Trees

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

- Chapter 18

Trees

Question: Why are we considering a new data structure?

- Inserting into an unsorted array or linked list is $\Theta(1)$, but searching is $\Theta(n)$ worst case.
- Inserting into a sorted array list takes $\Theta(n)$ worst case, but searching takes $\Theta(\log n)$ in the worst case for array or $\Theta(n)$ for linked list.

Question: Why are we considering a new data structure?

- Inserting into an unsorted array or linked list is $\Theta(1)$, but searching is $\Theta(n)$ worst case.
- Inserting into a sorted array list takes $\Theta(n)$ worst case, but searching takes $\Theta(\log n)$ in the worst case for array or $\Theta(n)$ for linked list.

Answer: In this topic we introduce **Trees** which:

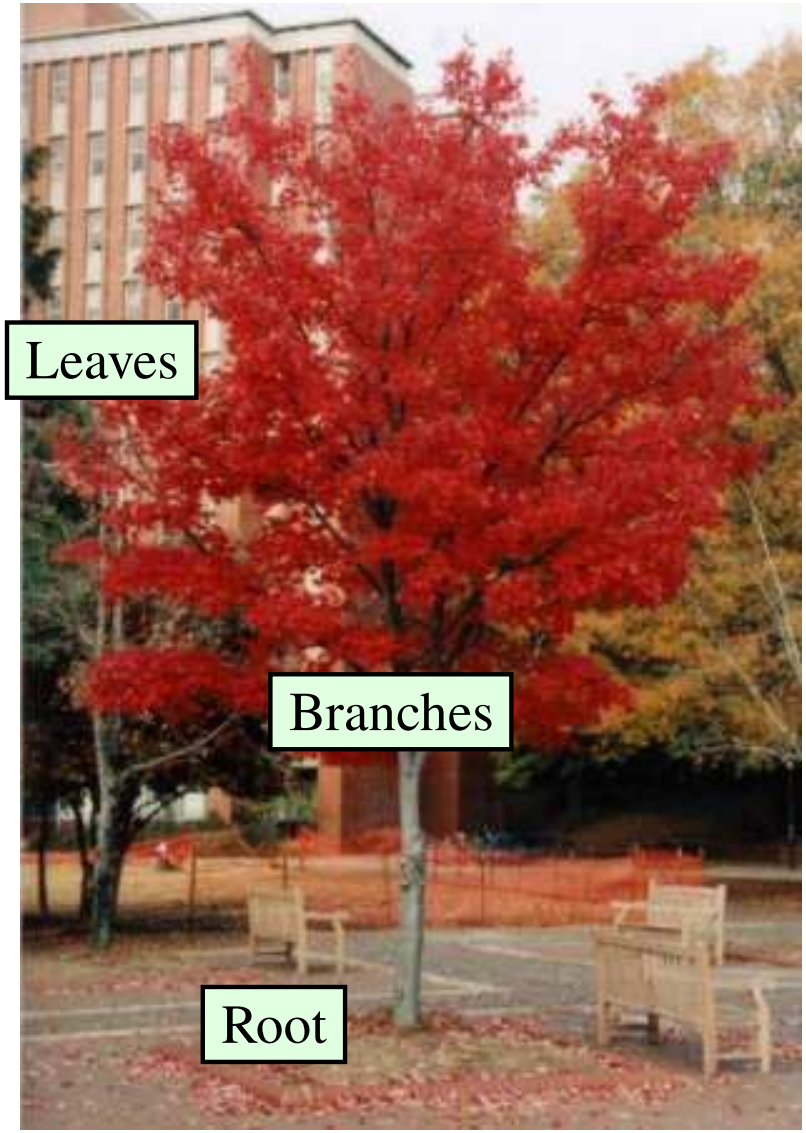
- are simple data structures
- most operations can be done in $\Theta(\log n)$ operations on average

Trees

- Trees are mathematical abstractions which model **hierarchies**.
- Based on trees we will define efficient data structures for
  - the **searching problem**: Binary Search Trees - insert, delete, search in $\Theta(\log n)$ average running time
  - **priority queues**: Heaps - insert, findMin in $\Theta(\log n)$ worst-case running time.

Trees – Terminology

☞ Trees are a data structure which resemble ...

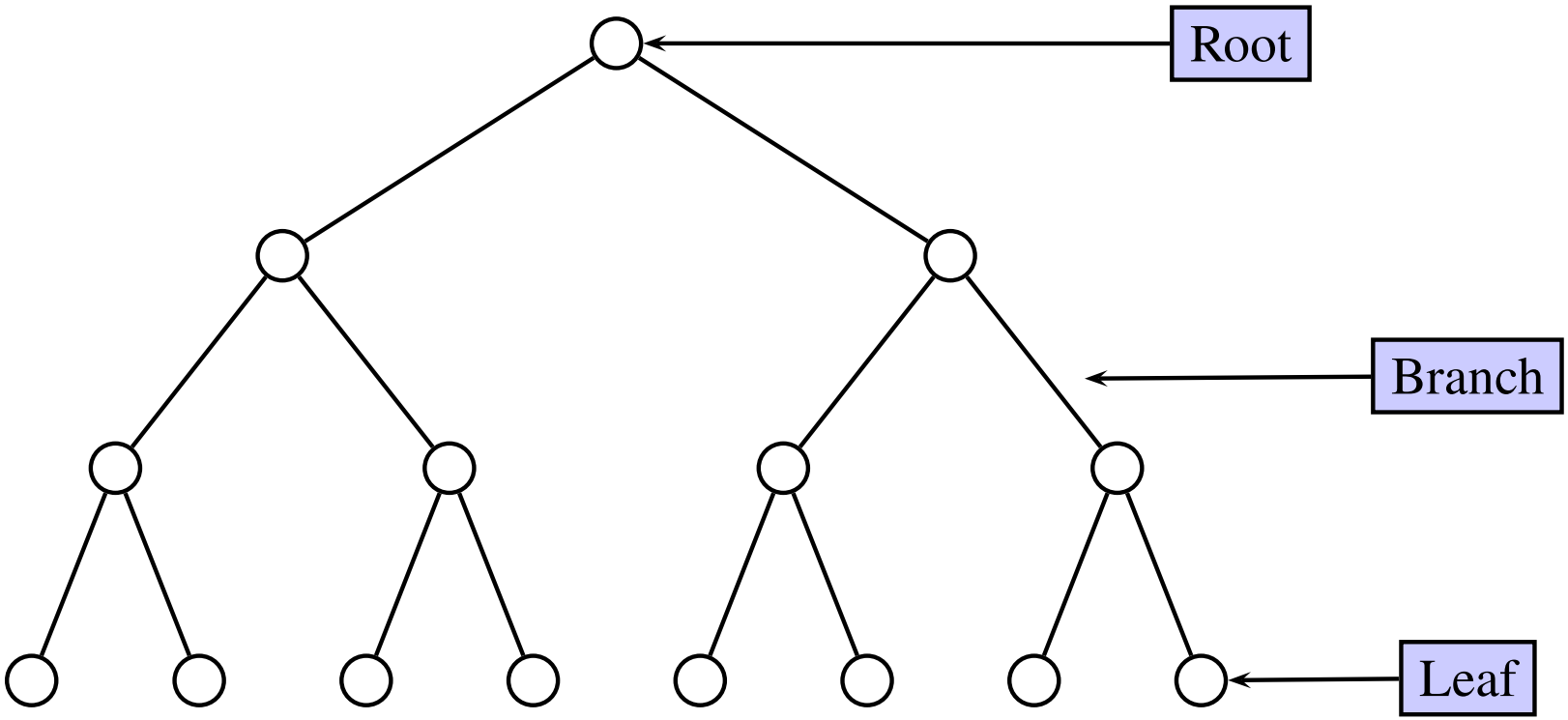


Trees – Terminology

☞ However, software designers typically consider **inverted** trees

Trees – Terminology

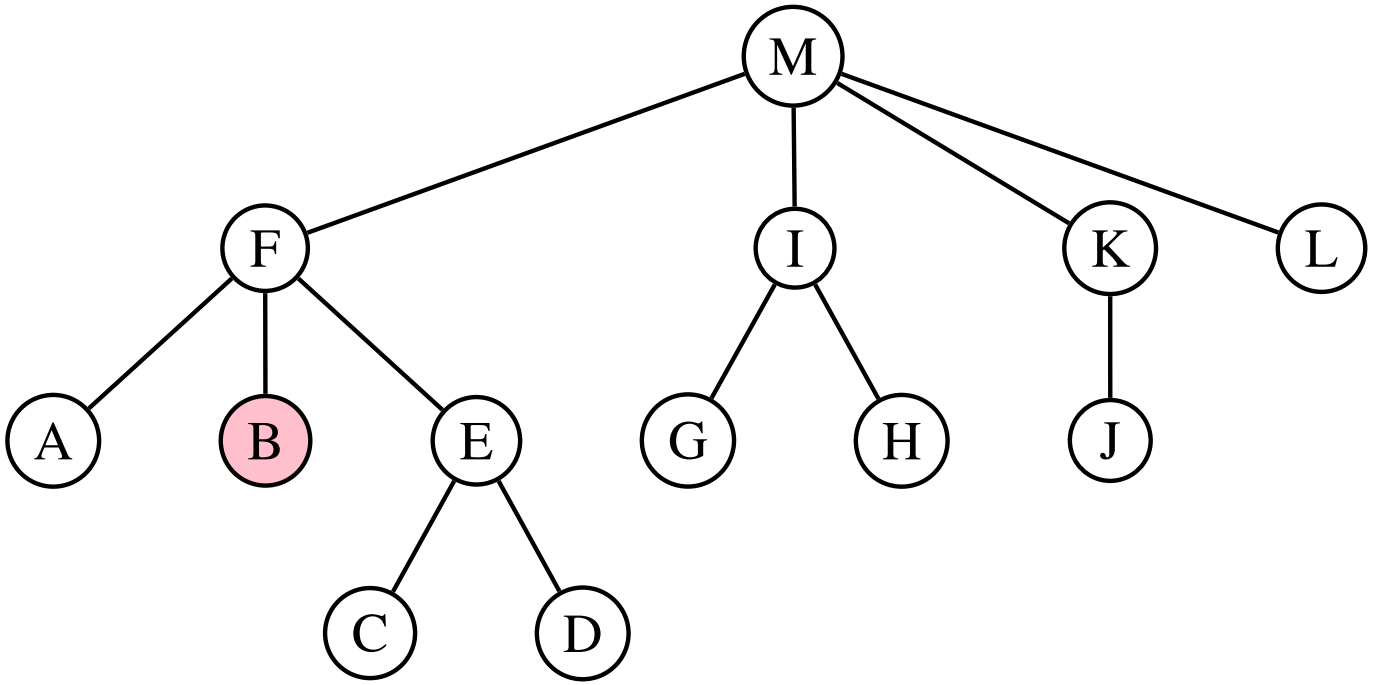
- A **tree** consists of a series of nodes and edges or branches
- There is a designated node called the **root** of the tree.
- The remaining nodes (if any) are grouped into sets of distinct nodes, each of which is a tree
- The root of each subtree is connected to the root of the tree by an edge



Trees – Recursive Definition

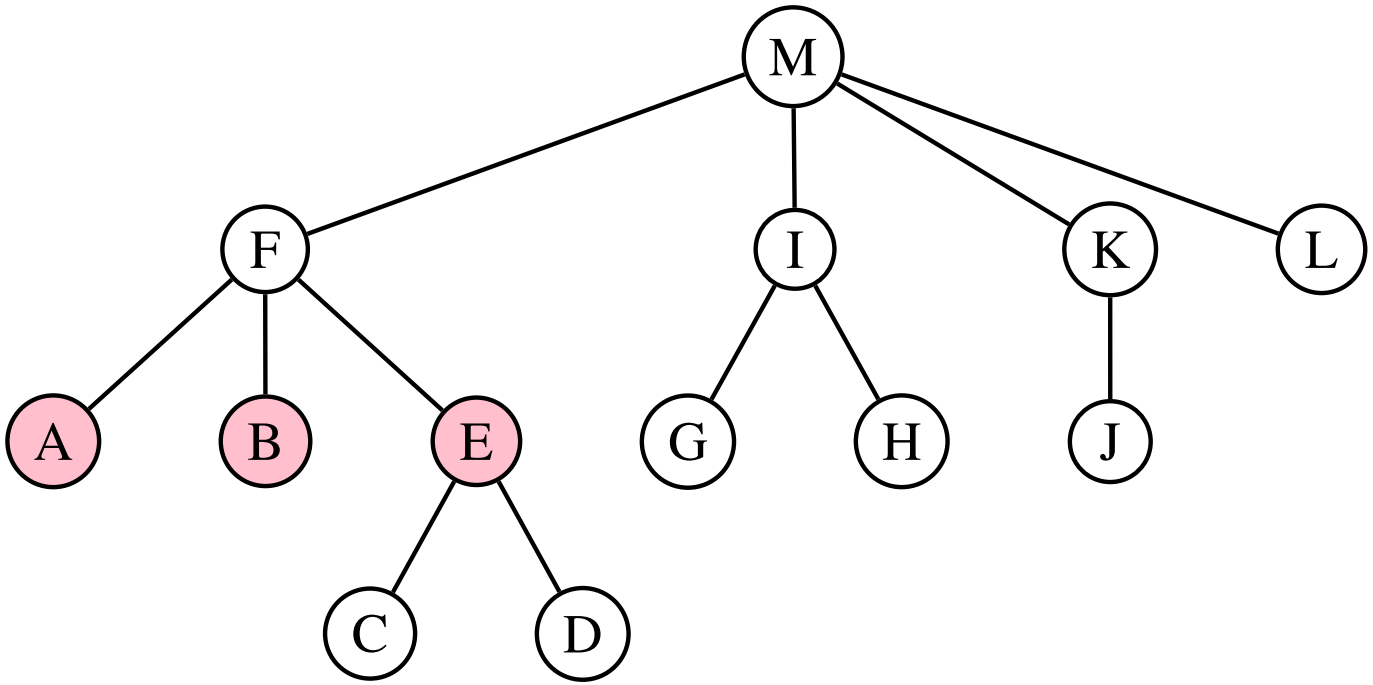
- A **(rooted) tree** consists of a series of nodes and edges or branches, defined as follows:
  - A tree is either the empty set or
  - it consists of a node called the **root** and
  - zero or more non-empty subtrees $T_1, T_2, \cdots, T_k$
    - the root of each subtree is connected to the root of the tree by an edge (or branch).

Example:



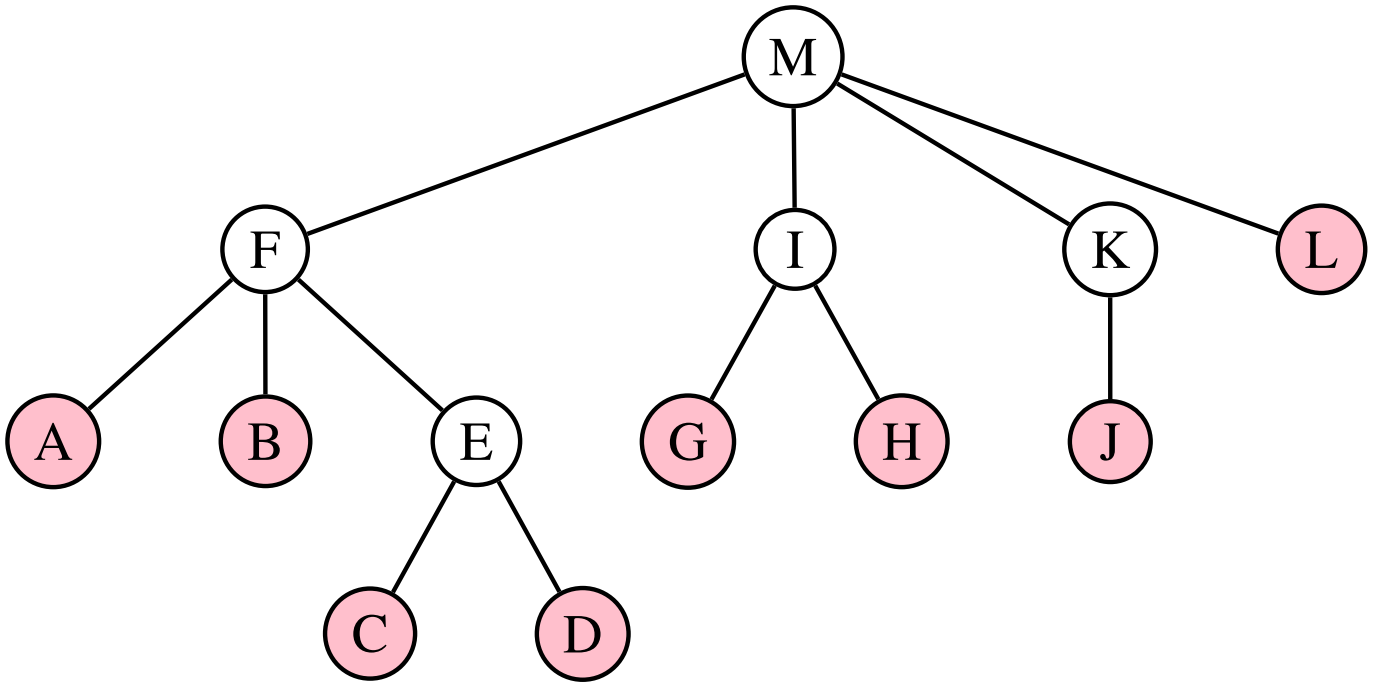
- If v is a node, other than the root, then the **parent** of v is the unique node that must be accessed last to arrive at v starting from the root via the branches.
  - The parent of node B is F
  - Every node other than the root has exactly one parent.

Example:



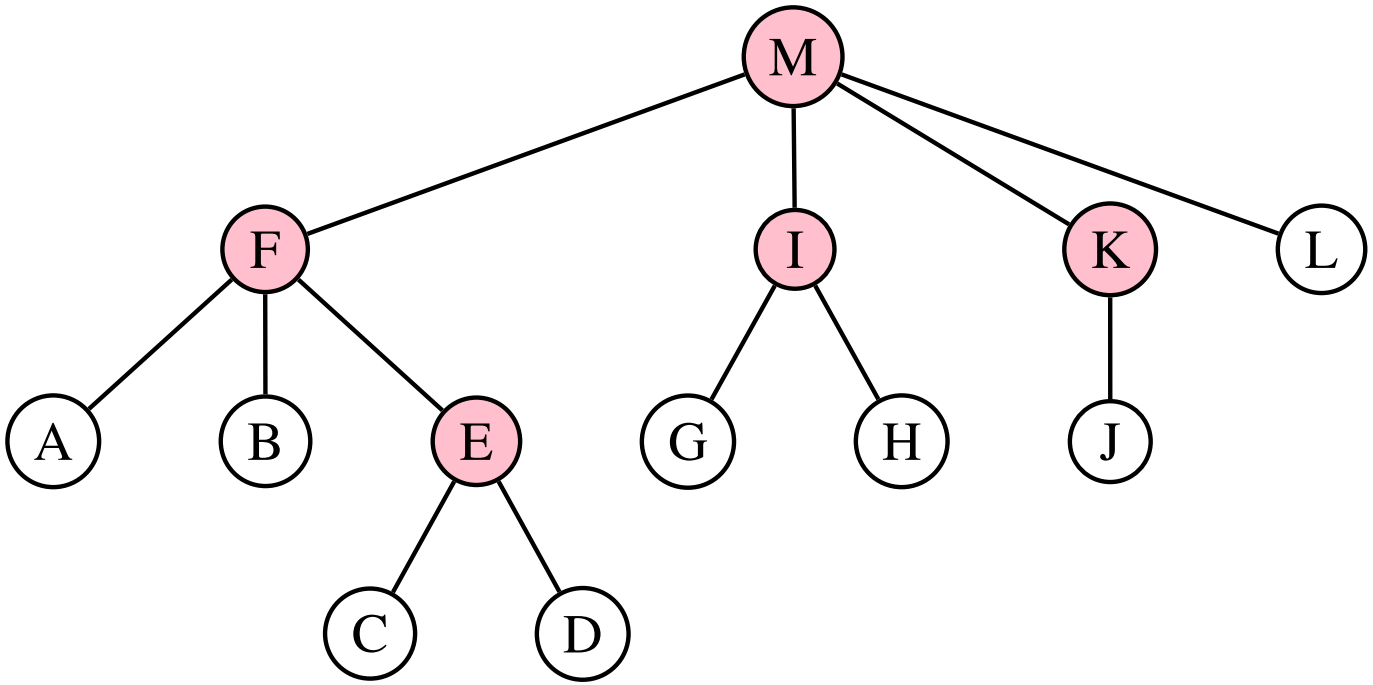
- If v is the parent of a node u, then u is the **child** of v.
  - The parent of nodes A, B, E is F, therefore they are children of F
  - Nodes sharing the same parent are called **siblings**, i.e., A, B, E are siblings.

