COMP2421 — DATA STRUCTURES & ALGORITHMS

Trees

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Trees

- Trees are non-linear data structures
- They allow implementing algorithms faster than linear data structures such as lists, sequences, queues, …

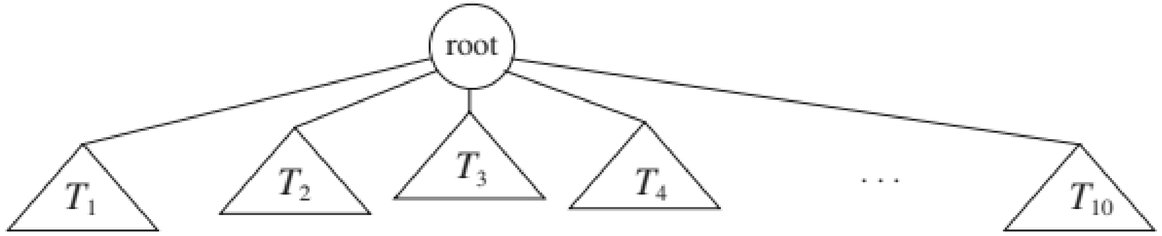


Figure 4.1 Generic tree

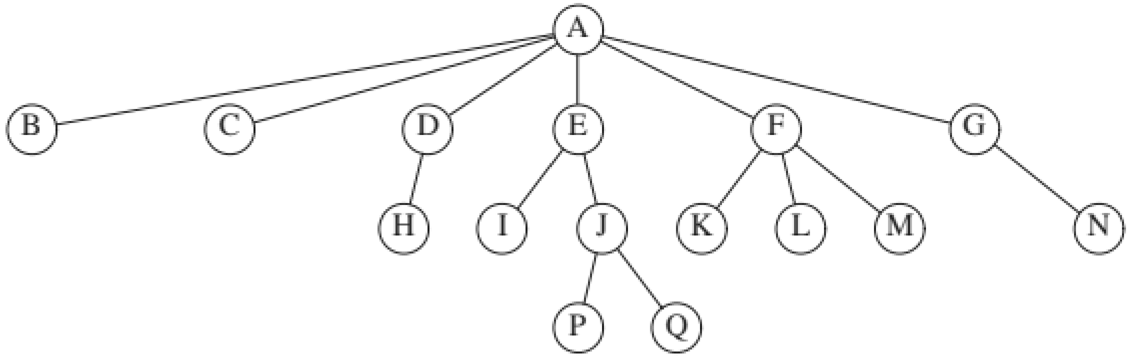


Figure 4.2 A tree

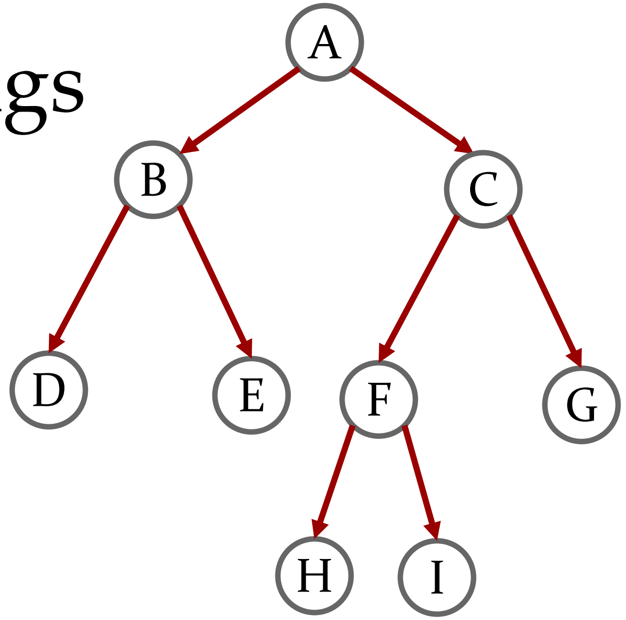
Trees (2)

- Trees provide a natural organisation of data. They are used in file systems, graphical user interfaces, data bases, and websites
- Trees provide efficient insertion and searching
- The relationships in trees are hierarchical (parent, child, ancestors, descendants,…)
- A tree consists of a distinguished node called root, and zero or subtree T_1, T_2, …, T_n. The root of each subtree has a direct connection to the root of the tree *r*.

