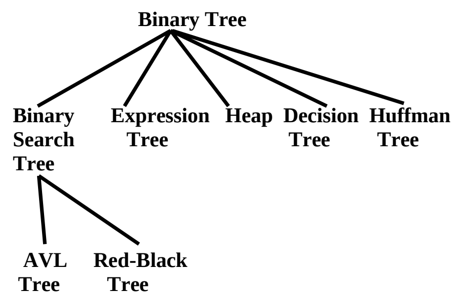
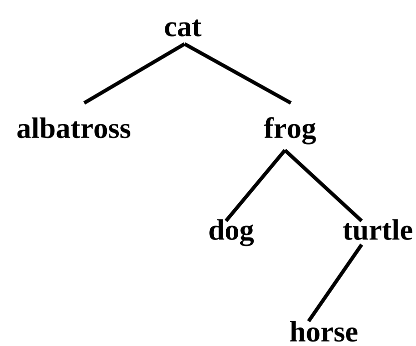
Chapter 9

Binary Trees

Here are some binary trees we will be studying in the next few chapters



A *binary tree* t is either empty or consists of an element, called the *root element,* and two distinct binary trees, called the *left subtree* and *right subtree* of t.



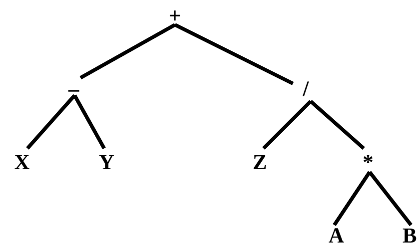
Those two subtrees are written as leftTree(t) and rightTree(t). Functional notation is used instead of object notation – such as t.leftTree( ) – because there will be no BinaryTree class.

Why is there no BinaryTree **class? It would not be flexible enough for the binary-tree-based classes already in the Java collections framework (**TreeMap **and** TreeSet).

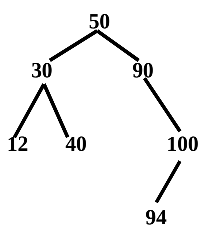
Using botanical terminology (besides root and tree):

A *leaf* is an element whose left and right subtrees are empty.

A *branch* is a line drawn from an element to its left or right subtree.



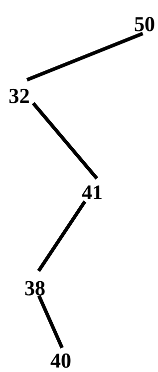
This is an *expression tree*: Each leaf is an operand, and each non-leaf is a binary operator.



This is a *binary search tree*: Each element in the left subtree is less than the root element, each element in the right subtree is greater than the root element, and the left and right subtrees are themselves binary search trees.

Can you create a binary tree in which each element in the left subtree is less than the root element and each element in the right subtree is greater than the root element, but the binary tree is *not* a binary search tree?

Another binary search tree:



Suppose a binary tree *t* is a chain – that is, each element except the only leaf has exactly one branch. If *t* has *n* elements, how many branches are there from the root, the only leaf?

How can we define *leaves(t)*, the number of leaves in a binary tree *t*?

Recursively!

Simplest case: When t is empty

Other simple case: When t has only 1 element

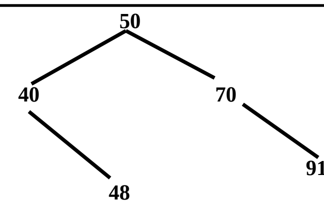
Otherwise, express the number of leaves in t in terms of the number of leaves in leftTree(t) and rightTree(t).

```
if t is empty
      leaves(t) = 0
else if t consists of a root element only
     leaves(t) = 1
else
    leaves(t) = leaves(leftTree(t)) +
                  leaves(rightTree(t))
```

How about *n(t)*, the number of elements in t?

```
if t is empty
    n(t) =
else
    n(t) =
```

Now for some familial terminology:



40 is the left child of 50

50 is the parent of 40

50 is the parent of 70

40 and 70 are siblings

What is 91 to 50?

What is 91 to 48?