Example:



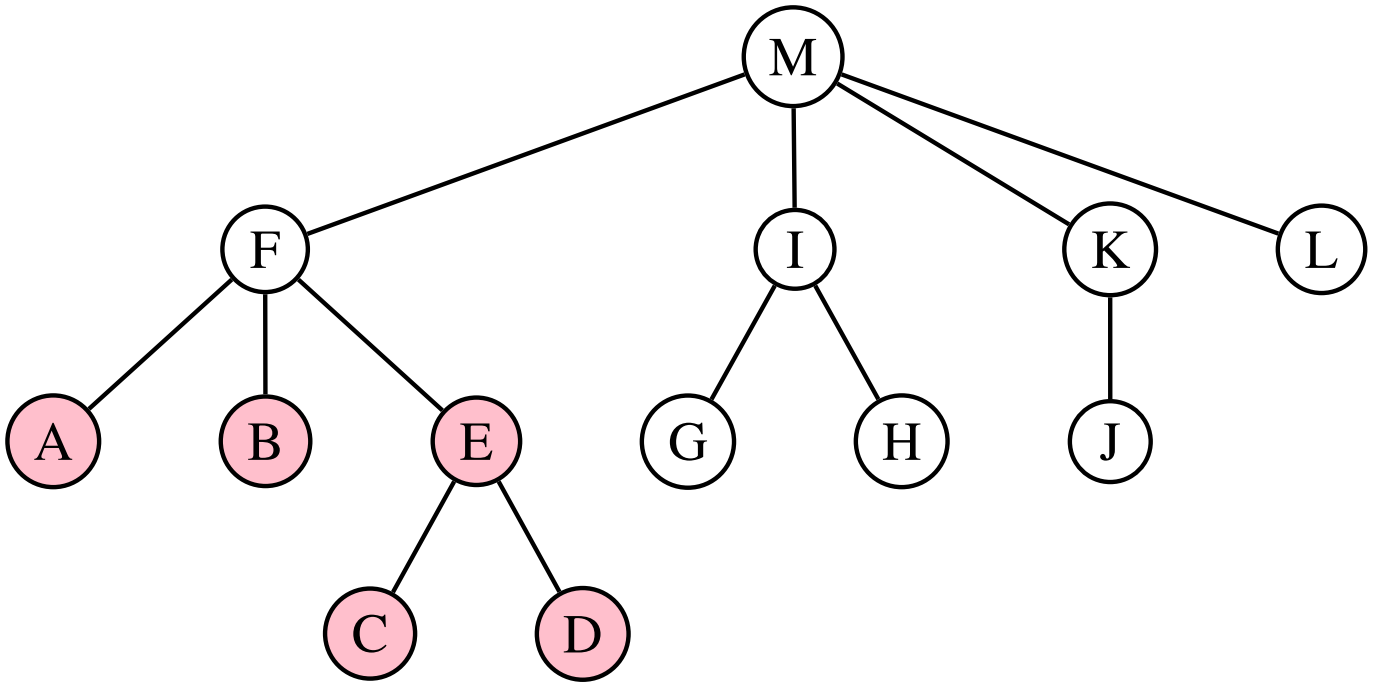
- A **leaf** is a node with no children.
  - The leaves in the tree are A, B, C, D, G, H, J, L

Example:



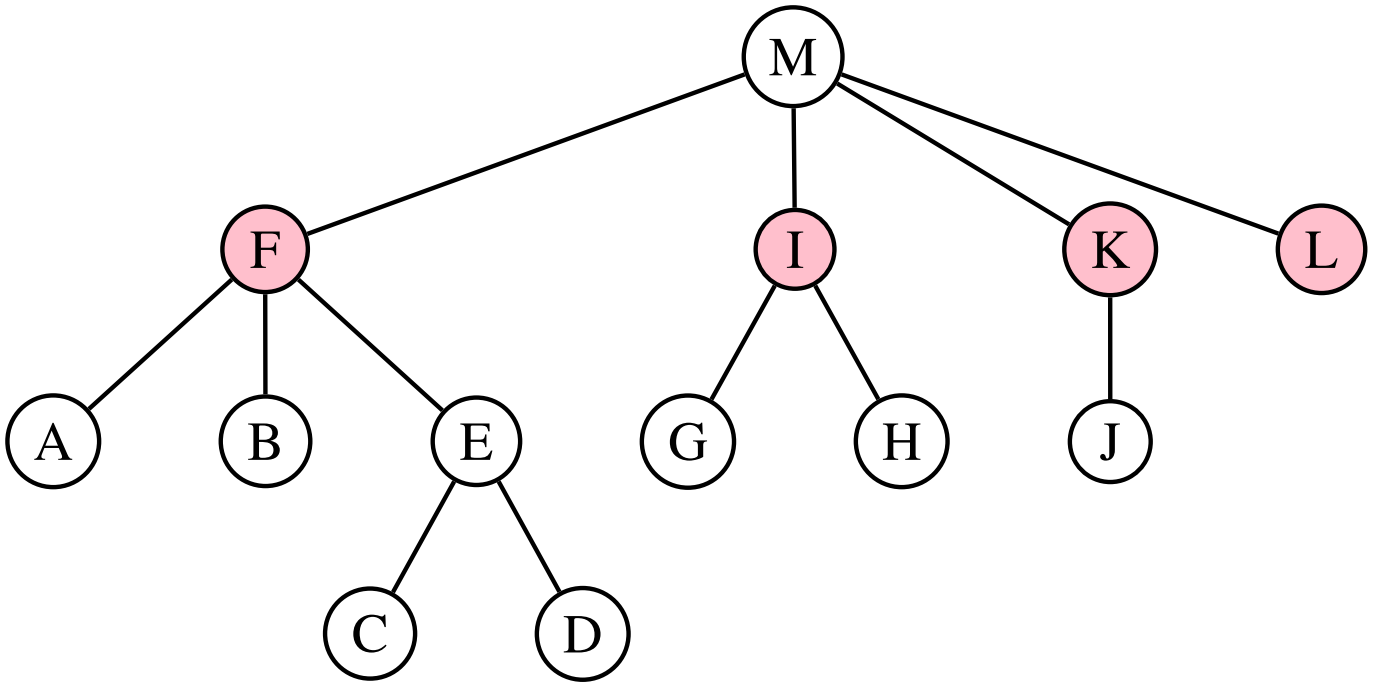
- An **internal** node is one with children.
  - The internal nodes in the tree are E,F,I,K,M

Example:



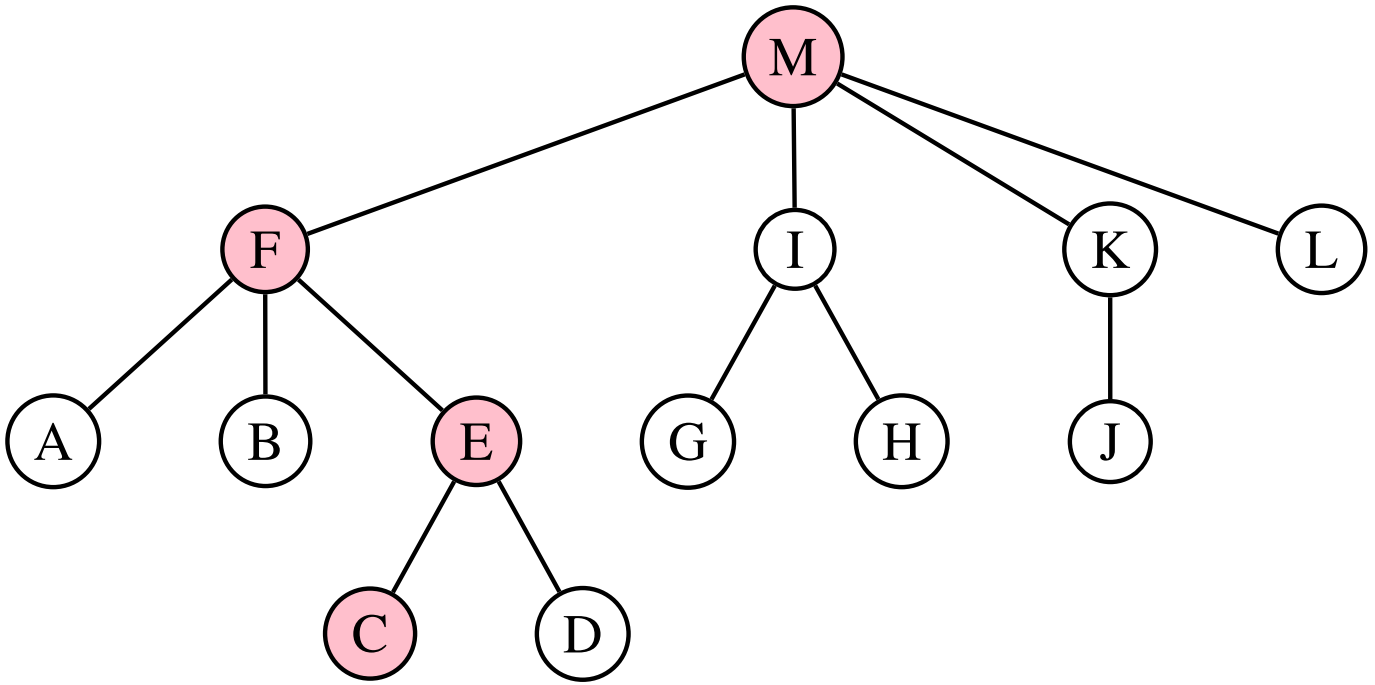
- The **degree of a node** is the number of edges incident with it
  - The degree of F is 3

Example:



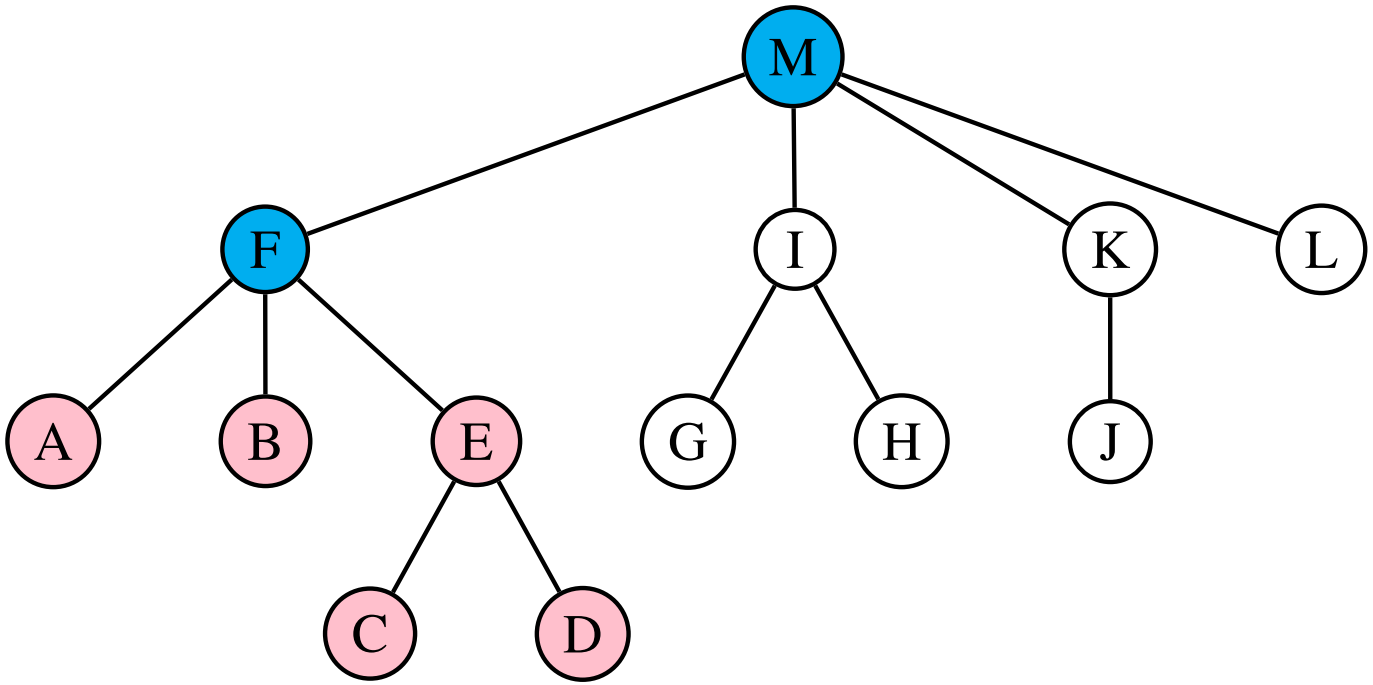
- The **degree of a tree** is the maximum degree of any node.
  - The degree of this tree is 4.

Example:



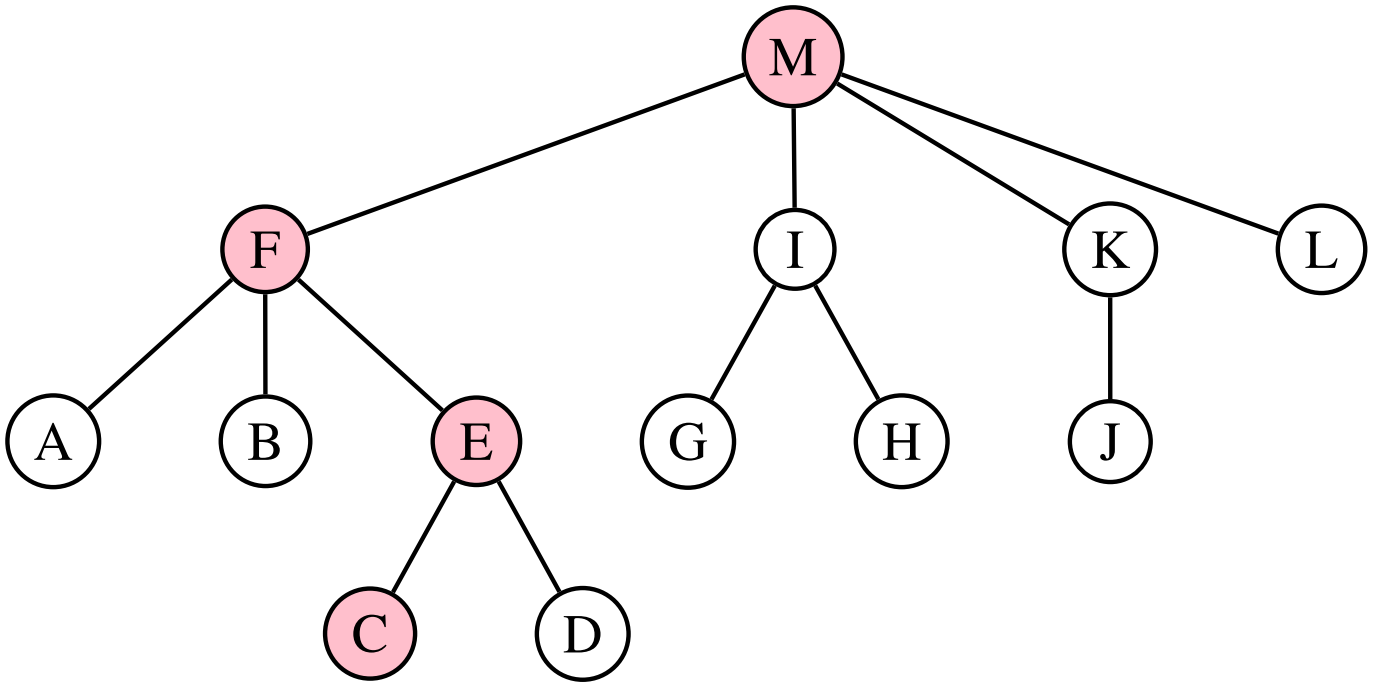
- A **path** of length $k-1$ from node n_1 to n_k is defined as the sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1}.
  - Consider the path of length 3 from the root: M, F, E, C

Example:



- If there is a path from node u to node v then u is an **ancestor** of v and v is a **descendant** of u. If $u \neq v$ then u is a **proper ancestor** of v and v is a **proper descendant** of u.
  - All proper descendants of F are A,B,E,C and D
  - All proper ancestors of E are F and M

Example:



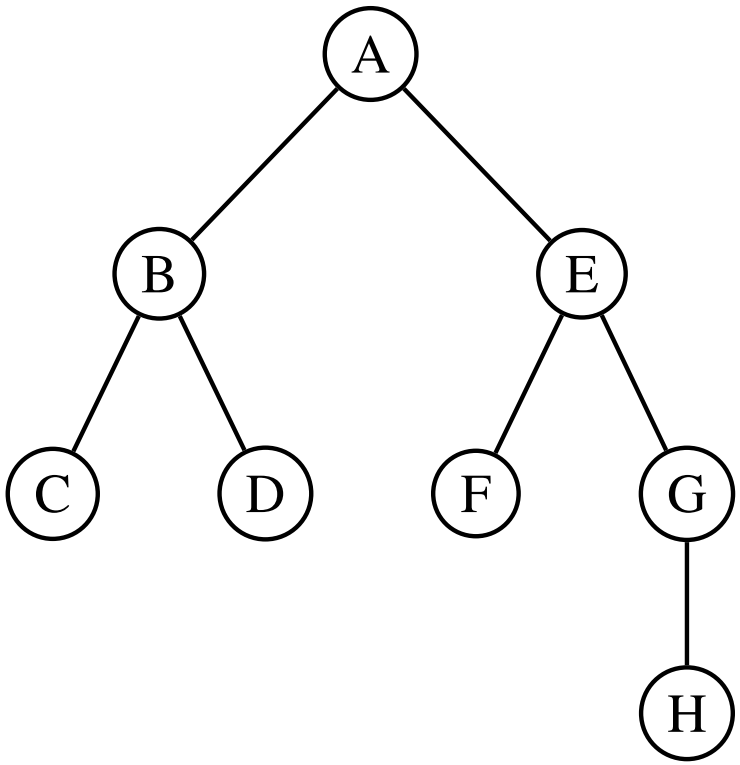
- The **depth** of a node is the length of the unique path from the root. Notice that depth(child(x))=depth(x)+1. The **height** of a tree is the length of the longest path to a leaf.
  - depth(M) = 0, depth(F)=1, depth(C)=3, Tree height = 3

Trees – Properties

- A tree does not have any *cycles*, i.e., it is impossible to leave a node and return to the same node without going over an edge twice
- Trees are minimally connected, i.e., removing any edge breaks a tree into two trees.
- The path between any two nodes in the tree is **unique**, i.e., there is only one way to get from one node to another.
- A tree with n nodes has $n - 1$ edges

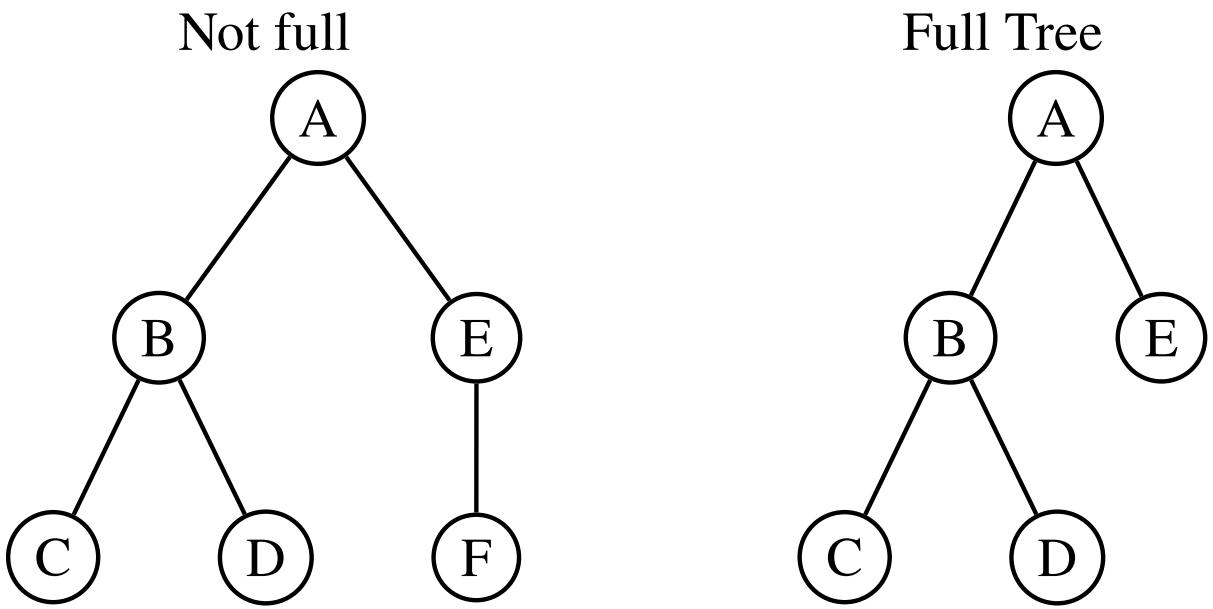
Binary Trees

→ A **binary tree** is a tree where each node has at most 2 children.



