Chapter 9 Binary Trees

- In this chapter, we will drift away from the linear structure to introduce what is essentially a non-linear construct: The binary tree.
- This chapter focuses on the definition and properties of binary trees and what will provide the necessary background for the next chapters.

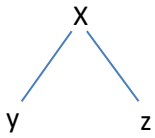
9.1 Definition of Binary Tree

- The **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.
- Functional notations leftTree(t) and rightTree(t) are used instead of the object notation such as t.leftTree(), because there is no binary-tree data structure.
- Different types of binary threes have widely different methods – even different parameters lists – for such operations as inserting and removing.
- Definitions of a binary tree is recursive and many definitions associated with binary tree are naturally recursive.

  Examples of the binary trees page 378.

9.2 Properties of Binary trees

- The line from the root element to a subtree is called a **branch**.
- An element whose associated left and right subtrees are both empty is called a **leaf**. A leaf has no branches going down from it.
- Some binary trees concepts use familiar terminology.



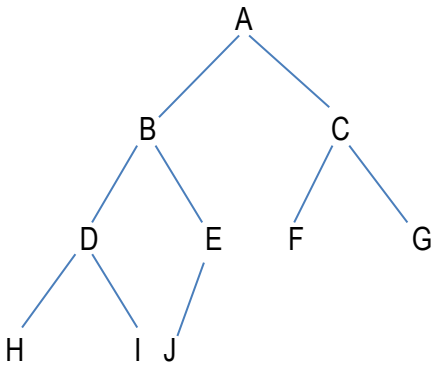
- We say that x is the parent of y and that y is a left child of x.
- Also, x is the parent of z and z is a right child of x.
- Let t be a binary tree. The number of leaves in t, written ***leaves(t)***, can be defined as follows:

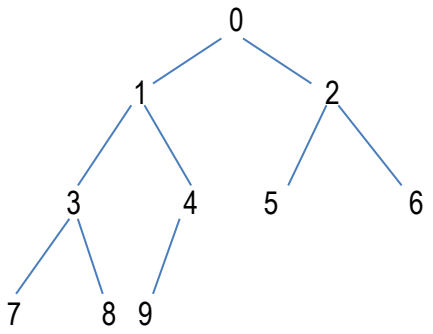
```
if t is empty
    leaves(t) = 0
else if t consists of a root element only
   leaves(t) = 1
else
  leaves(t) = leaves(leftTree(t)) + leaves(rightTree(t))
```

This is mathematical definition, not the Java method.

- In binary tree, each element can have a zero, one, or two children.
- In binary tree, a root element does not have a parent, and every other element has exactly one parent.
- For example, an element A is an ancestor of element B if B is in the subtree whose root element is A.
- If A is an ancestor of B, the **path** from A to B, the path from A to B is the sequence of elements, starting with A and ending with B, in which each element is the sequence (except the last) is the parent of the next element.
- For example, Figure 9.1e path from 37 to 32 will have a sequence: 37, 25, 30, and 32.
- The height of the binary tree is the number of branches between the root and the farthest leaf, that is, the leaf with the most ancestors. Example in text 9.3 page 380 has a height 3.
- Height of the empty subtree has the height of -1.
- For each element in the binary tree, we can define a similar concept as height: the level of the element.
- The **height** of the binary tree is the number of branches between the root and the farthest leaf.
- The **level** of element x is the number of branches between the root element and element x. Figure 9.4 page 380.
- A **two-tree** is a binary tree that either is empty or in which each non-leaf has 2 branches going down from it. Example, figure 9.5a has a two-tree and Figure 9.5b is not a two-tree.
- A binary tree t is **full** if t is a two-tree with all of its leaves on the same level. Every full binary tree is a two-tree but the converse is not necessarily true.
- A binary tree t is **complete** if t is full through the next – to – lowest level and all of the leaves at the lowest level are as far to the left as possible. The "lowest level", mean the level farthest from the root.
  Example 9.8 page 382.

- In a complete binary tree, we can associate a position with each element. The root element is assigned at position 0.
- For non-negative i, if the element at position i has children, the position of its left child is 2i+1 and the position of its right child is 2i+2.





A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

9.3 Binary Tree Theorem

- The following theorem characterizes the relationships among the leaves(t), height(t), and n(t).
- leaves(t) - the number of leaves in the binary tree t
- height(t) - the height of the binary tree t
- n(t) – number of elements in the binary tree

Binary tree Theorem: For any non-empty binary tree t,

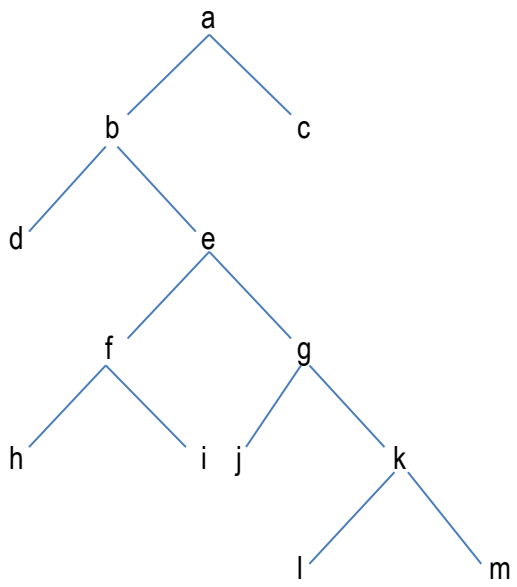
1. leaves(t) $\leq \frac{n(t)+1}{2.0}$
2. $\frac{n(t)+1}{2.0} \geq 2^{height(t)}$
3. Equality holds in part1 if and only if t is a two – tree
4. Equality holds in part 2 if and only if t is full

9.4 External Path Length

- Let t be a non-empty tree. E(t), the **external path length** of t, is the sum of the depths of all the leaves in t.
- External Path Length Theorem: Let t be a binary tree with k>0 leaves. Then,

E(t) ≥ (k/2)floor ($\log_2$ k)

- Example.