Descendant?

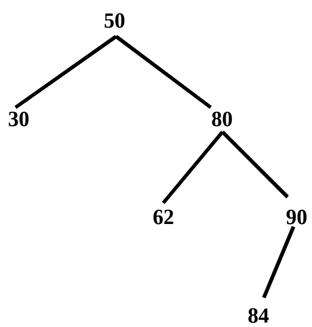
d is a *descendant* of a if

a is the parent of d

Or if

The parent of d is …?

A *path* in a binary tree is a sequence of elements in which each element except the last is the parent of the next element in the sequence.

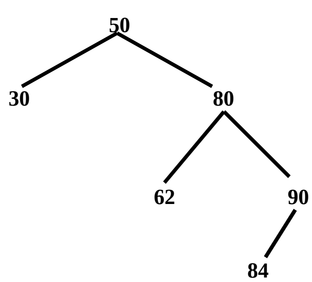


Determine the path from 50 to 84.

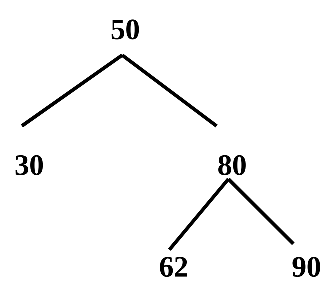
Answer:

50, 80, 90, 84

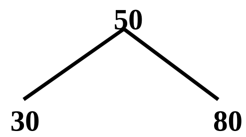
In a binary tree t, *height(t)* is the number of branches from the root to the farthest leaf.



Determine height(t)



height(t) = ?



height(t) = ?

50

height(t) = ?

One more example: What is the height of the tree if the height of the left subtree is 4 and the height of the right subtree is 10?

So if a binary tree has height 0, its left and right subtrees must each have a height of?

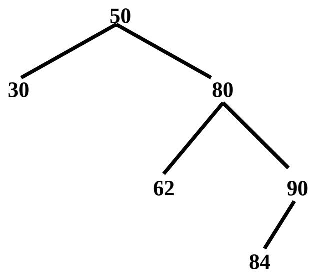
If t is empty
height (t) = – 1
Else
height (t) =

***depth(x)*, the depth of an element x is the number of branches from the root element to x.**

If x is the root element
depth(x) = 0
Else
depth(x) =

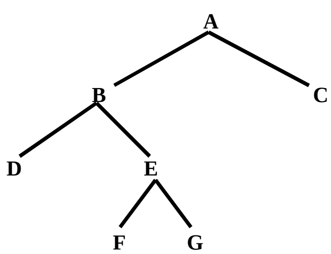
***level(x)*, the level of x, is the same as the depth of x.**

In the following binary tree, what is depth(62)? level(90)? The height of the subtree rooted at 90?

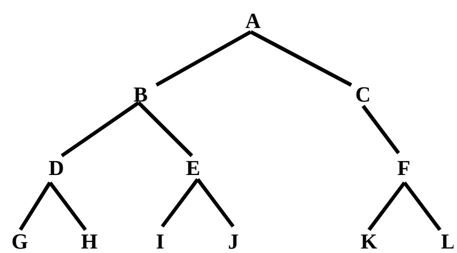


A binary tree t is a *two-tree* if t is empty or if each non-leaf in t has two branches.

An example of a two-tree:



Is this a two-tree?

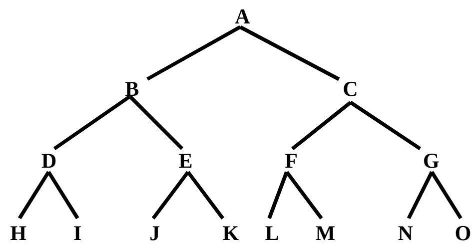


Recursively speaking:

A binary tree t is a *two-tree* if t is empty or if leftTree(t) and rightTree(t) are either both empty or both non-empty two-trees.

A binary tree t is *full* if t is a two-tree and all of the leaves of t have the same depth.