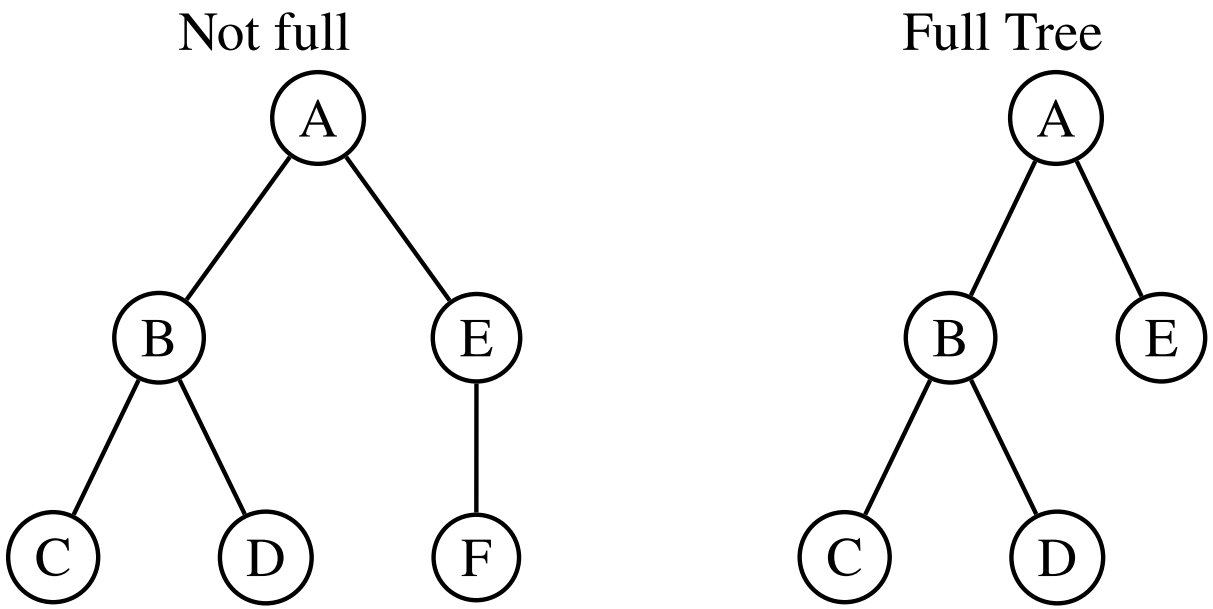
➠ This tree has $n = 8$ nodes, $n - 1 = 7$ edges and 4 leaves

→ A **full** binary tree is one in which **every** internal node **has two children**.



Binary Trees

→ A **full** binary tree is one in which **every** internal node **has two children**.

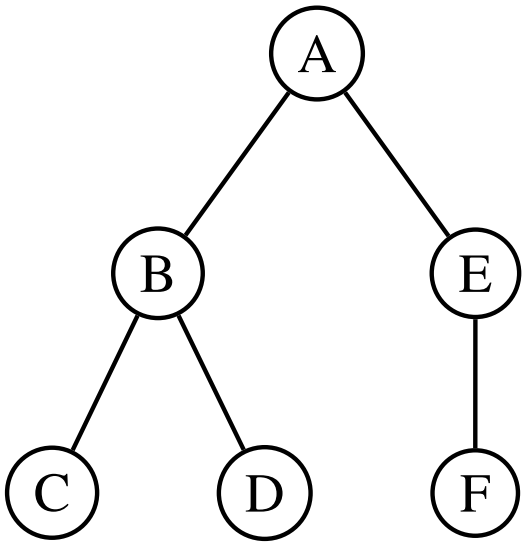


➠ A full binary tree with n vertices has

- → $n-1$ edges
- → $(n-1)/2$ internal vertices
- → $(n+1)/2$ leaves

Binary Trees

→ There are **at most** 2^d nodes at a depth of d in a binary tree.



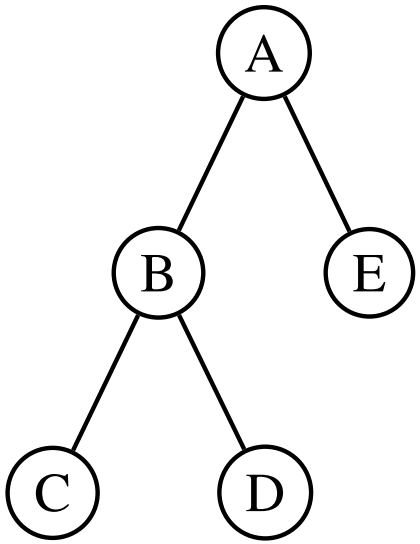
At most 2^0 nodes at depth 0

At most 2^1 nodes at depth 1

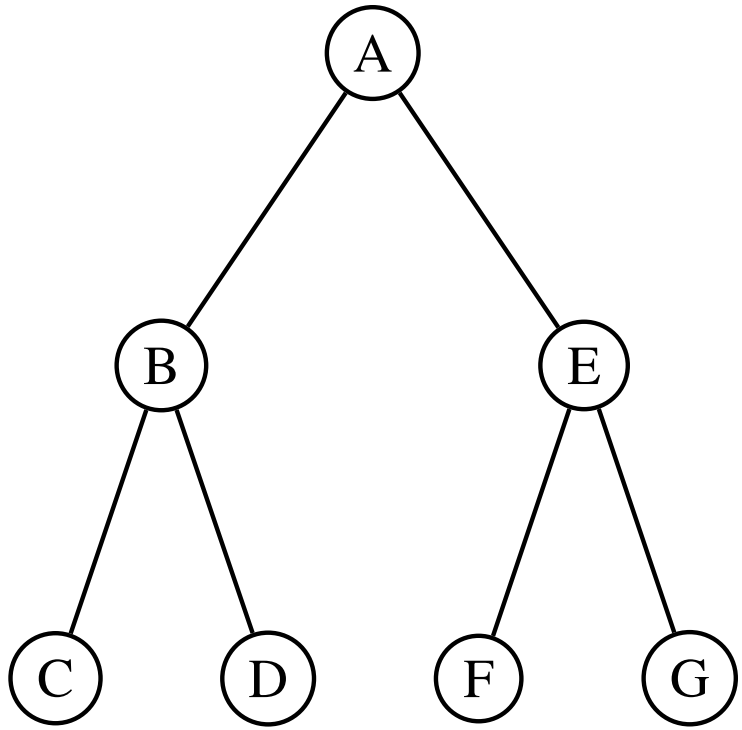
At most 2^2 nodes at depth 2

→ A **complete** binary tree is a full binary tree where every leaf is at the same depth.

Full but Not Complete Tree



Complete Tree



→ For a binary tree of height h, the number of nodes can be bounded as,

$$h + 1 \leq n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

→ The upper bound is achieved when the tree is complete.
→ The lower bound is achieved when every internal node has only one child.

→ Conversely, for a tree with n nodes, the height satisfies

$$n - 1 \geq h \geq \lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2 n \rfloor$$

where $\lceil x \rceil$ is the ceiling operator and returns the closest integer greater than or equal to x

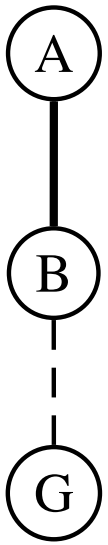
→ Conversely, for a tree with n nodes, the height satisfies

$$n - 1 \geq h \geq \lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2 n \rfloor$$

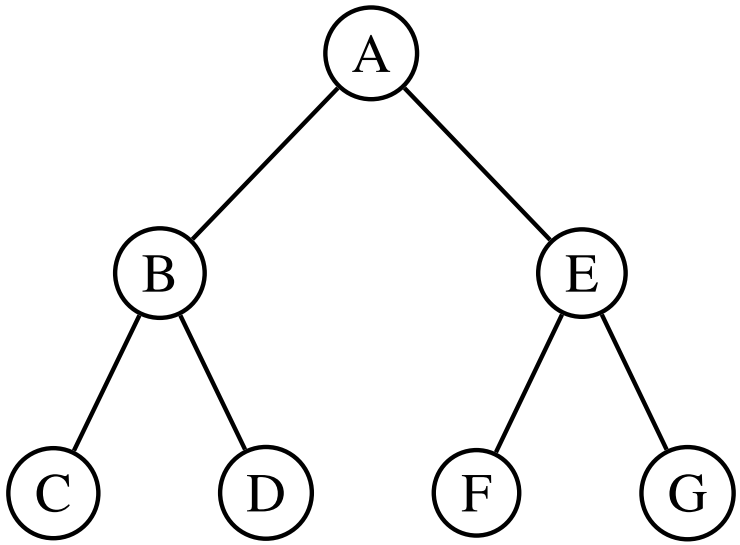
where $\lceil x \rceil$ is the ceiling operator and returns the closest integer greater than or equal to x

→ Thus, a tree which has all levels completely filled except possibly for the last one, **minimizes** the height of the binary tree with n nodes.

Example: Consider the following two binary trees both with 7 nodes



Height = 6



Height = 2

General Tree ADT operations

There are many operations that we can specify on an tree.

- → add a node
- → delete a node
- → access data in some node
- → change data in some node
- → access a child of a node
- → print the values stored in tree nodes
- → ...
- ☞ Notice that the ADT definition does not tell us **how** to implement these routines.

Array Implementation of Binary Trees

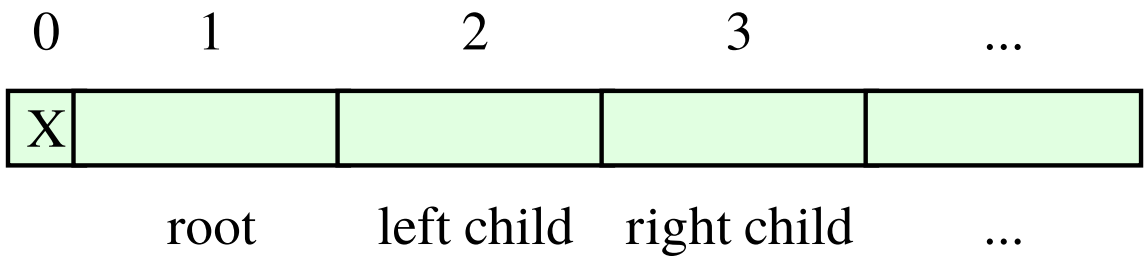
Question: How do you implement the data structure for trees?

Array Implementation of Binary Trees

Question: How do you implement the data structure for trees?
Answer: One way is to use an array.

Consider a binary tree. Store root at index 1.

- For a node stored at index i
  - leftchild$(i) = 2i$
  - rightchild$(i) = 2i + 1$
- If any of the children is missing, mark the corresponding array element as empty.

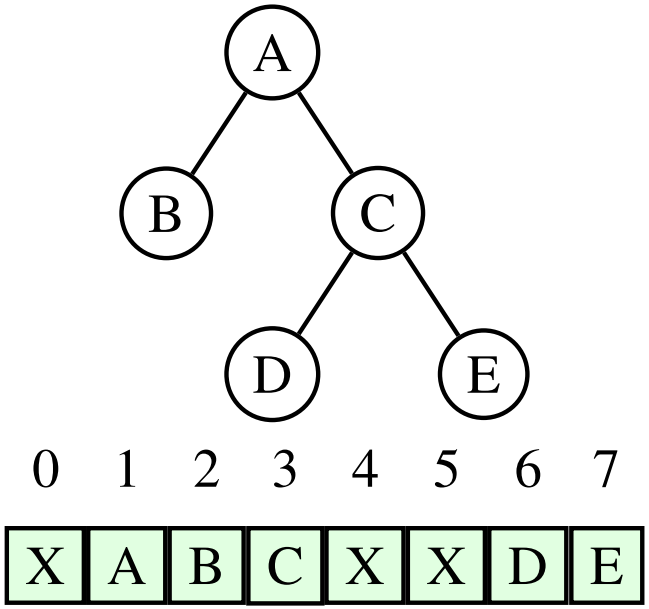


- Notice that parent$(i) = \lfloor \frac{i}{2} \rfloor$
- In `Java`, the array elements contain a reference to an existing object for valid nodes, otherwise a `null` reference (to represent the 'X' character).

Array Implementation of Binary Trees

Example:

$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor, \quad \text{leftchild}(i) = 2i, \quad \text{rightchild}(i) = 2i + 1$$



- Fill the array from left to right with the nodes of a complete binary tree, in *level order*.
  - → Level order: root, nodes on level 1 from left to right, nodes on level 2 from left to right ...
- If the tree is not complete, more the array positions corresponding to missing nodes as empty.

Array Implementation of Binary Trees

Advantage:

- The tree operations are very fast.
- The array implementation can be very memory efficient, especially if the tree is complete.

Disadvantage:

- Can be wasteful in space since you need to allocate an array as if the tree was complete.

Linked Implementation of Binary Trees

Solution: In order to provide better memory performance for trees with are not complete, consider a linked-list based implementation.

Reference to left child

reference to right child

Notice that there is no reference to the parent.

Linked Implementation of Binary Trees

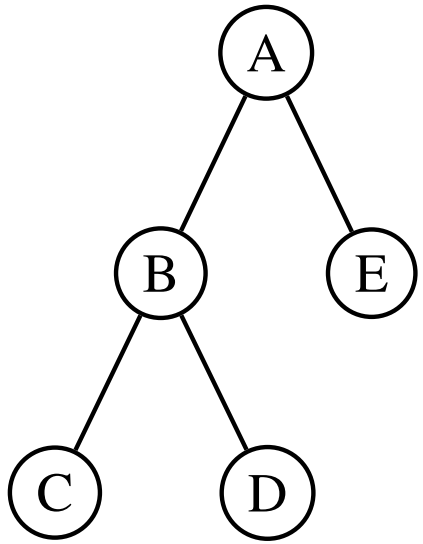
Each node of the tree can be represented using the class:

```
1  class TNode<E>{
2      E element;
3      TNode<E> left;
4      TNode<E> right;
5
6      public TNode(E e, TNode<E> l, TNode<E> r){
7          element=e;
8          left=l;
9          right=r;
10     }
11 }
```

- Each element of the tree is then of type `TNode`
- Requires some overhead in order to store the references to left and right children

Tree Traversal

There are three primary ways to visit all nodes of the tree, or to **traverse** the tree.
Example: Consider the tree



- **Pre-Order** traversal - work is done at each node before progressing recursively to children. In example, traversal order: ABCDE
- **In-Order** traversal - work is done recursively on the left child, followed by the node and then the right child. In example, traversal order: CBDAE
- **Post-Order** traversal - work at each node after the children are recursively evaluated. In example, traversal order: CDBEA

Tree Traversal

```
1  public void preOrder(TNode<E> subroot){
2      if (subroot != NULL){
3          visit(subroot); //Perform action, e.g., print(subroot.element)
4          preOrder(subroot.left);
5          preOrder(subroot.right);
6      }
7  }
```

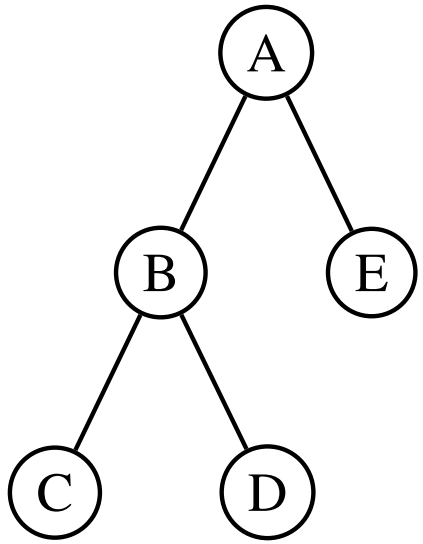
```
1  public void inOrder(TNode<E> subroot){
2      if (subroot != NULL){
3          inOrder(subroot.left);
4          visit(subroot); //Perform action, e.g., print(subroot.element)
5          inOrder(subroot.right);
6      }
7  }
```

Tree Traversal

```
1 public void postOrder(TNode<E> subroot){
2     if (subroot != NULL){
3         postOrder(subroot.left);
4         postOrder(subroot.right);
5         visit(subroot); //Perform action, e.g., print(subroot.element)
6     }
7 }
```

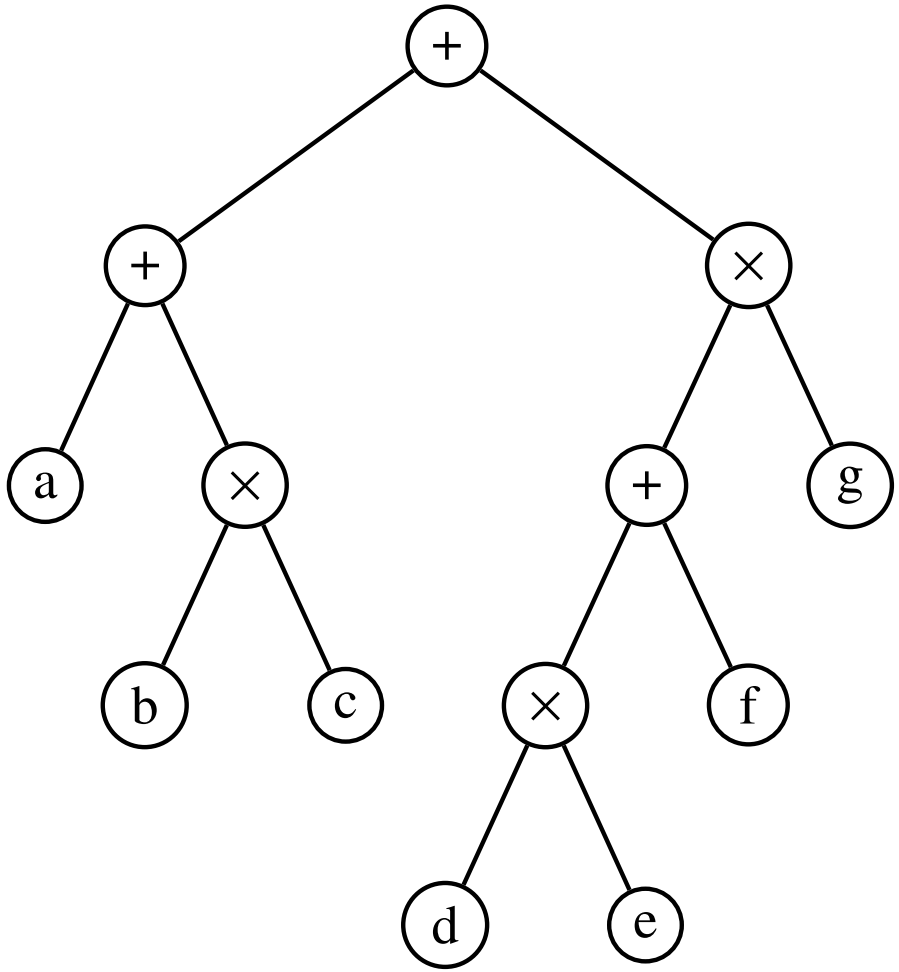
Tree Traversal - Level Order Traversal

Example: Consider the tree



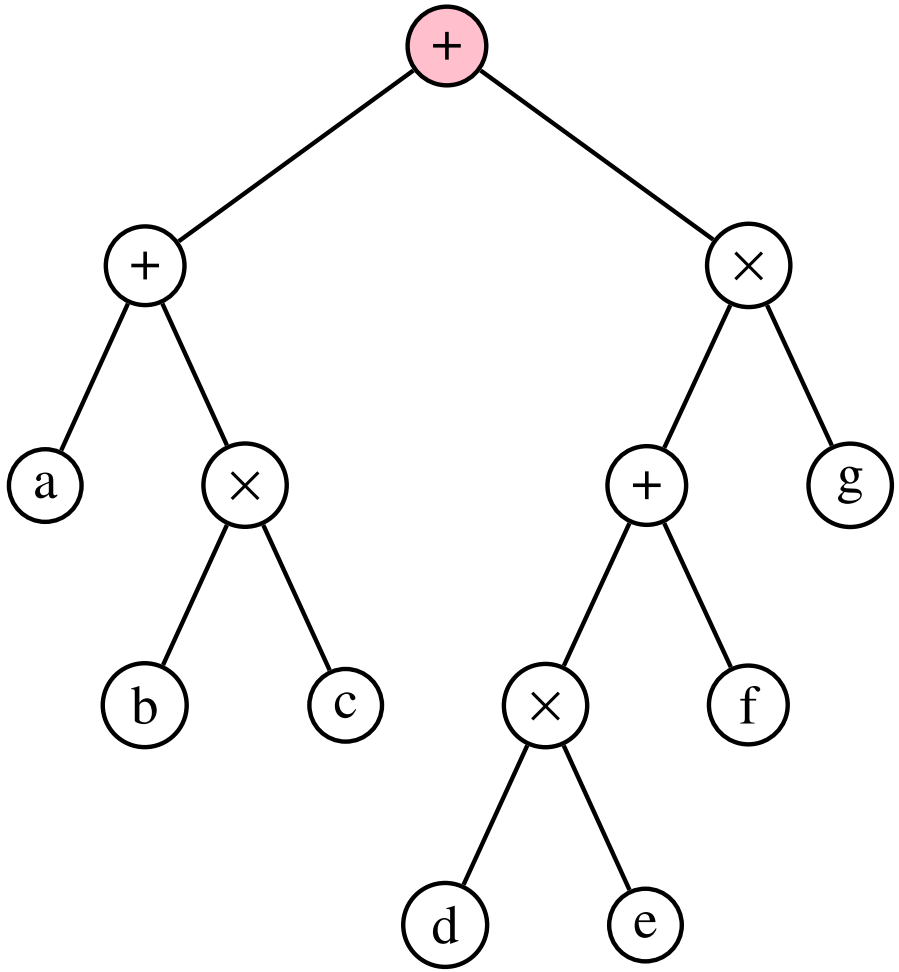
- **Level-Order** traversal:
  - → Visit root, then visit nodes at depth 1, then visit nodes at depth 2, and so on.
  - → Nodes at the same depth are visited in order from left to right.
  - → In example: ABECD