Trees (3)

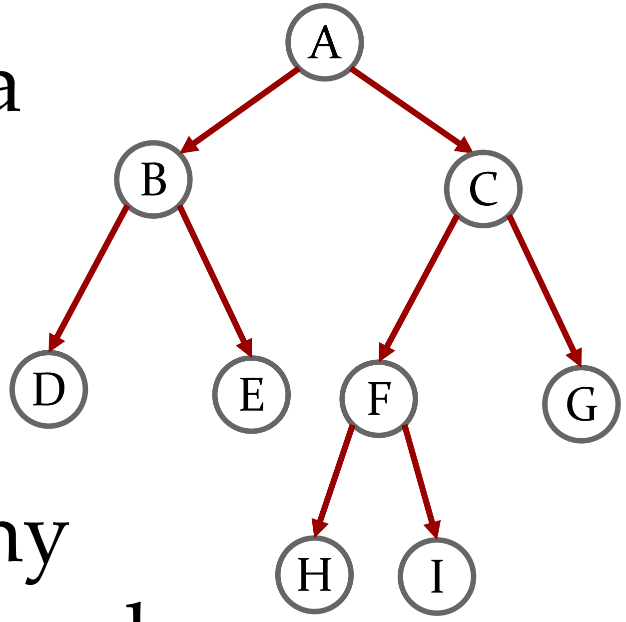
- Formal definition: Tree T to be a set of nodes storing elements in parent-child relationship with the following properties:
  - If T is non-empty, it has a special node called root of T, which has no parent
  - Each node v in T different from the root has a unique parent node w; every node with parent w is a child of w

Trees (4)



- Two nodes of the same parent are called siblings
- A node V is called internal if it has one more children
- A node V is called external (leaf node) if it has no children
- An edge of tree is pair of nodes (u, v) such that u is the parent of v, or vice versa
- Number of edges in a tree = n – 1 (n is the # of nodes)

Trees (5)

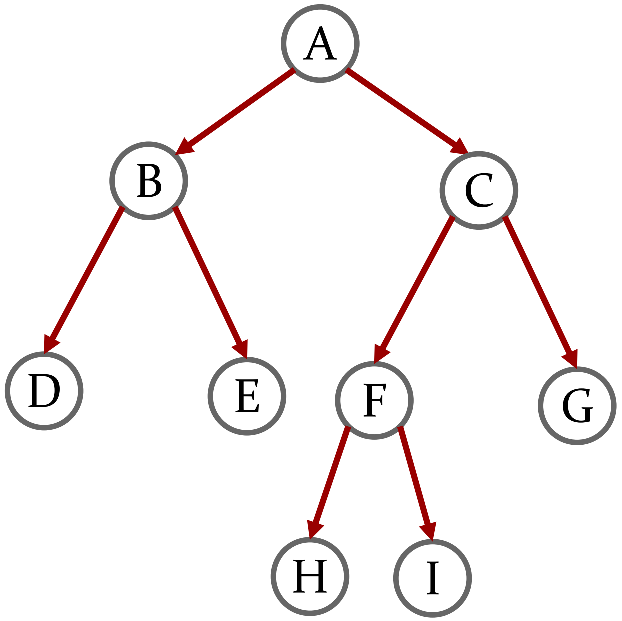


- **Depth** of a node is the number of edges from a node v to the root node
- **Height** of a tree is the maximum number of edges till a leaf node
- A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge
- E.g., A →C→F→H