A full binary tree:

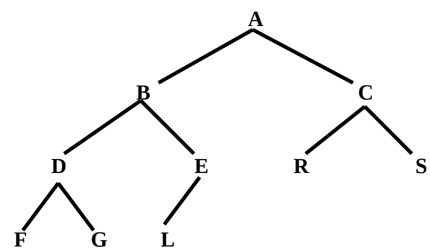


Recursively speaking:

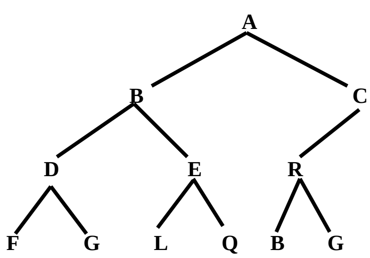
A binary tree t is *full* if t is empty or if height(leftTree(t)) = height(rightTree(t)) and both leftTree(t) and rightTree(t) are …?

A binary tree t is *complete* if t is full through a depth of height(t) - 1, and each leaf whose depth is height(t) is as far to the left as possible.

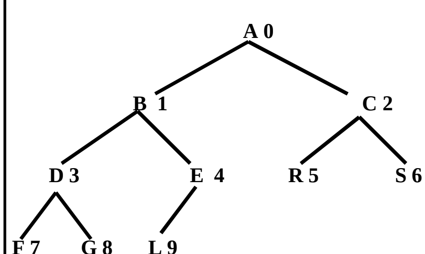
A complete binary tree:



A binary tree that is not complete:



With each element in a complete binary tree, we associate a non-negative integer as follows:



This association suggests that a complete binary tree can be implemented with an array:

A	B	C	D	E	R	S	F	G	L

Then the random-access property of arrays allows quick access of parent from child and children from parent.

Parent at 0, children at 1, 2
Parent at 1, children at 3, 4
Parent at 2, children at 5, 6
…
Parent at i, children at ?

Child at 1, parent at 0
Child at 2, parent at 0
Child at 3, parent at 1
Child at 4, parent at 1
Child at 5, parent at 2
Child at 6, parent at 2
…
Child at i, parent at ?

So it is efficient to implement a complete binary tree with an array.

Can a complete binary be stored in an ArrayList**? Yes, same idea as for an array.**

How about a LinkedList**? not good.**

Exercise: Construct a binary tree t such that

1. **t is a two-tree (each element in t has either 2 children or no children);**
2. **t is complete;**
3. **height (leftTree (t)) = height (rightTree (t));**
4. **t is not full.**

The binary-tree theorem:

For any non-empty binary tree t:

1. $\text{leaves(t)} <= \dfrac{n(t) + 1}{2.0}$

2. $\dfrac{n(t) + 1}{2.0} <= 2^{\text{height(t)}}$

3. Equality holds in part 1 if and only if t is a two-tree.

4. Equality holds in part 2 if and only if t is full.

What is the significance of the binary tree theorem? Suppose t is full. Then

$$\frac{n(t) + 1}{2.0} = 2^{\text{height(t)}}$$

so

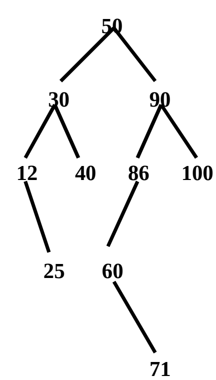
$$\text{height(t)} = \log_2\left(\frac{n(t) + 1}{2.0}\right)$$

That is, a full tree has height that is logarithmic in *n*.

The height of a complete binary tree is also logarithmic in *n*.

What about the height of a chain?

In Chapter 10, we will look at a special kind of binary tree: The binary search tree.



In a binary search tree, each element in the left subtree is less than the root element, each element in the right subtree is greater than the root element, and the left and right subtrees are …?