Example: Expression Tree (Pre-Order Traversal) node,left,right



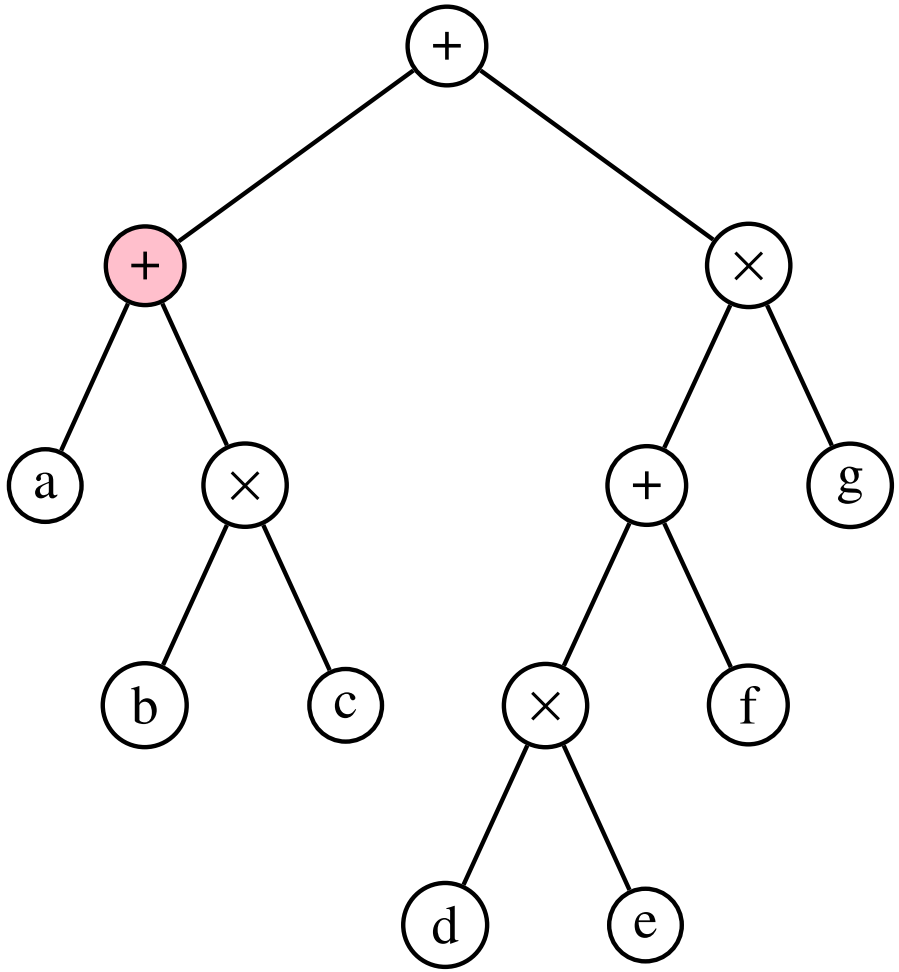
Expression:

Example: Expression Tree (Pre-Order Traversal)



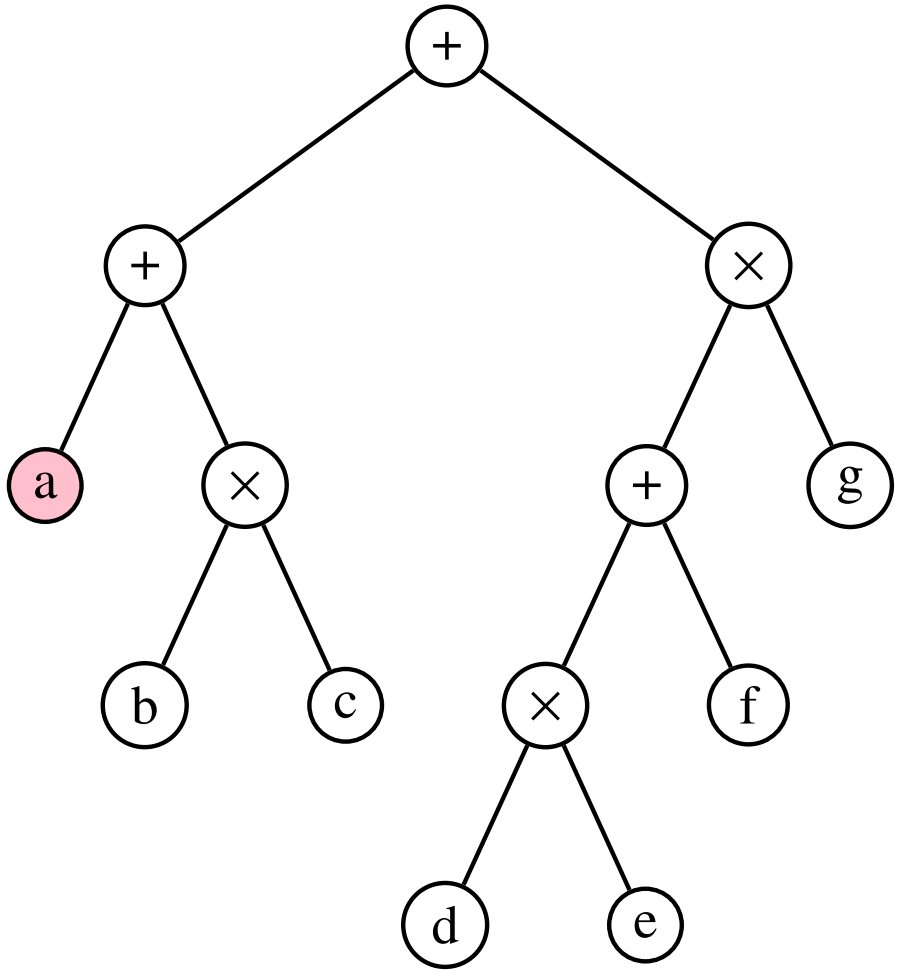
Expression: +

Example: Expression Tree (Pre-Order Traversal)



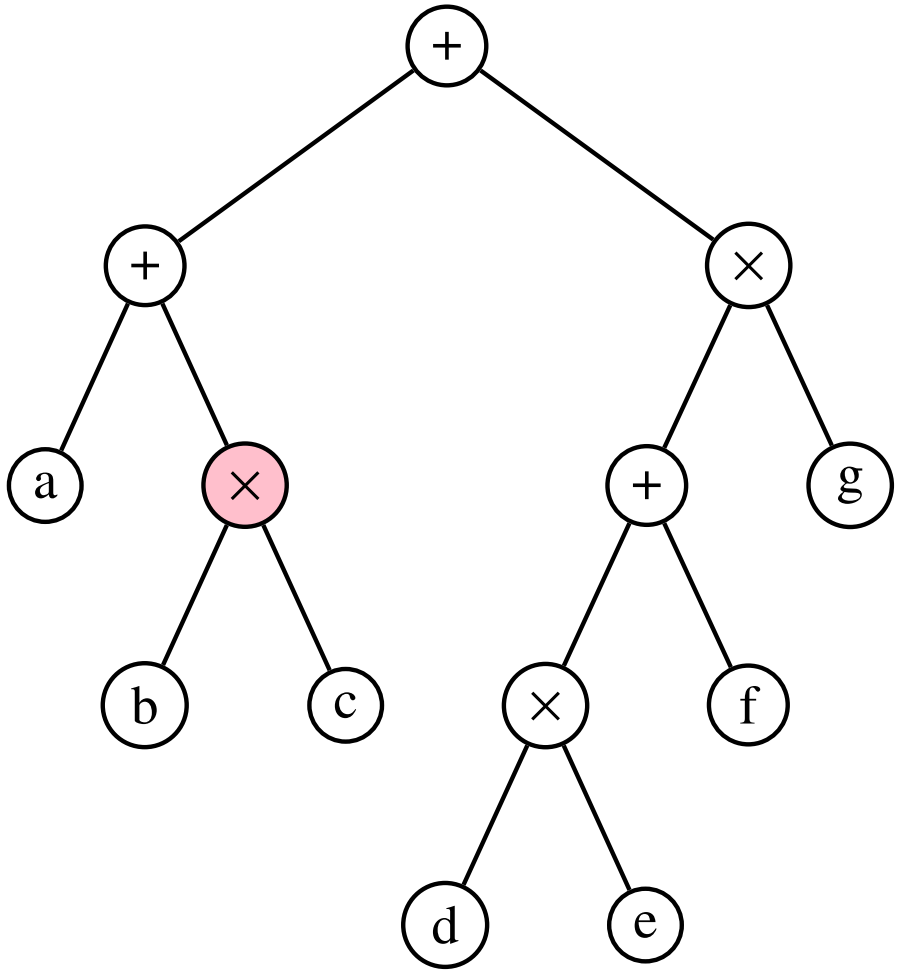
Expression: ++

Example: Expression Tree (Pre-Order Traversal)



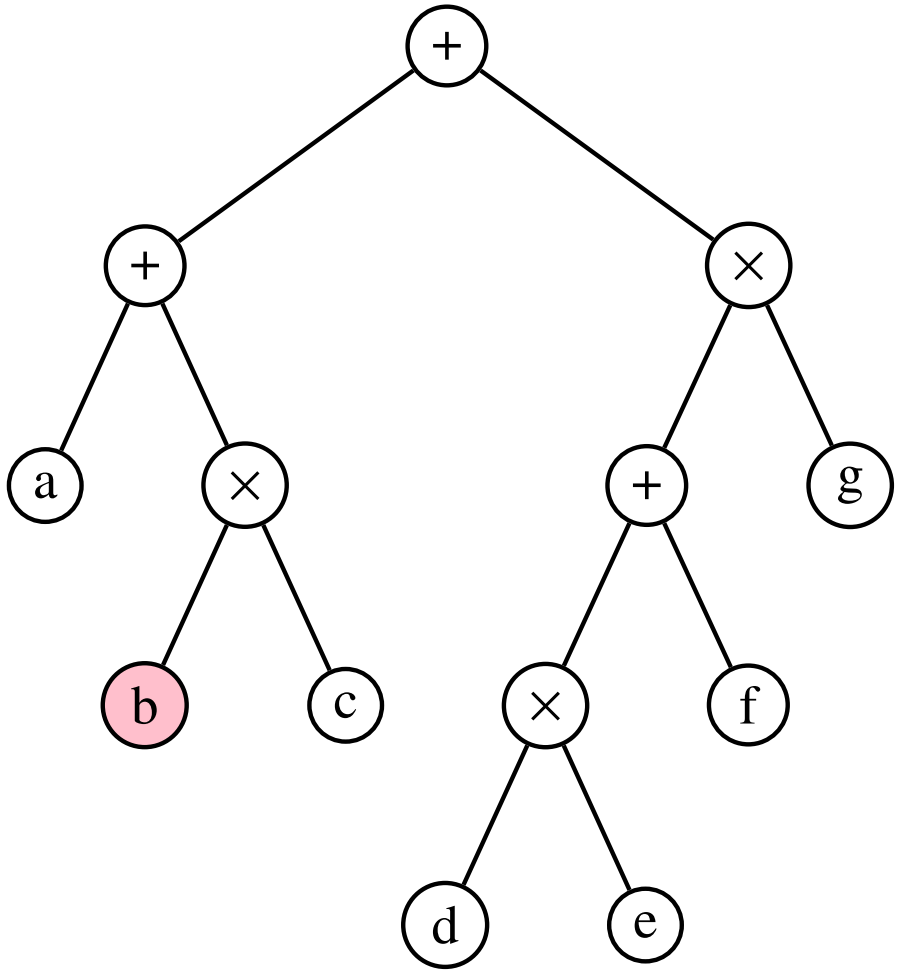
Expression: ++a

Example: Expression Tree (Pre-Order Traversal)



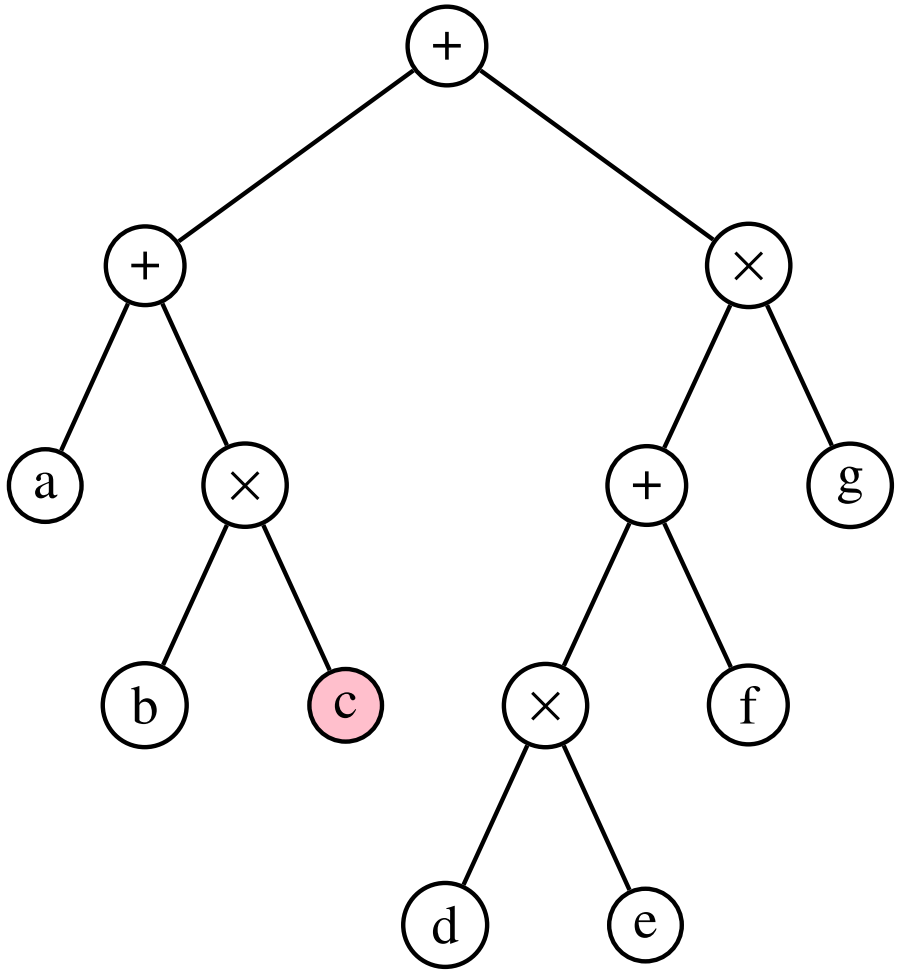
Expression: ++a×

Example: Expression Tree (Pre-Order Traversal)



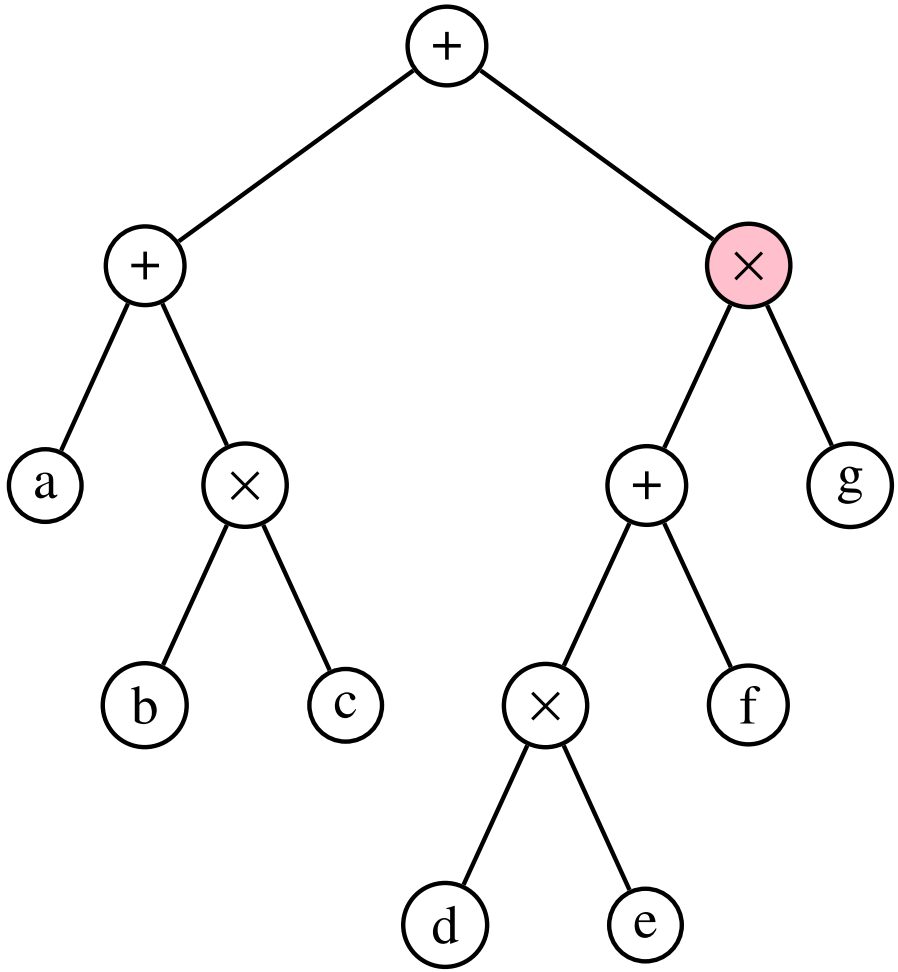
Expression: ++a×b

Example: Expression Tree (Pre-Order Traversal)



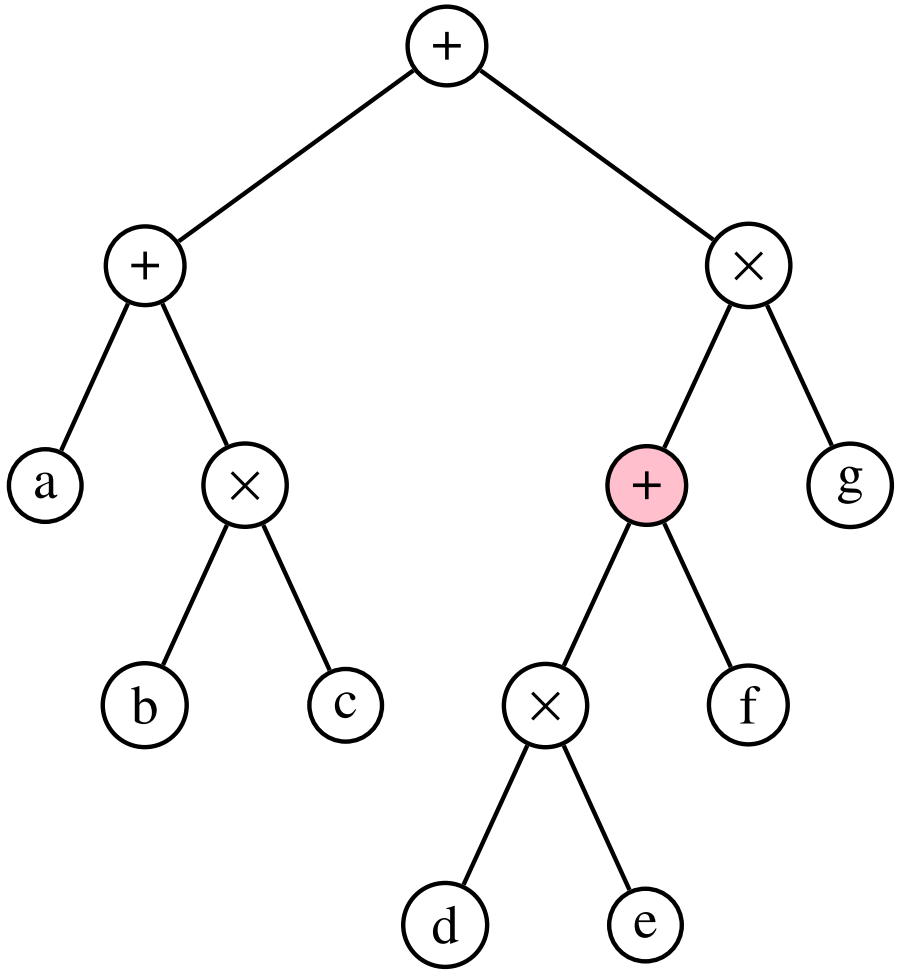
Expression: ++a×bc

Example: Expression Tree (Pre-Order Traversal)