TREE TRAVERSAL ALGORITHMS

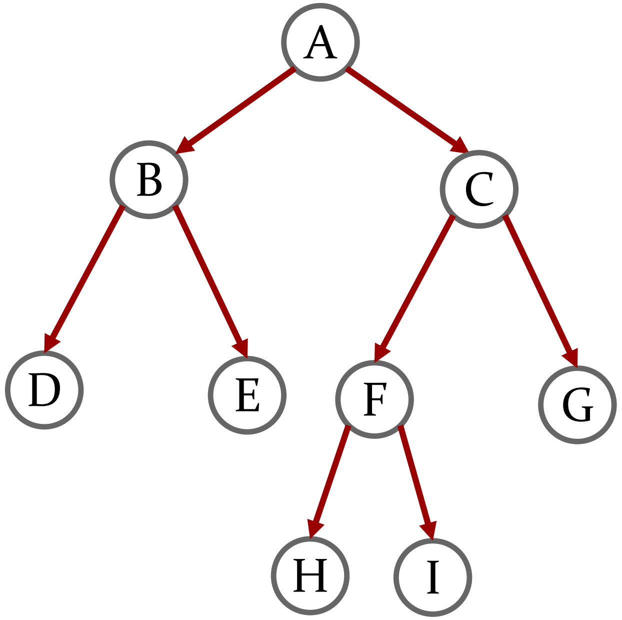
Tree Traversal Algorithms

- A tree traversal of a tree T is a systematic way of accessing or visiting all the nodes of T.
- There are three methods to read a tree:
  - Inorder (left → root → right)
  - Preorder (root → left → right)
  - Postorder (left → right → root)



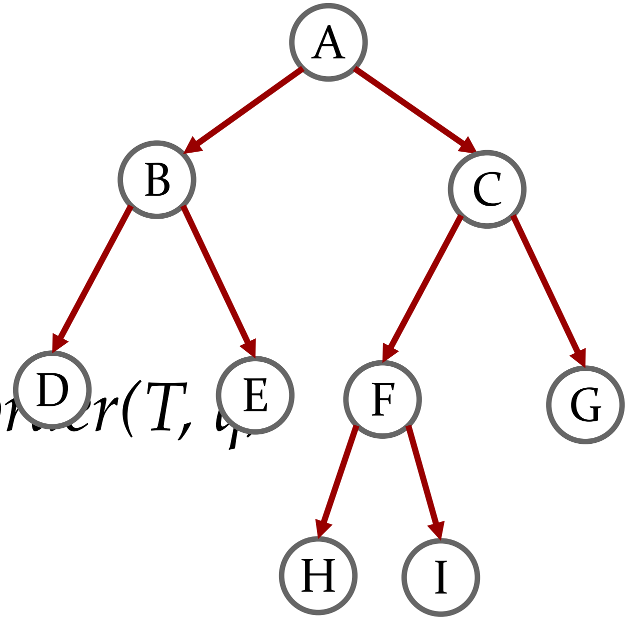
Inorder Traversal

- *Algorithm Inorder(T, p):*
  - *Recursively traverse the left subtree rooted at q by calling inorder(T, q)*
  - *Perform the "visit" action of node p*
  - *Recursively traverse the right subtree rooted at v by calling inorder(T, v)*



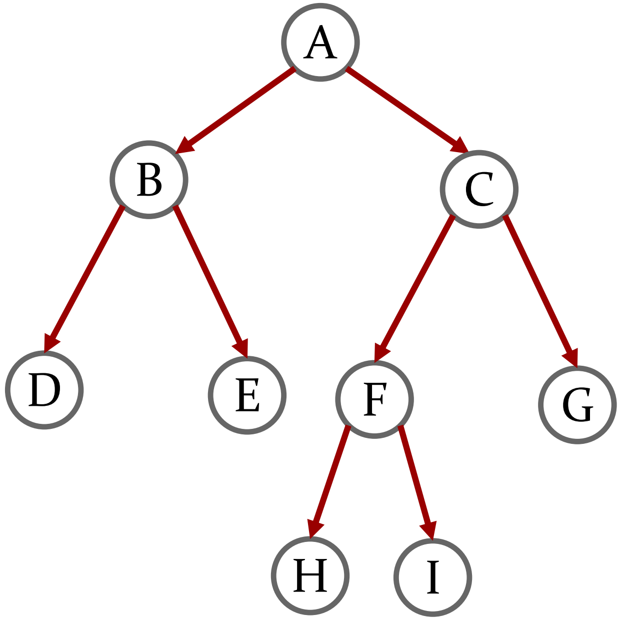
Preorder Traversal

- *Algorithm preorder (T, p):*
  - *Perform the "visit" action of node p*
  - *For each child q of p do:*
    - *Recursively traverse the subtree rooted at q by calling preorder(T, q)*