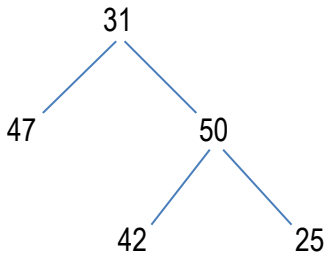
9.5 Traversal of a Binary Tree

- A traversal of a binary tree t is an algorithm that processes each element in t exactly once.
- In this section, our attention will be on the algorithms not the methods. That is, there are no classes or interfaces declared.
- Traversals studied in this section will be related to iterators
- Four kinds of traversal: inOrder, postOrder, preOrder and BreadthFirst

Traversal 1. inOrder Traversal: **Left – Root – Right**

- Basic idea of this traversal algorithm is that we perform an inOrder traversal of the left subtree, then we process the root element, and finally, we perform an inOrder traversal of the right subtree.

```
inOrder(t)
{
        if(t is not empty)
        {
                inOrder(leftTree(t));
                preocces the root element of t;
                inOrder(rightTree(t));
        }

}
```

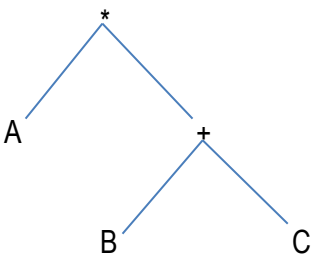


inOrder traversal: 47, 31, 42, 50, and 25.

Traversal 2. postOrder Traversal: **Left – Right – Root**

- Basic idea behind this recursive algorithm is that we perform postOrder traversals of the left and right subtrees before processing the root element.

```
inOrder(t)
{
        if(t is not empty)
        {
                postOrder(leftTree(t));
                postOrder(rightTree(t));
                preocces the root element of t;

        }

}
```

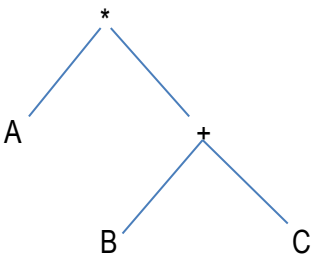


postOrder traversal: ABC + *

Traversal 3. PreOrder Traversal: **Root – Left – Right**

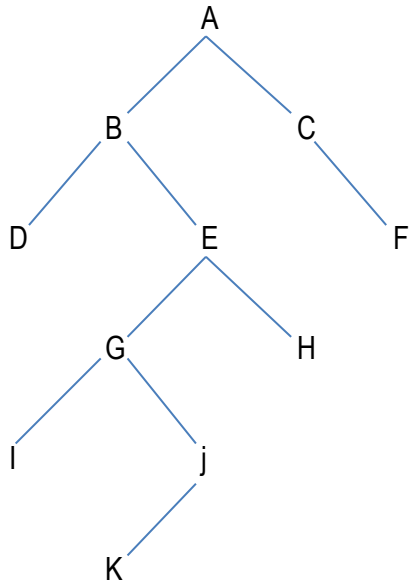
- Idea behind this recursive algorithm is to process the root element and then perform left and right subtrees.

```
inOrder(t)
{
        if(t is not empty)
        {
                preocces the root element of t;
                preOrder(leftTree(t));
                preOrder(rightTree(t));

        }

}
```



PreOrder traversal: *A+BC

- A search of a binary tree that employs a preorder traversal is called a **depth-first-search** because the search goes to the left as deeply as possible before searching the right.



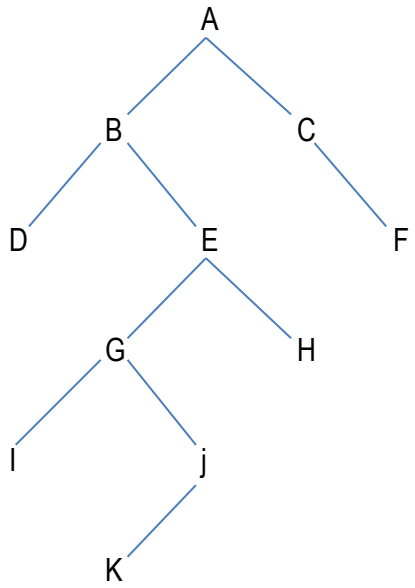
Depth-first-search of letter H: A, B, D, E, G, I, J, K, H.

- The search stops when (if) element sought is found, so the traversal may not be completed.

Traversal 4. BreadthFirst Traversal: **Level - by – Level**

- To perform a breath-first traversal of a non-empty binary tree t, first process the root element, then the children of the root, from left to right, then the grandchildren of the root, from left to right and so on.

```
breadthFirst (t)
{
        if (t is not empty)
        {
                queue.enqueu(t);
                while(queue is not empty)
                {
                        tree = queue.dequeue();
                        process tree's root;
                        if (leftTree(tree) is not empty)
                                queue.enqueue(leftTree(tree));
                        if (rightTreee (tree) is not empty)
                                queue.enqueue(rightTree(tree));
                }


        }

}
```



BreadthFirst traversal: A, B, C, D, E, F, G, H, I , J, K.

- During each while loop iterations, one element is processed, so worstTime(n) is linear in n.
- Queue is used for a breadth – first traversal because we want the subtrees retrieved in the same order they were saved (First-in, First-Out).
- With inOrder, postOrder, and preOrder traversals, the subtrees are retrieved in the reverse of the order they were saved (Last-in, First-out).