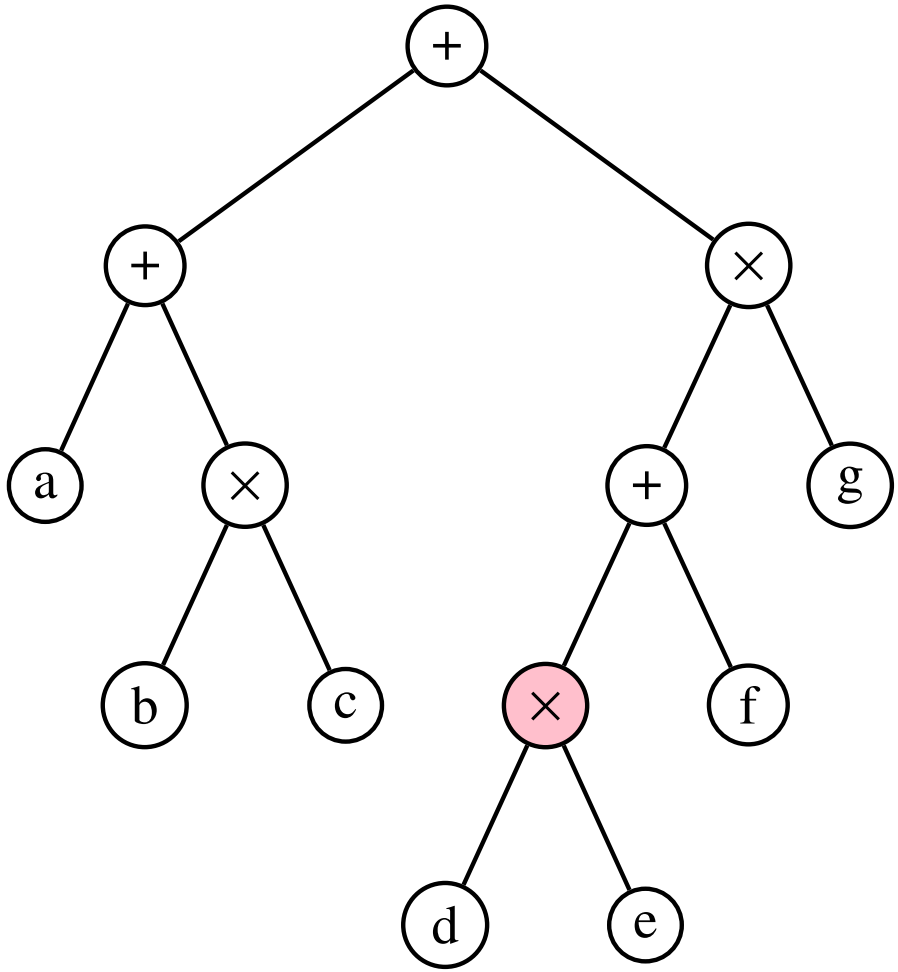
Expression: ++a×bc×

Example: Expression Tree (Pre-Order Traversal)



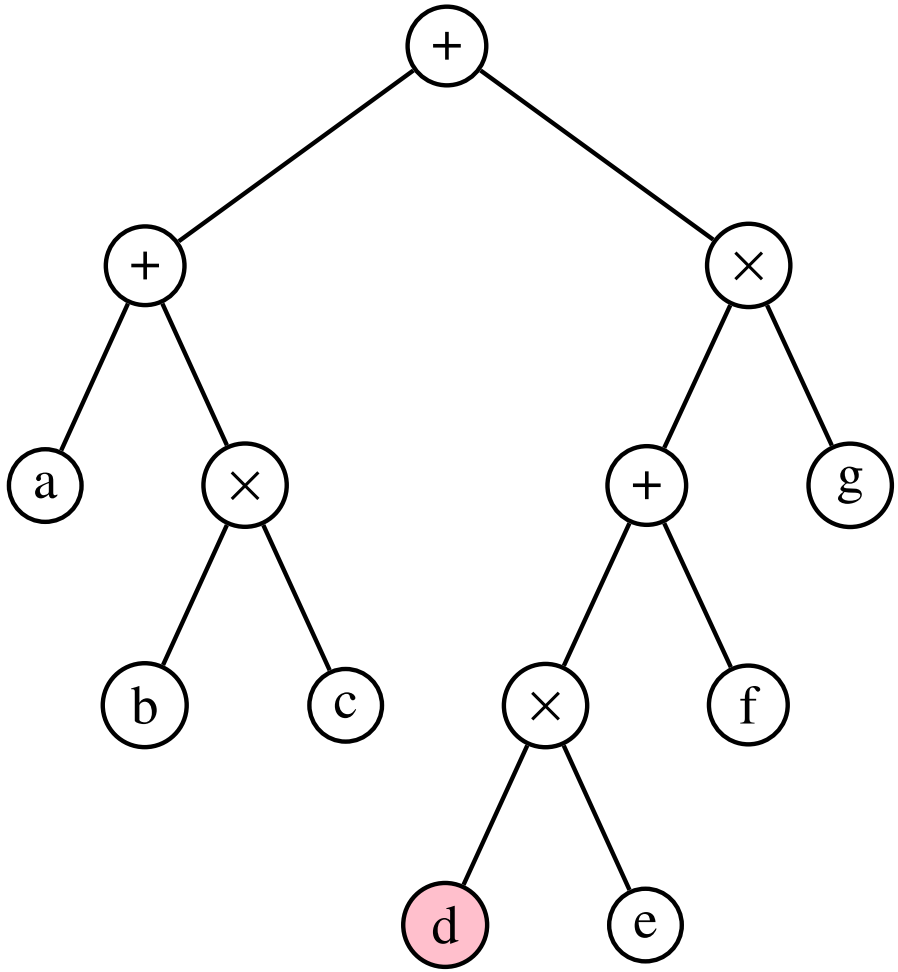
Expression: ++a×bc×+

Example: Expression Tree (Pre-Order Traversal)



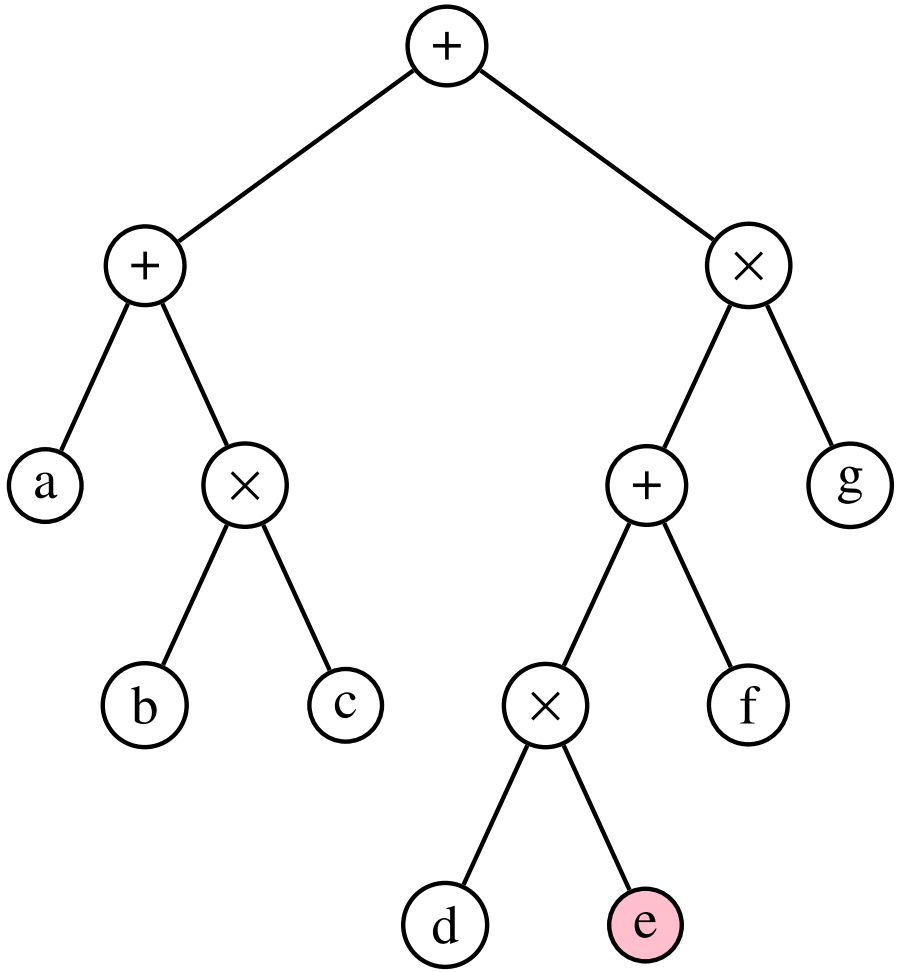
Expression: ++a×bc× + ×

Example: Expression Tree (Pre-Order Traversal)



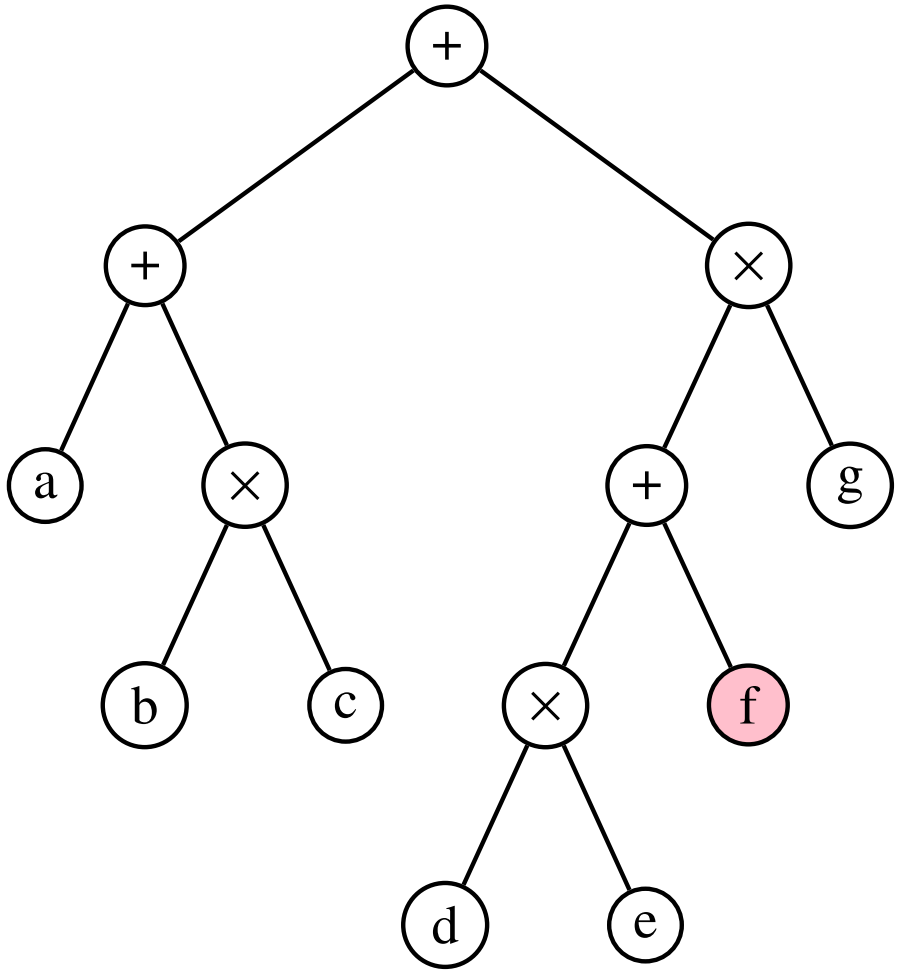
Expression: ++a×bc× + ×d

Example: Expression Tree (Pre-Order Traversal)



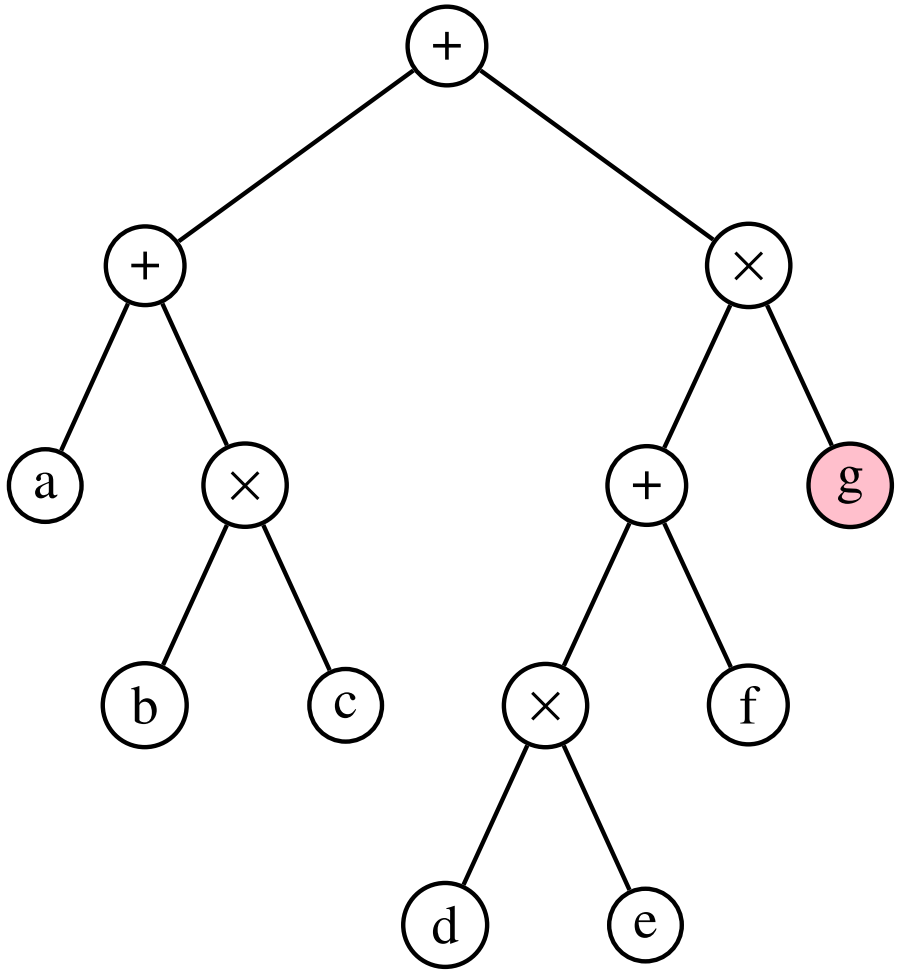
Expression: ++a×bc× + ×de

Example: Expression Tree (Pre-Order Traversal)



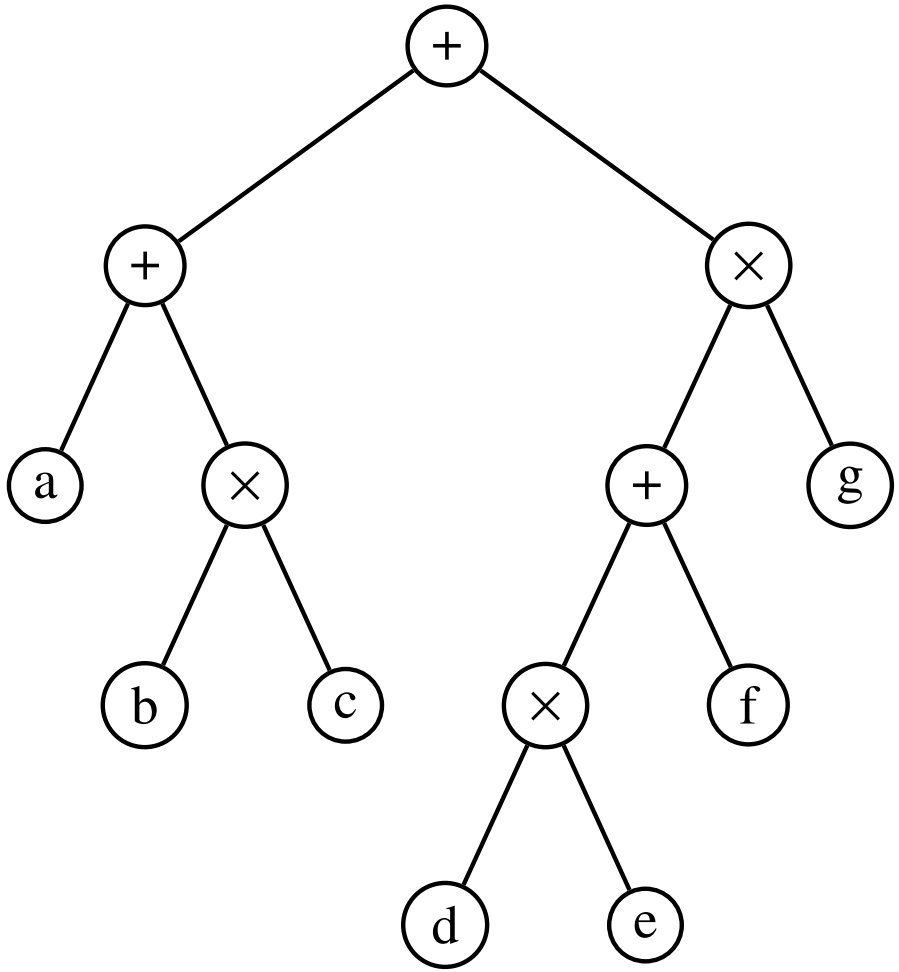
Expression: ++a×bc× + ×def

Example: Expression Tree (Pre-Order Traversal)



Expression: ++a×bc× + ×defg

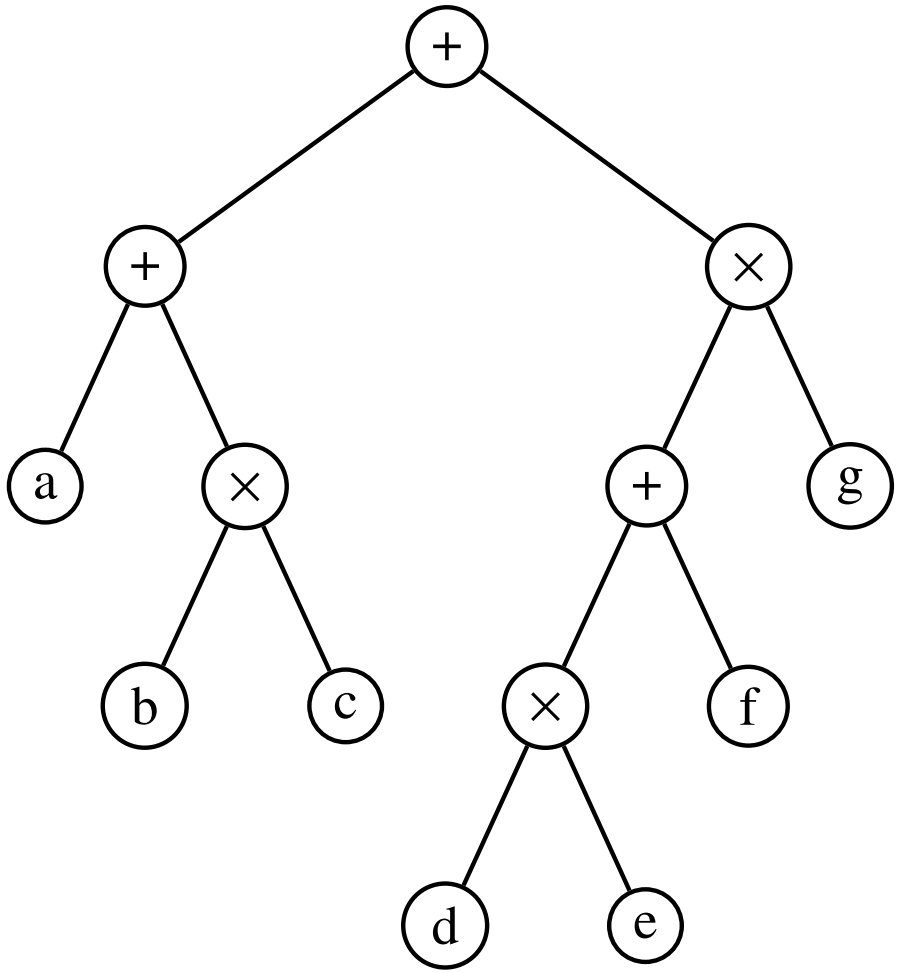
Example: Expression Tree (Pre-Order Traversal)