In the BinarySearchTree **class, the "average" height of a** BinarySearchTree **object is logarithmic in *n*, and so the average time to insert, remove or search is logarithmic in *n*.**

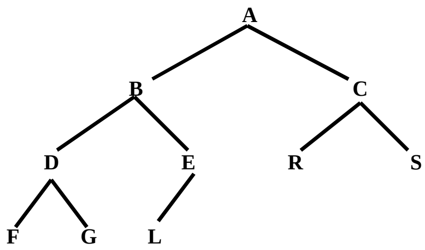
But a BinarySearchTree **object can be a chain; then the time to insert, remove or search is linear in *n*.**

For an (unsorted) array, ArrayList**, or** LinkedList **collection, the time to insert, remove or search is linear in *n*.**

External Path Length

Let t be a non-empty binary tree. *E(t)*, the external path length of t, is the sum of the depths of all leaves in t.

For example, find the external path length of the following binary tree:



E(t) = 3 + 3 + 3 + 2 + 2 = 13

The external path length theorem:

Let t be a binary tree with k > 0 leaves. Then

E(t) >= (k / 2) floor ($\log_2 k$).

This result is used in Chapter 11 when we establish a lower bound for sorting algorithms.

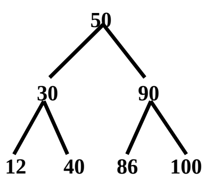
Traversals of a Binary Tree

A *traversal* of a binary tree is an algorithm that accesses each item in the binary tree.

The following traversal algorithms are not methods because we have not defined, and will not, define a binary-tree class.

```
1. inOrder(t)
{
   if (t is not empty)
   {
      inOrder(leftTree(t));
      access the root element of t;
      inOrder(rightTree(t));
   } // if
} // inOrder traversal
```

Left – Root – Right



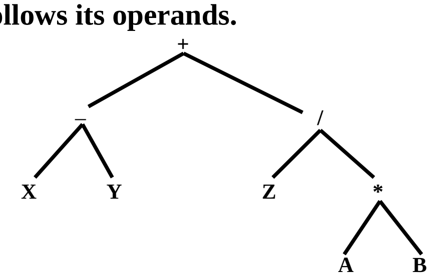
Determine the order in which the elements would be accessed during an in-order traversal.

Answer: 12, 30, 40, 50, 86, 90, 100

```
2.  postOrder (t)
    {
      if (t is not empty)
      {
         postOrder(leftTree(t));
         postOrder(rightTree(t));
         access the root element of t;
      } // if
    } // postOrder traversal
```

Left – Right – Root

Determine the order in which the elements would be accessed during a post-order traversal. Hint: An operator immediately follows its operands.



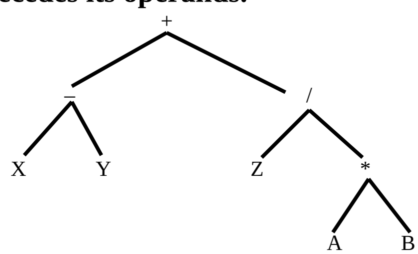
Answer: X, Y, -, Z, A, B, *, / +

Postfix!

3. **preOrder (t)**
```
{
    if (t is not empty)
    {
        access the root element of t;
        preOrder (leftTree (t));
        preOrder (rightTree (t));
    } // if
} // preOrder traversal
```

Root – Left – Right

Determine the order in which the elements would be accessed during a pre-order traversal. Hint: An operator immediately precedes its operands.

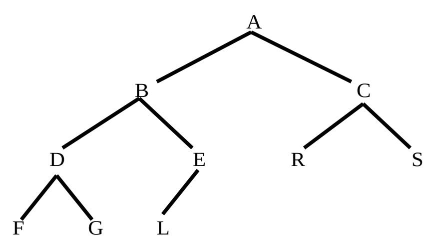


Answer: +, -, X, Y, /, Z, *, A, B

Prefix!

4. **To perform a breadth-first traversal of a non-empty binary tree, first access the root element, then the children of the root element, from left to right, then the grandchildren of the root element, from left to right, and so on.**

Perform a breadth-first traversal of the following binary tree.



Answer: A, B, C, D, E, R, S, F, G, L

Perform the other traversals of that tree:

inOrder (Left-Root-Right)
postOrder (Left - Right - Root)
preOrder (Root - Left - Right)