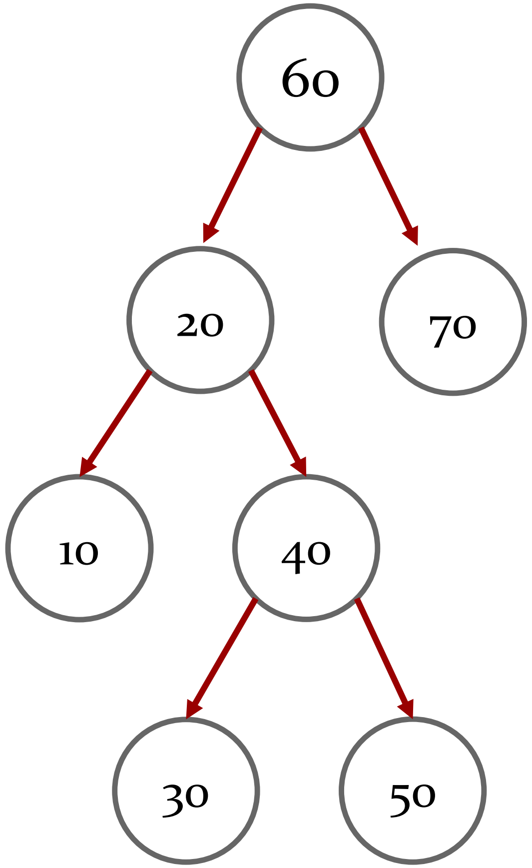
Postorder Traversal

- *Algorithm postorder (T, p):*
  - *For each child q of p do:*
    - *Recursively traverse the subtree rooted at q by calling postorder(T, q)*
  - *Perform the "visit" action of node p*



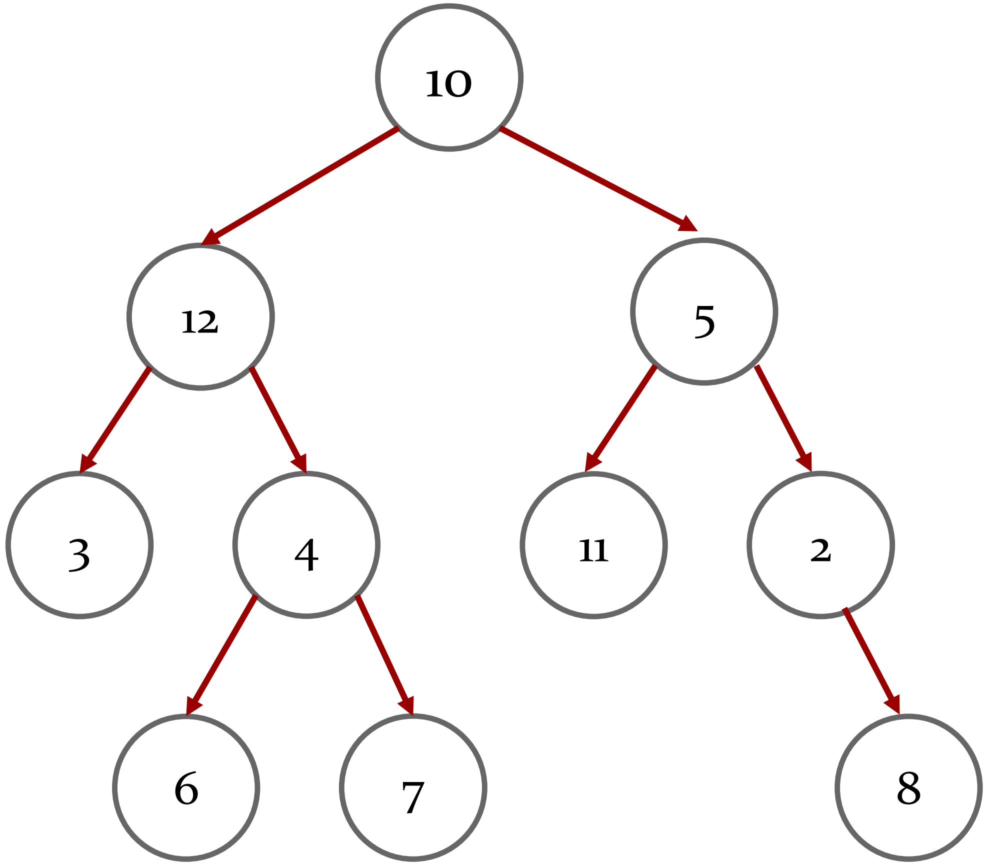
Tree Traversal – Example

- Print the content of the following tree in inorder, preorder, and postorder



Tree Traversal – Example (2)