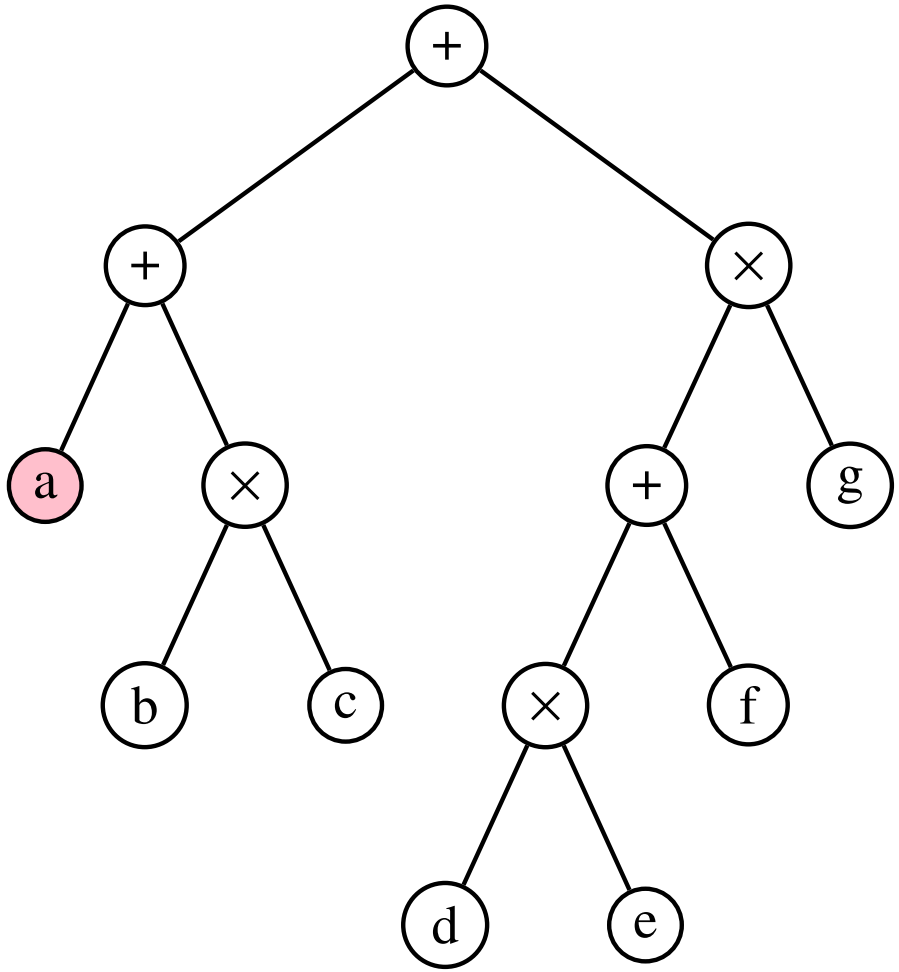
Expression: ++a×bc× + ×defg ⟶ **Prefix Notation**

Example: Expression Tree (In-Order Traversal)

→ **Open bracket**: *descending* to node, **Close bracket**: *ascending* from node

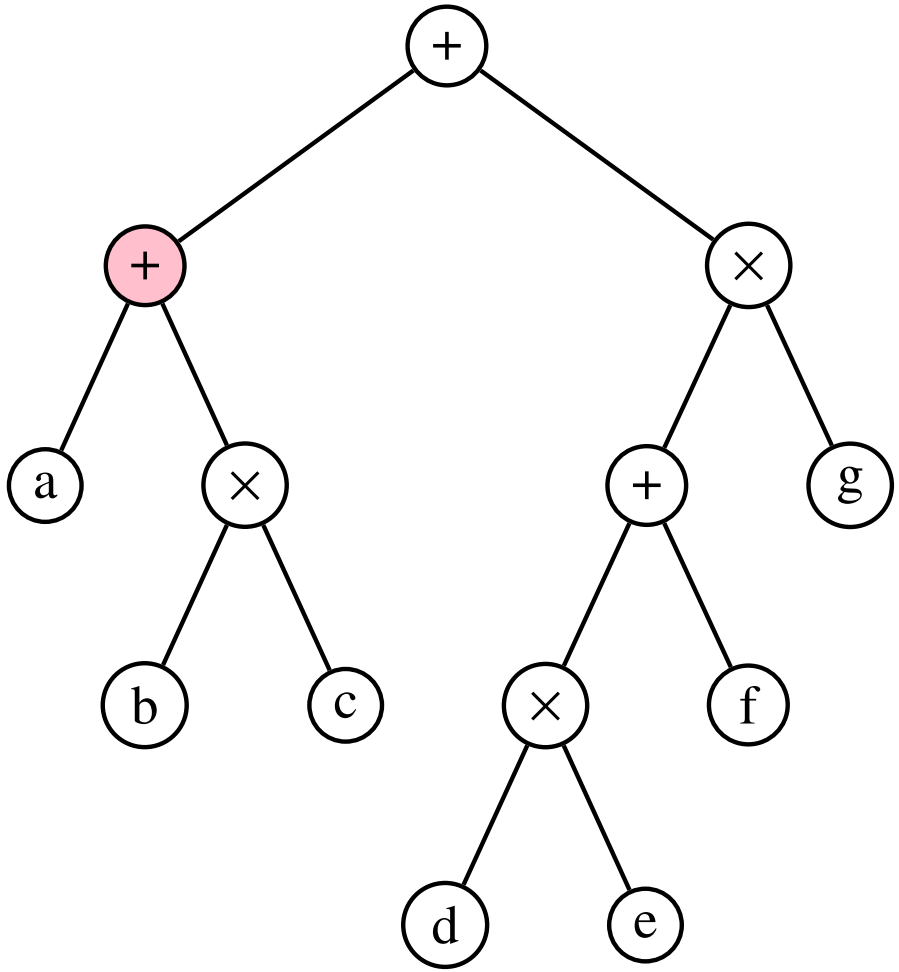


Example: Expression Tree (In-Order Traversal)



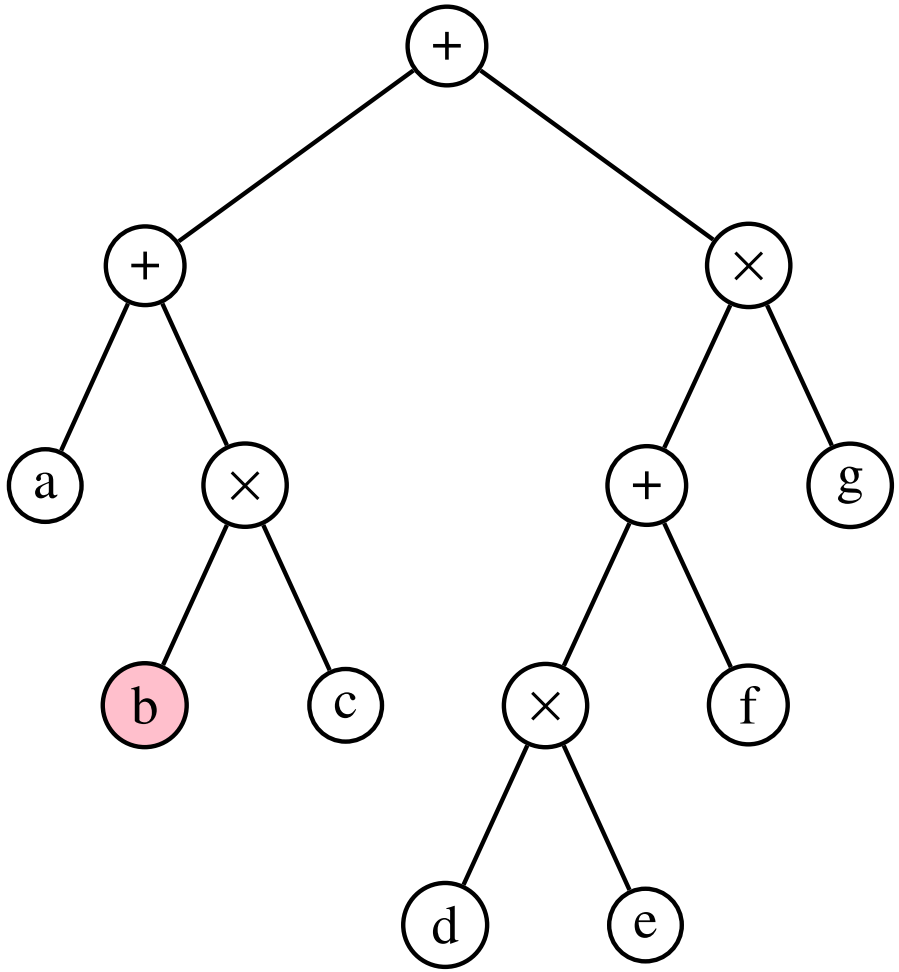
Expression: ((a

Example: Expression Tree (In-Order Traversal)



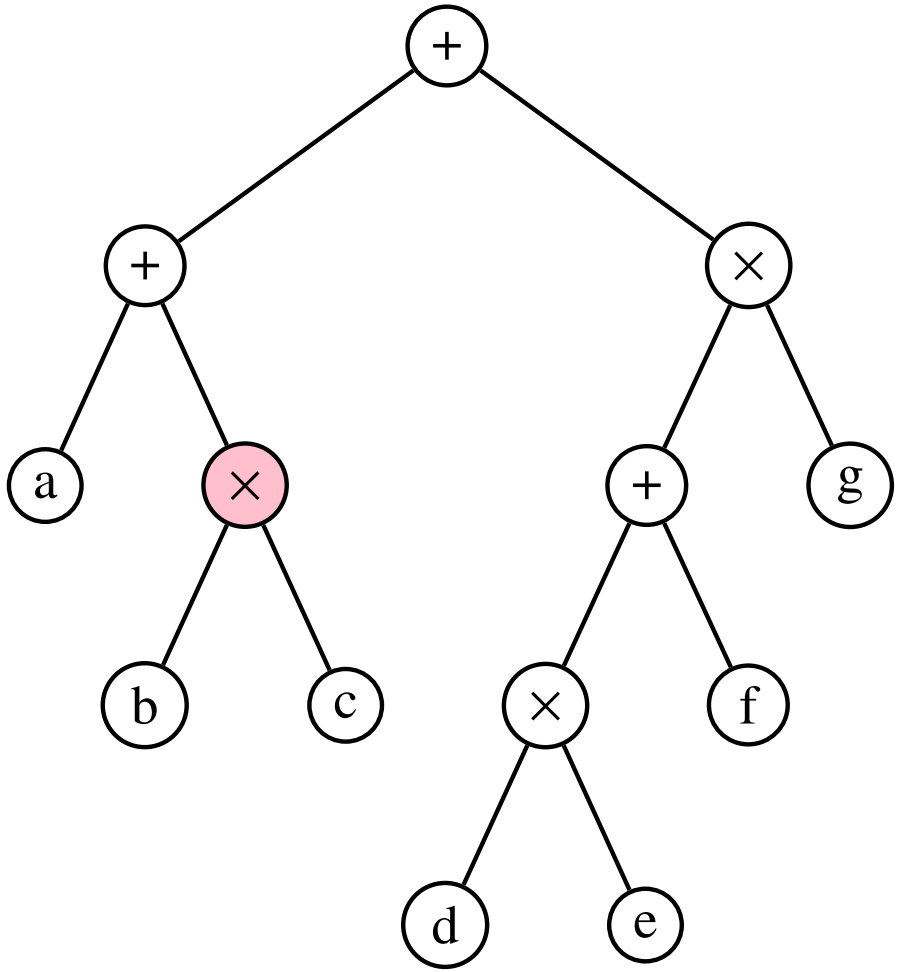
Expression: ((a)+

Example: Expression Tree (In-Order Traversal)



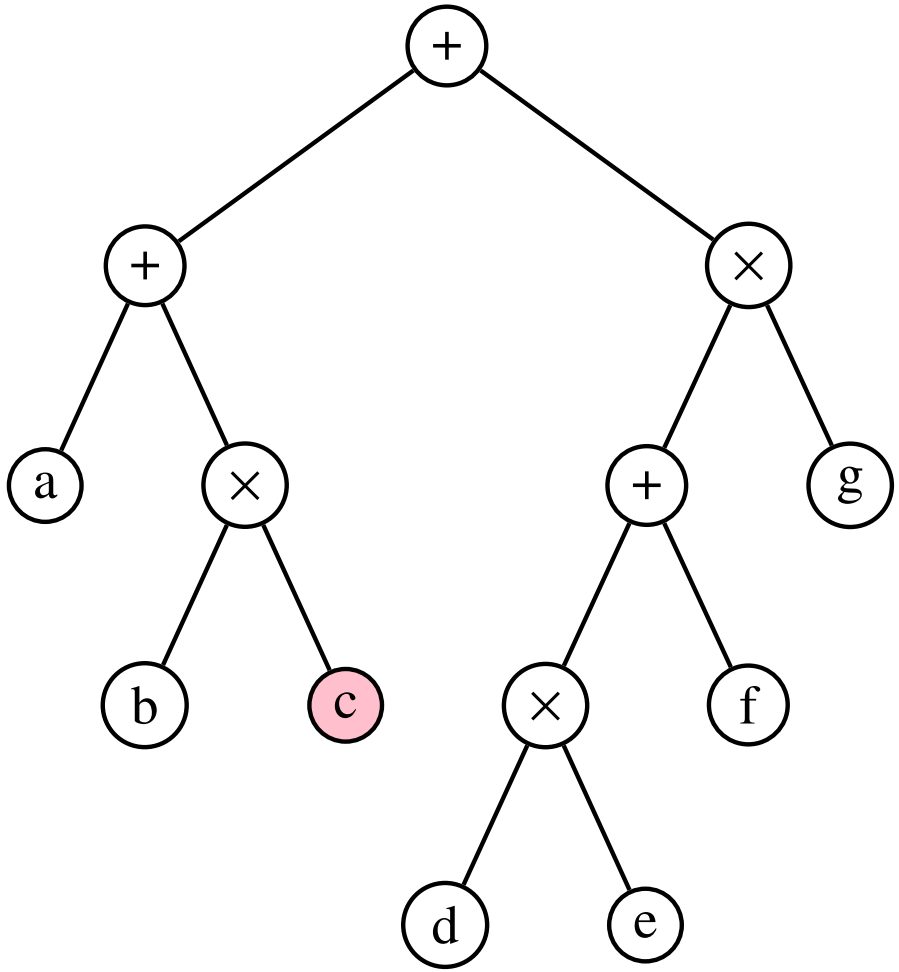
Expression: ((a)+((b

Example: Expression Tree (In-Order Traversal)



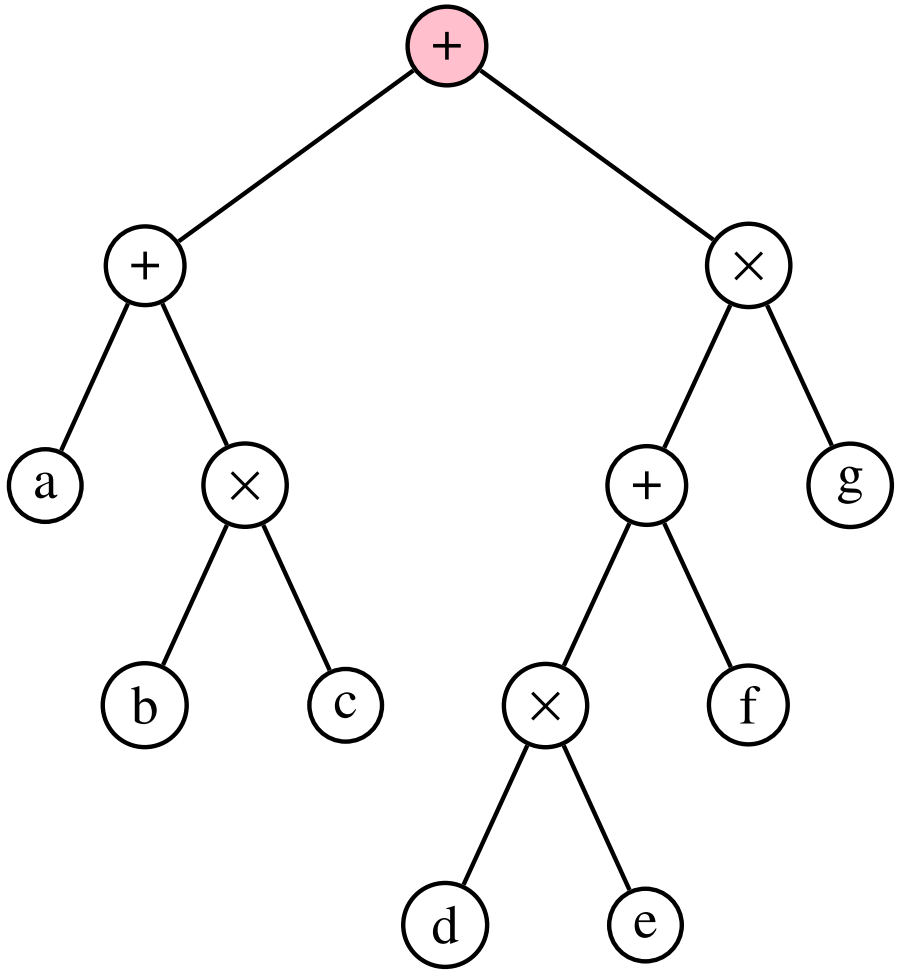
Expression: ((a)+((b)×

Example: Expression Tree (In-Order Traversal)



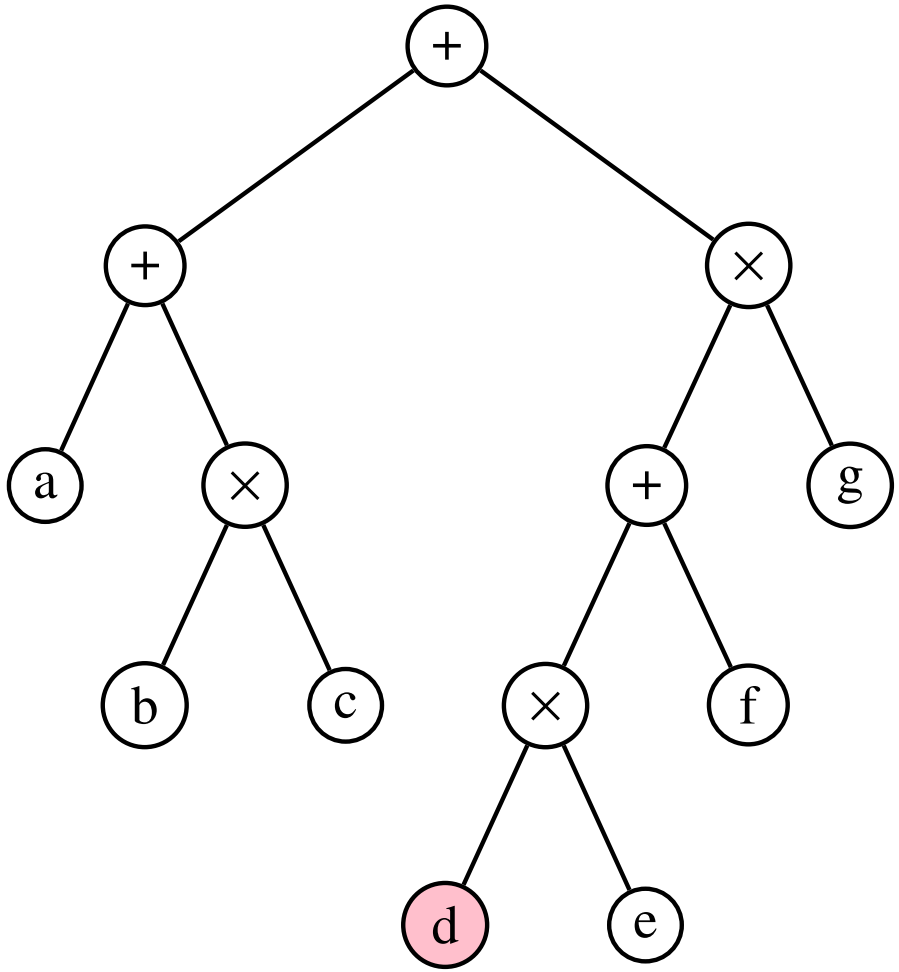
Expression: ((a)+((b)×(c

Example: Expression Tree (In-Order Traversal)



