of the left or right trees. Since both are empty, `Math.max` returns -1 and the final result is (1 + -1) or 0. For larger trees, `height` returns the larger of the height of the left subtree or the height of the right subtree.

leafs

Traversal algorithms allow all nodes in a binary tree to be visited. So the same pattern can be used to search for elements, send messages to all elements, or count the number of nodes. In these situations, the entire binary tree will be traversed.

The following methods return the number of leafs in a tree When a leaf is found, the method returns 1 + all leafs to the left + all leafs to the right. If `t` references an internal node (not a leaf), the recursive calls to the left and right must still be made to search further down the tree for leafs.

```
public int leafs() {
  return leafs(root);
}

private int leafs(TreeNode t) {
  if (t == null)
    return 0;
  else {
    int result = 0;
    if (t.left == null && t.right == null)
      result = 1;
    return result + leafs(t.left) + leafs(t.right);
  }
}
```

findMin

The `findMin` method returns the string that precedes all others alphabetically. It uses a preorder traversal to visit the root nodes first (`findMin` could also use be a postorder or inorder traversal). This example show that it may be easier to understand or implement a binary tree algorithm that has an instance variable initialized in the public method and adjusted in the private helper method.

```
// This instance variable is initialized it in the public method findMin.
private String min;

// Return a reference the String that alphabetically precedes all others
public String findMin() {
  if (root == null)
    return null;
  else {
    min = root.data;
    findMinHelper(root);
    return min;
  }
}

public void findMinHelper(TreeNode t) {
  // Only compare elements in nonempty nodes
  if (t != null) {
   // Use a preorder traversal to compare all elements in the tree.
   if (t.data.compareTo(min) < 0)
      min = t.data;
    findMinHelper(t.left);
    findMinHelper(t.right);
  }
}
```

Self-Check

20-5 To `BinaryTreeOfStrings`, add method `findMax` that returns the string that follows all others alphabetically.

20-6 To `BinaryTreeOfStrings`, add method `size` that returns the number of nodes in the tree.

Answers to Self-Checks

20-1 a) theRootValue c) baker, Charlie, echo, foxtrot e) echo, foxtrot
b) 7 d) theRootValue, able, delta f) 0 (the bottom level is level 2)

20-2 *Assume this code is in a method of class BinaryTreeOfStrings*

```
root = new tree node("theRootValue");
root.left = new TreeNode("able");
root.left.left = new TreeNode("baker");
root.left.right = new TreeNode("charlie");
root.right = new TreeNode("delta");
root.right.left = new TreeNode("echo");
root.right.right = new TreeNode("foxtrot");
```

20-3 a. inorder: 5 + 4 * 2 / 3 * 6 b. preorder: * + 5 4 / 2 * 3 6 c. postOrder: 5 4 + 2 3 6 * / *

20-4 *Output using the tree above would be:* `baker Charlie able echo foxtrot delta theRootValue`

```
private void postOrderPrint(TreeNode t) {
  if (t != null) {
    postOrderPrint(t.left);
    postOrderPrint(t.right);
    System.out.print(t.data + " ");
  }
}
```

20-5 *Alternatives exist. This solution shows that declaring an extra instance variable can make it easier to understand*

```
private int max;

public int findMax(TreeNode<Integer> t) {
  if (t == null)
    return 0;
  else {
    max = t.data;
    findMaxHelper(t);
    return max;
  }
}

public void findMaxHelper(TreeNode<Integer> t) {
  if (t != null) {
    int temp = ((Integer) t.data).intValue();
    if (temp > max)
      min = temp;
    findMaxHelper(t.left);
    findMaxHelper(t.right);
  }
}
```

20-6

```
public int size() {
  return size(root);
}

private int size(TreeNode t) {
  if (t == null)
    return 0;
  else
    return 1 + size(t.left) + size(t.right);
}
```