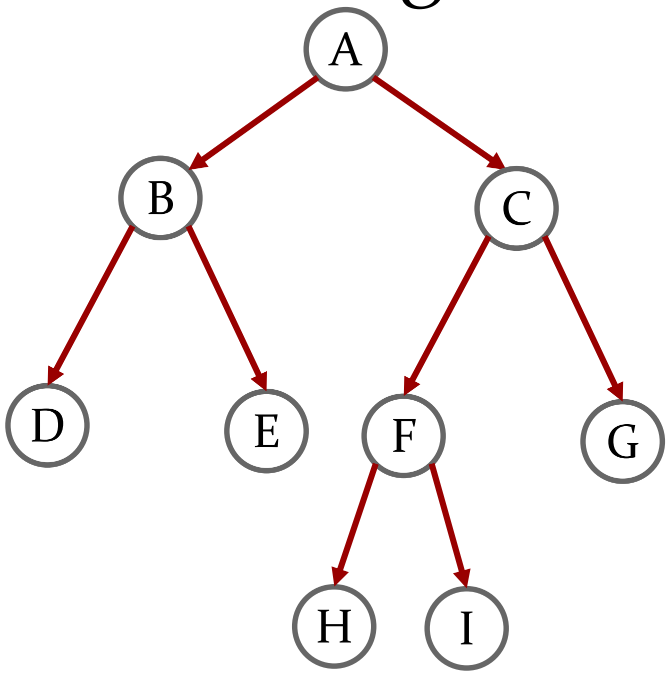
- Print the content of the following tree in inorder, preorder, and postorder



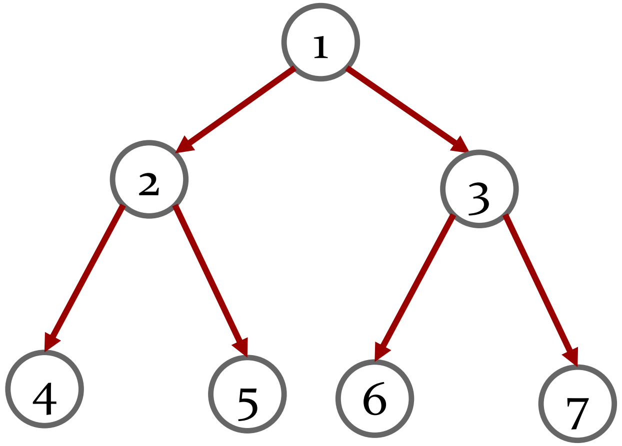
BINARY TREES

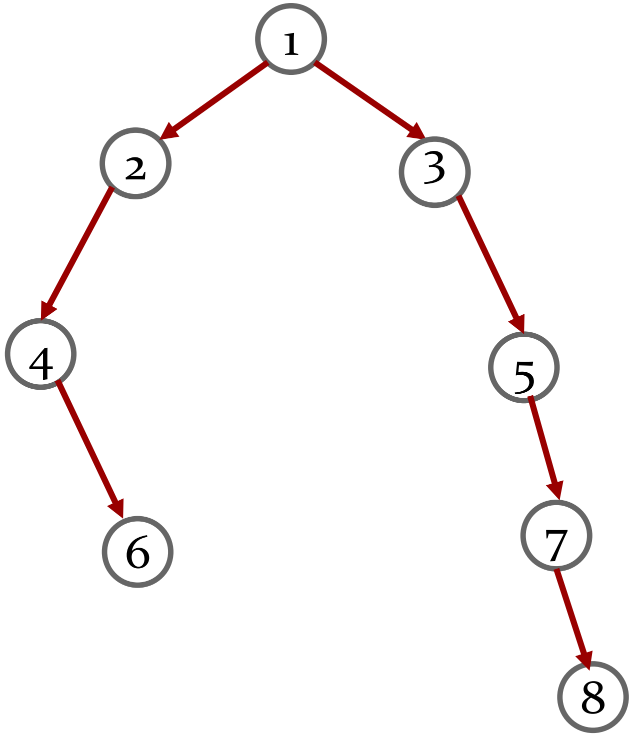
Binary Trees

- A tree in which no node has more than 2 children.
- Each child node is labeled as being either left child or right child.
- A left child precedes the right child in the ordering of children of a node.
- The depth of an average binary tree is smaller than n → O(log n) the average value of the depth for binary search tree.