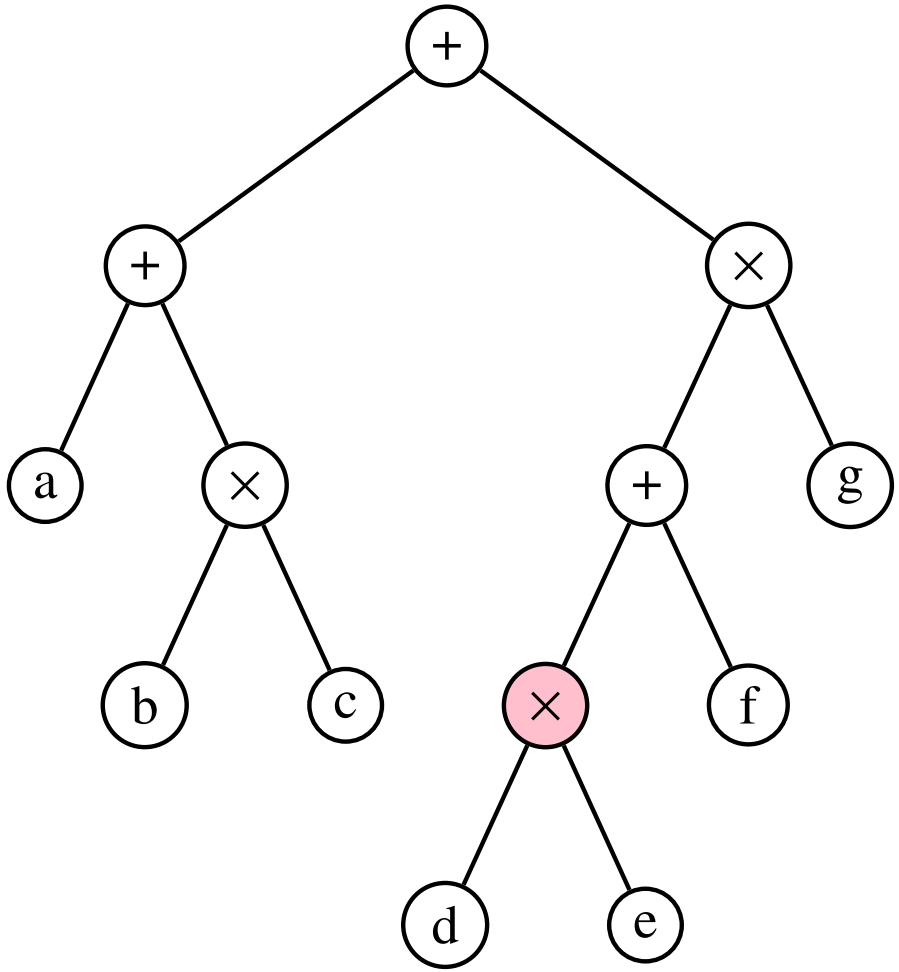
Expression: ((a)+((b)×(c)))+

Example: Expression Tree (In-Order Traversal)



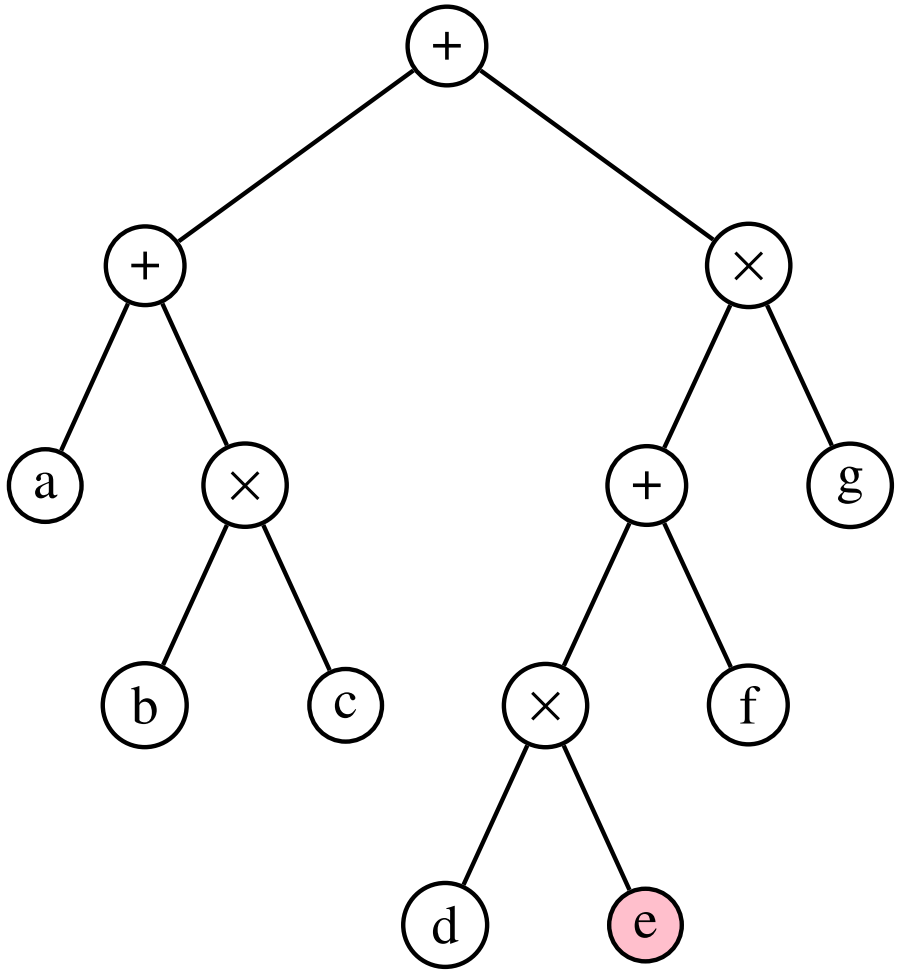
Expression: ((a)+((b)×(c)))+((((d

Example: Expression Tree (In-Order Traversal)



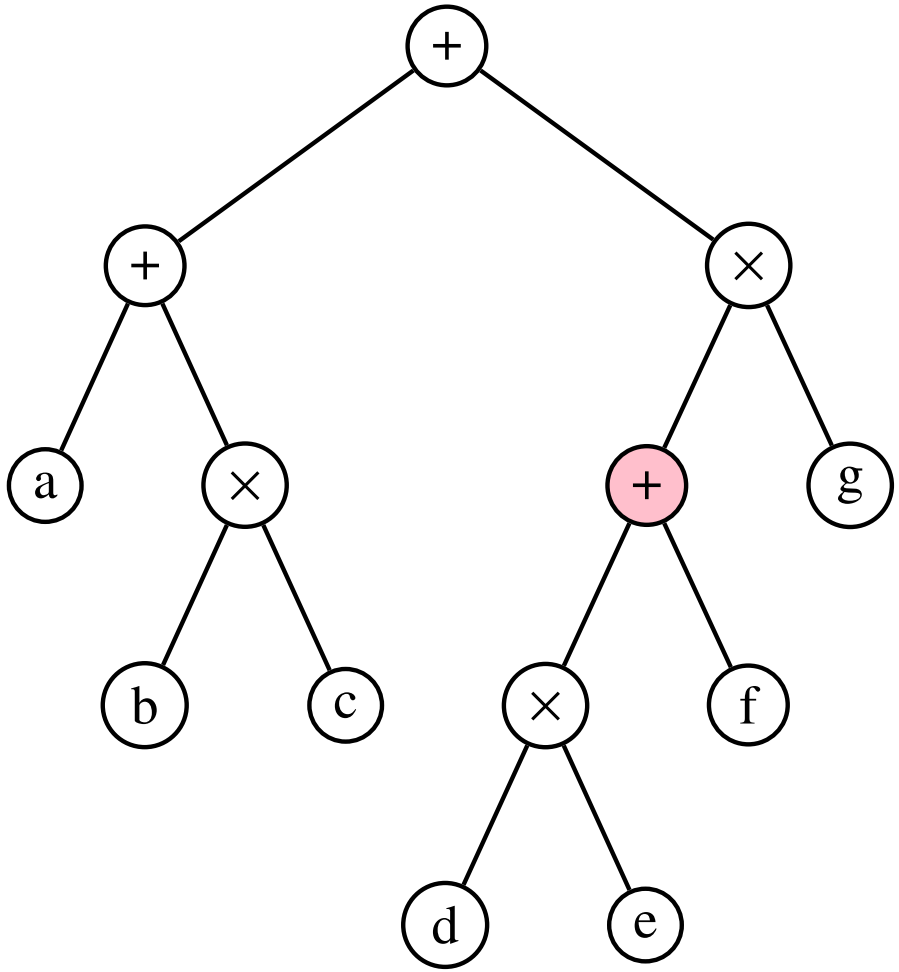
Expression: ((a)+((b)×(c)))+((((d)×

Example: Expression Tree (In-Order Traversal)



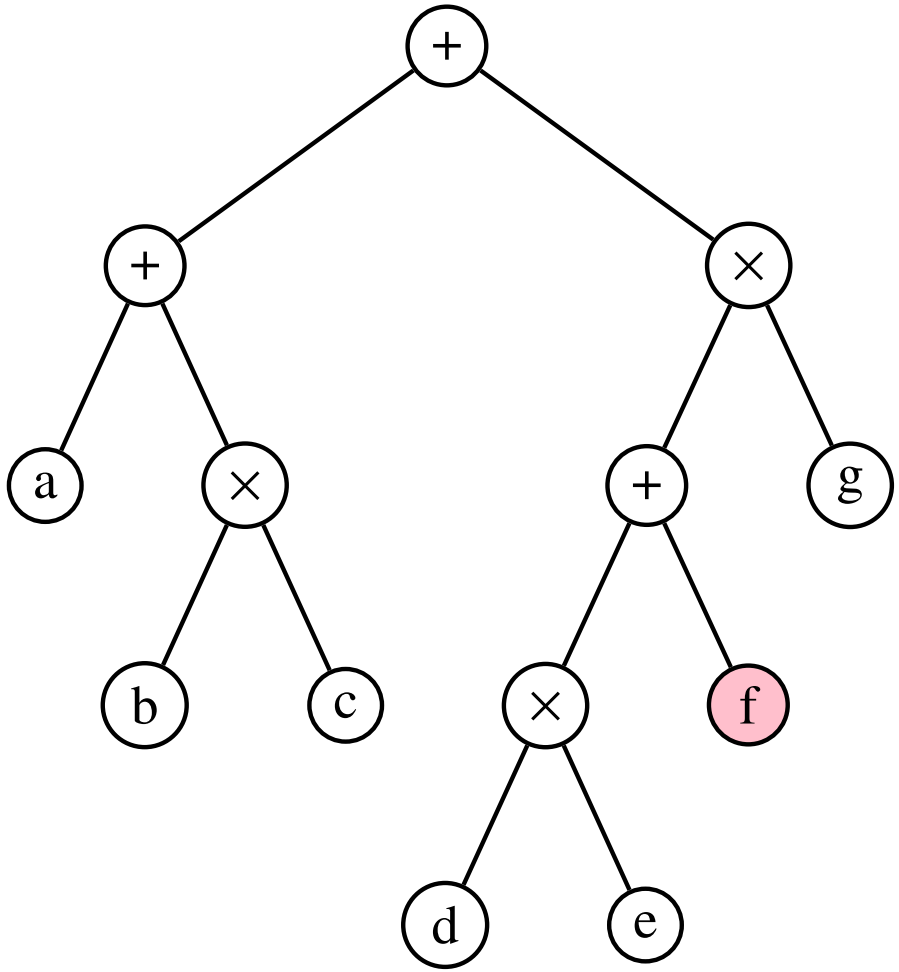
Expression: ((a)+((b)×(c)))+((((d)×(e

Example: Expression Tree (In-Order Traversal)



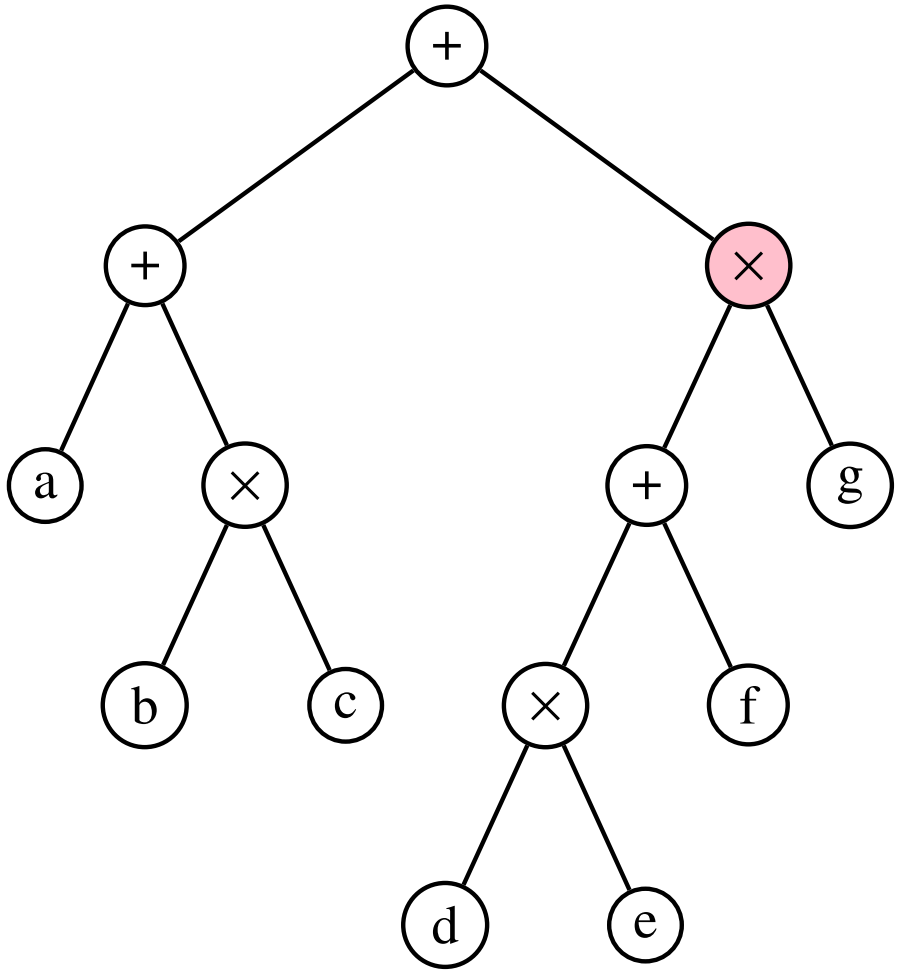
Expression: ((a)+((b)×(c)))+((((d)×(e))+

Example: Expression Tree (In-Order Traversal)



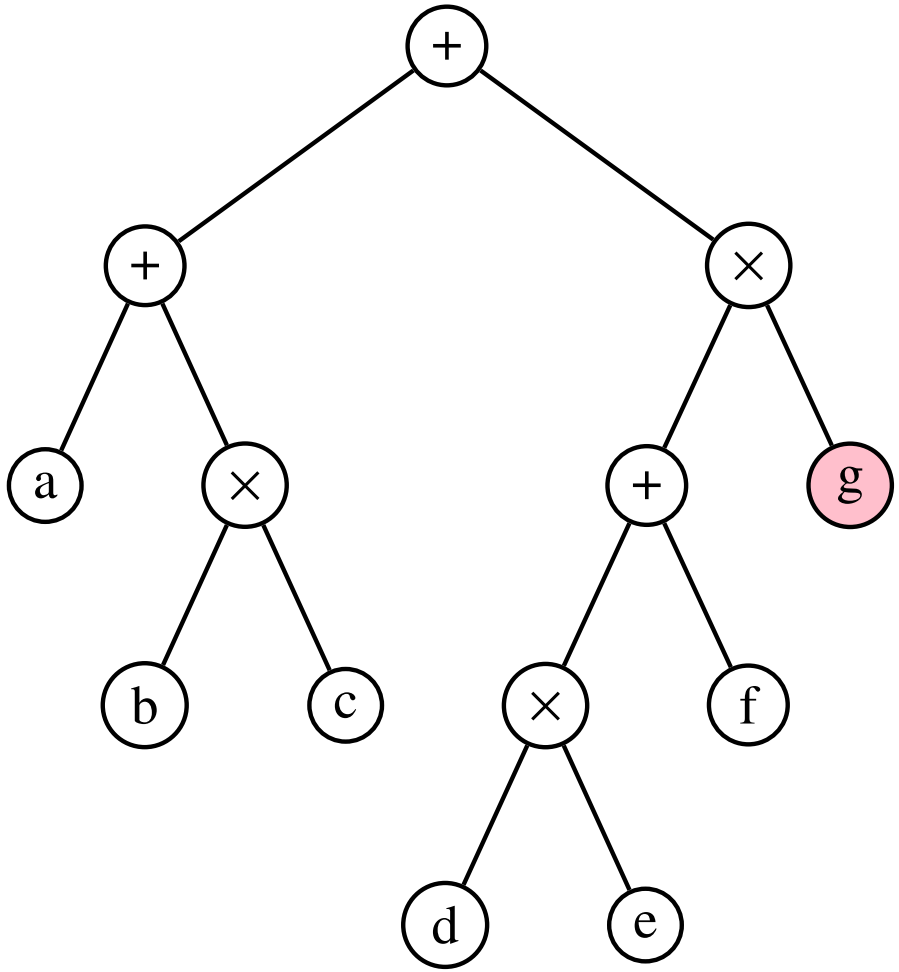
Expression: ((a)+((b)×(c)))+((((d)×(e))+(f

Example: Expression Tree (In-Order Traversal)



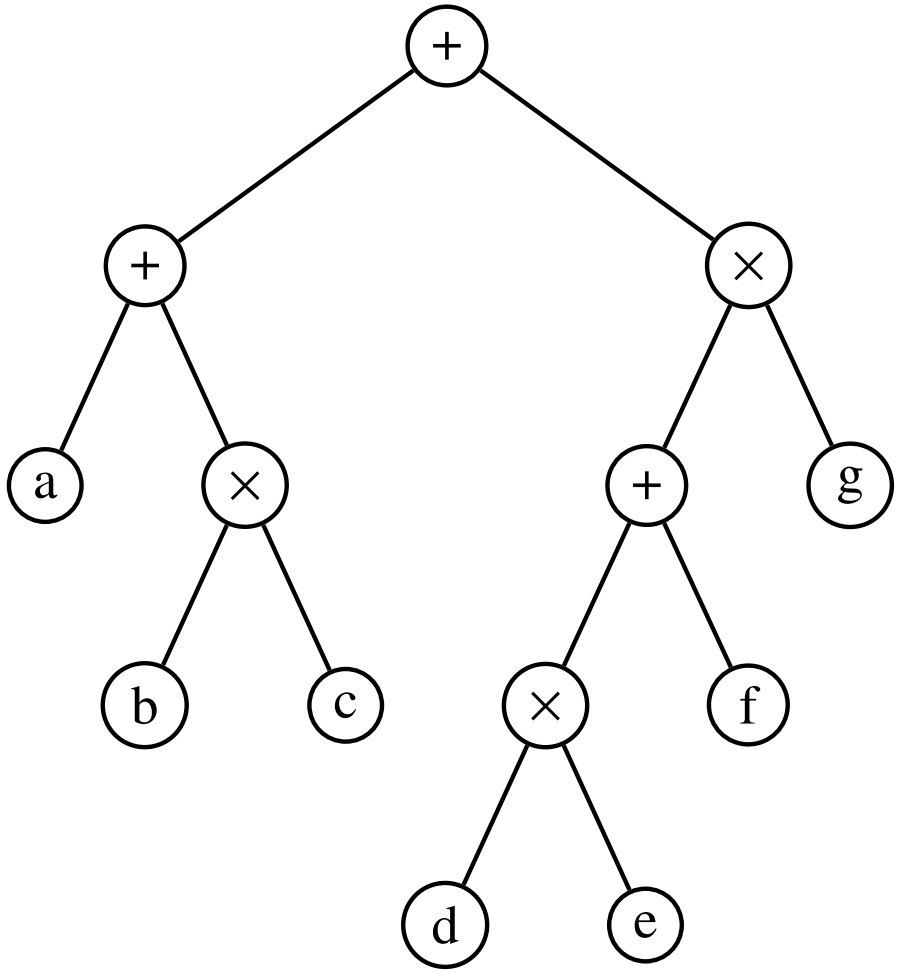
Expression: ((a)+((b)×(c)))+((((d)×(e))+(f))×