Binary Trees (2)

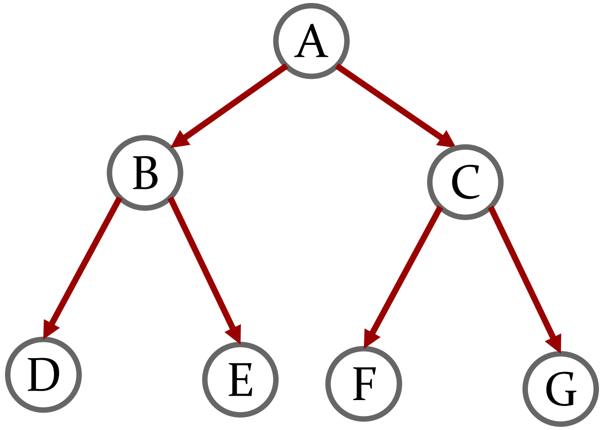




- A proper binary tree (full): if each node has either zero or two children.
- An improper tree: not a proper binary tree.

Binary Trees (3)

- A complete binary tree is very special tree that provides the best ratio between the number of nodes and the height.
- The height *h* of a complete binary tree with *N* nodes is at most O(log N). We can prove it by counting nodes on each level, starting with the root, assuming that each level has the maximum number of nodes:
  $n = 1 + 2 + 4 + \ldots + 2^{h-1} + 2^{h} = 2^{h+1} - 1$
- Solving this with respect to h, we obtain
  $h = O(\log n)$



EXPRESSION TREES

Expression Tree

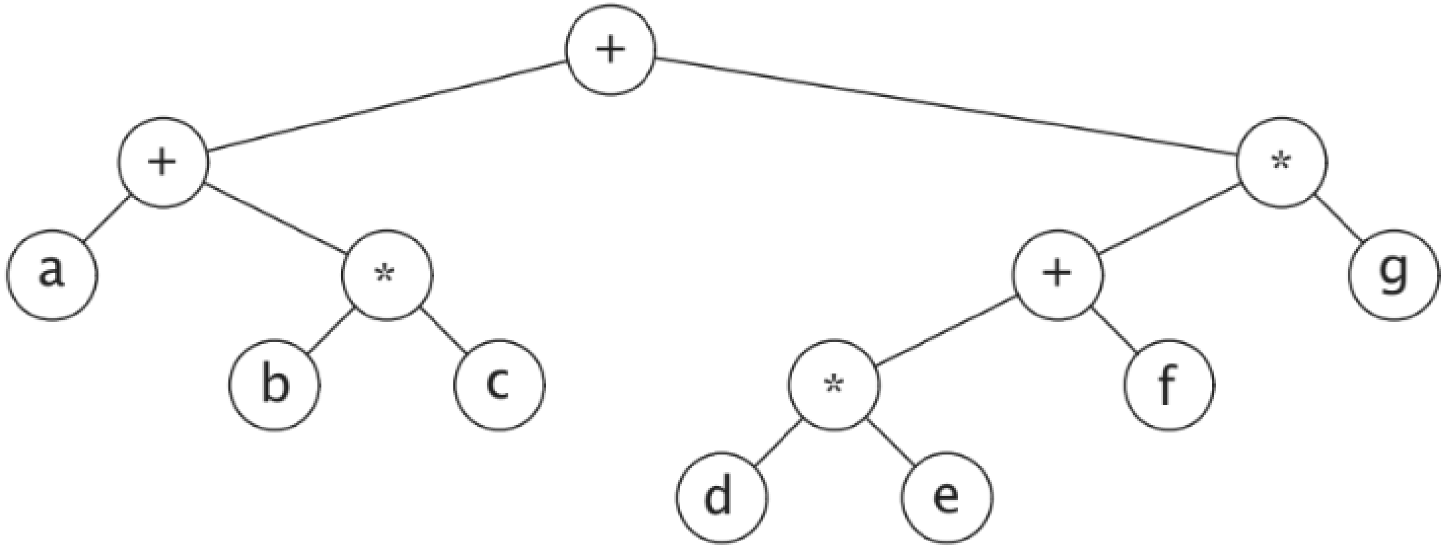
- An arithmetic expression can be represented by a tree whose external nodes are operands (variables or constants) and the internal nodes are operations.
- *If a node is external, then its value is a variable or constant.*
- *If a node is internal, then its value is defined by applying its operation to the values of its children.*

Expression Tree (2)

- An expression tree is a proper binary tree since each of the operators take exactly two operands.
- If the unary – (negation) is added, this would create an imporper binary tree.
- An expression tree can be evaluated by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

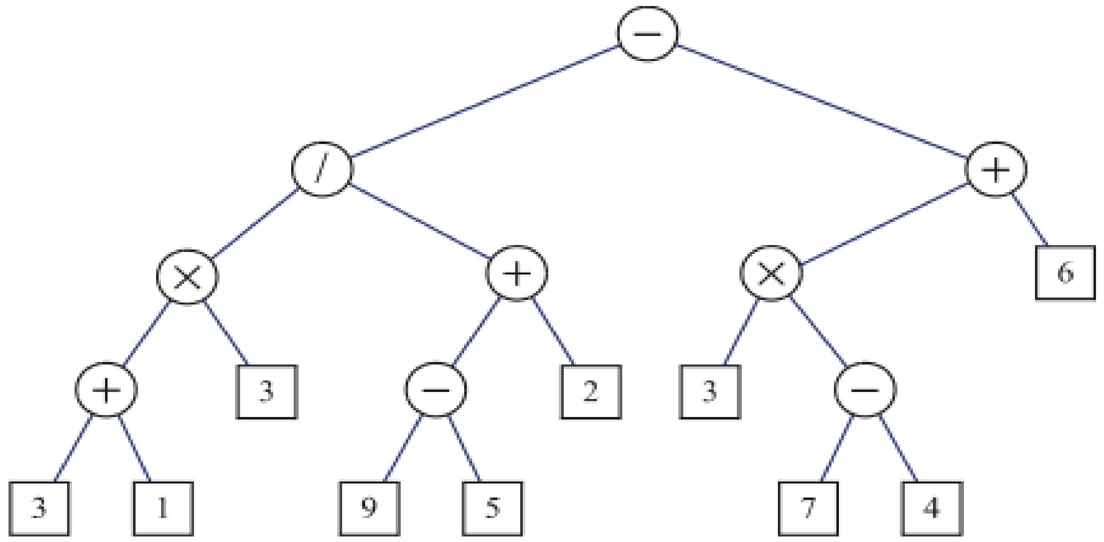
Expression Tree – Example

- Given the following expression tree, what is the inorder, preorder, and postorder expressions.



Expression Tree – Example (2)

- Given the following expression tree, what is the inorder, preorder, and postorder expressions.



Constructing Expression Tree

- The conversion is from **postfix** into an **Expression Tree**.
- If the expression is infix, then we convert the infix into postfix, and then postfix into expression tree.
- Steps:
  - Read one symbol at a time;
  - If the symbol is an operand, then create a one-node tree and push a pointer to it onto a stack;
  - If the symbol is an operator, we pop pointers to two trees T1 & T2 from the Stack (T1 is popped first) and form a new tree whose root is the operator and whose left & right children point to T2 & T1 respectively.

Constructing Expression Tree - Example

- a b + c d e + * *

- The first 2 symbols are operands, create one node trees and push them into the stack

Constructing Expression Tree – Example (2)

- a b + c d e + * *

- Next, operation + is read, so two trees are popped, a new tree is formed, and it is pushed onto the stack.

Constructing Expression Tree – Example (3)

- `a b + c d e + * *`

- Next, c, d, and e are read, and for each a one-node tree is created and the corresponding tree is pushed onto the stack.

Constructing Expression Tree – Example (4)

- a b + c d e + * *

- Next, operations are read and the elements are popped from the stack and pushed, apply the operations, and push back again.

Constructing Expression Tree – Example (5)

- a b + c d e + * *

- Finally, the last symbol is read, two trees are merged, and the final tree is left on the stack.

Constructing Expression Tree – Example (6)

- a b + c d e + * *