Example: Expression Tree (In-Order Traversal)



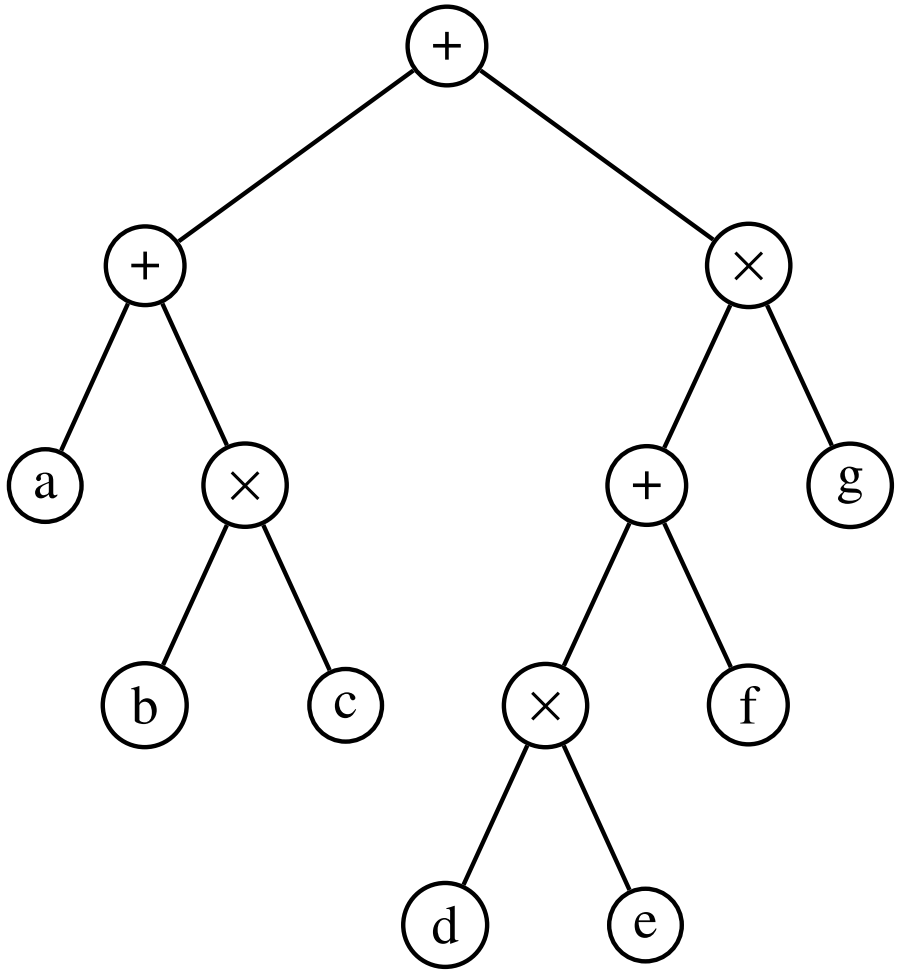
Expression: ((a)+((b)×(c)))+((((d)×(e))+(f))×(g

Example: Expression Tree (In-Order Traversal)



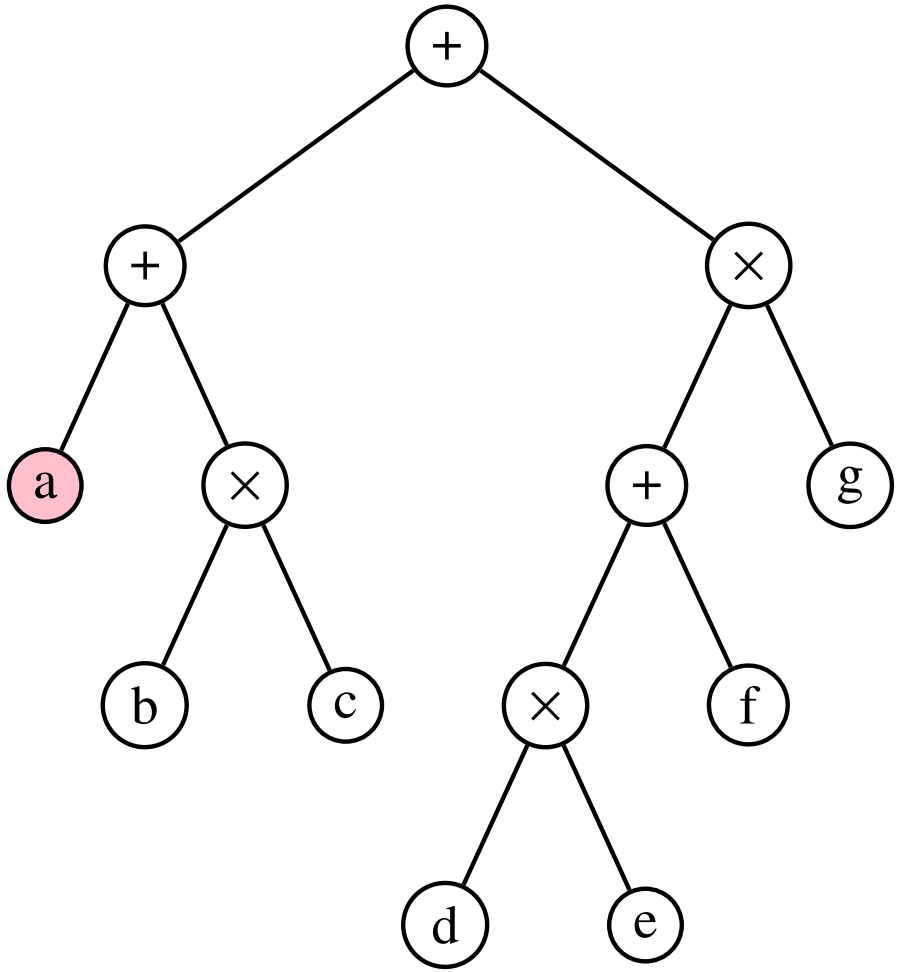
Expression: ((a)+((b)×(c)))+((((d)×(e))+(f))×(g)) ⟶ **Infix Notation**

Example: Expression Tree (Post-Order Traversal) left, right, node



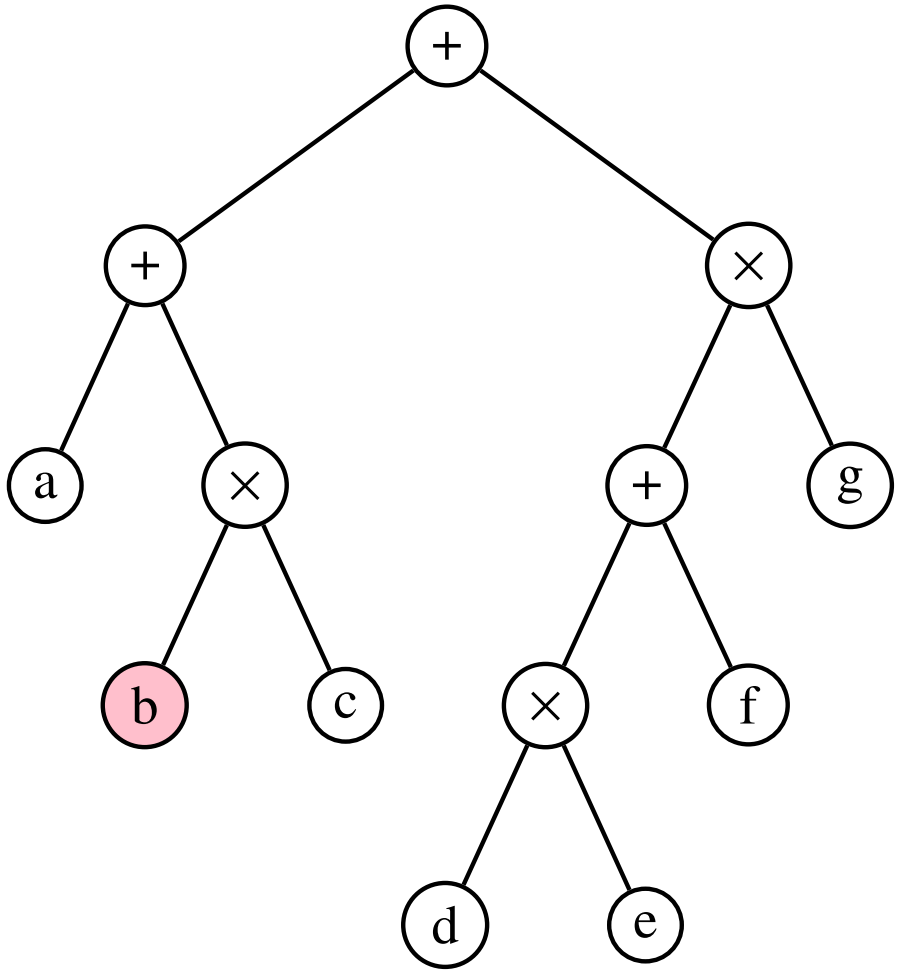
Expression:

Example: Expression Tree (Post-Order Traversal)



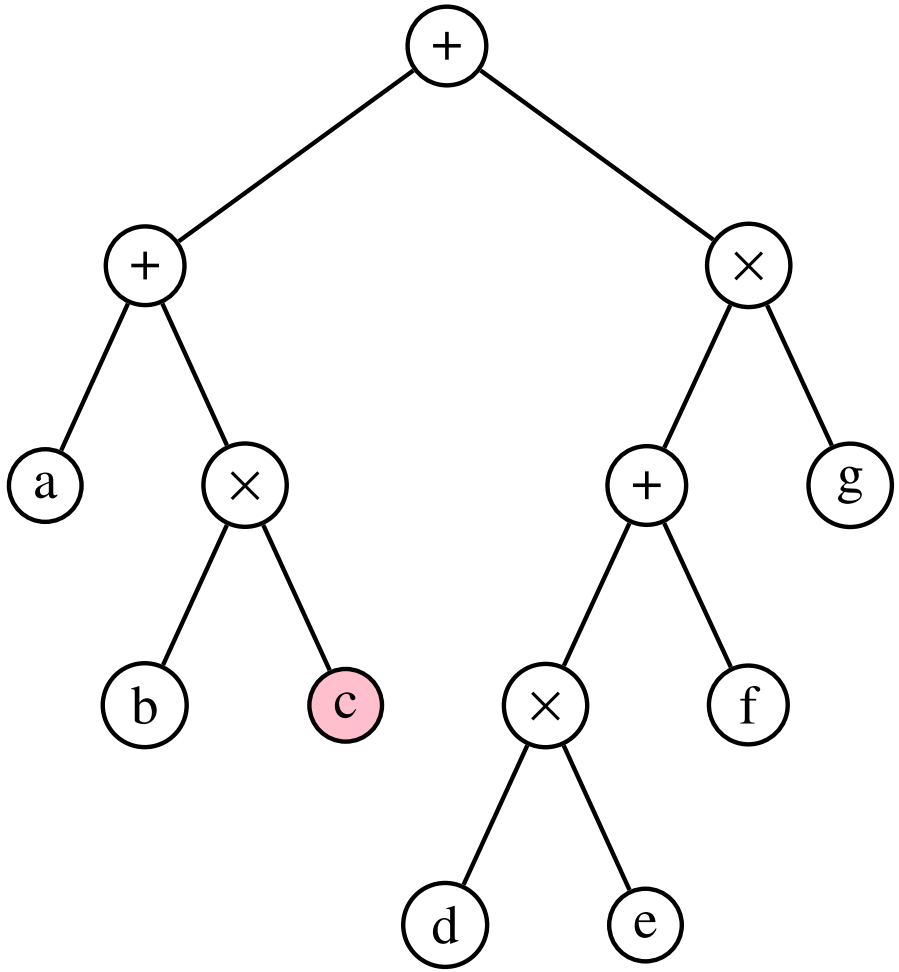
Expression: a

Example: Expression Tree (Post-Order Traversal)



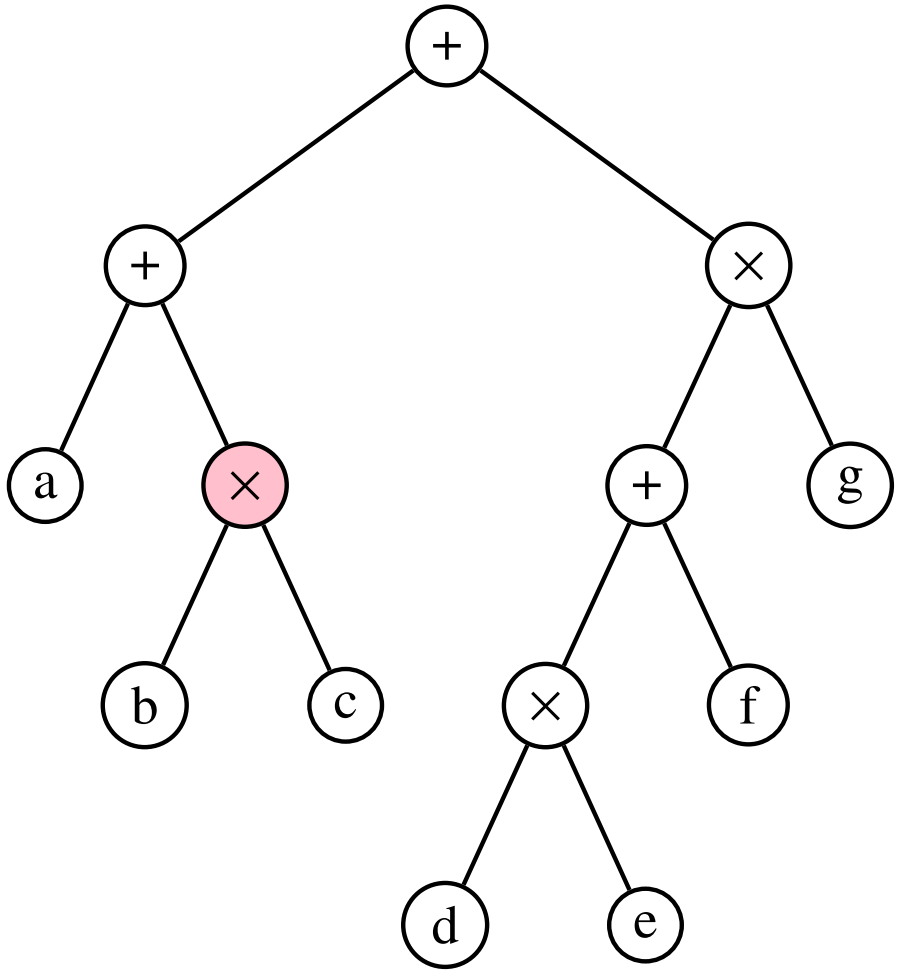
Expression: ab

Example: Expression Tree (Post-Order Traversal)



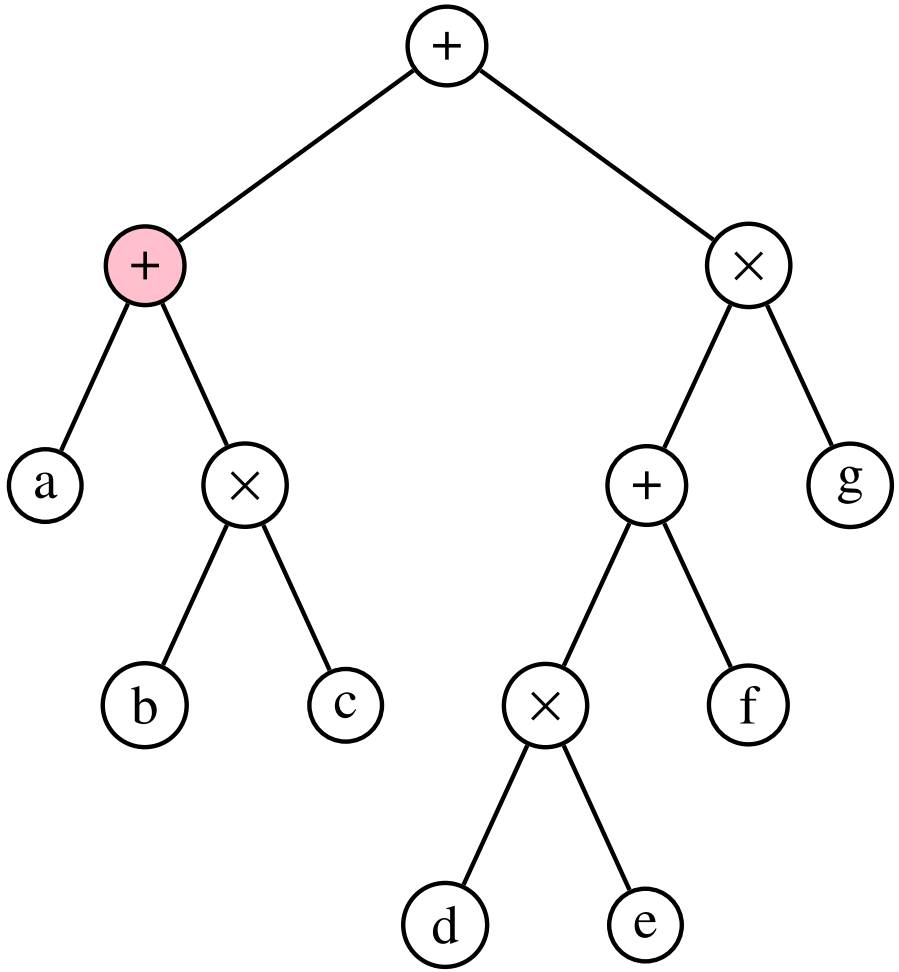
Expression: abc

Example: Expression Tree (Post-Order Traversal)



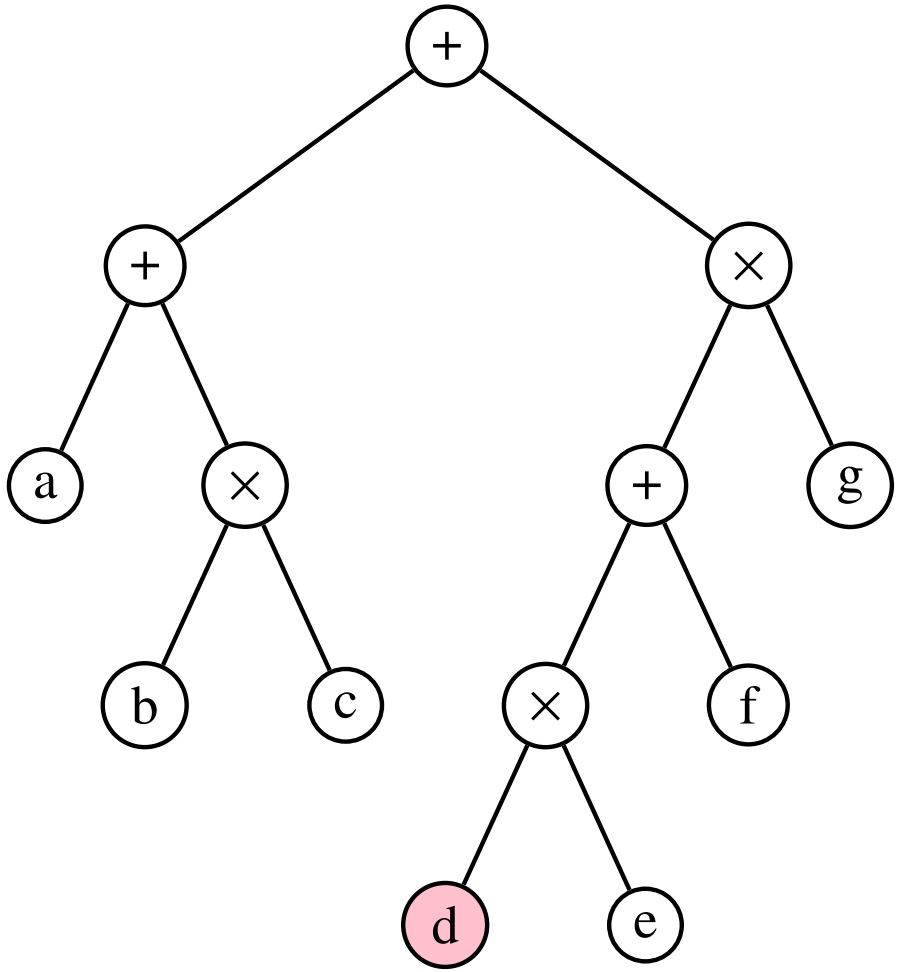
Expression: abc×

Example: Expression Tree (Post-Order Traversal)



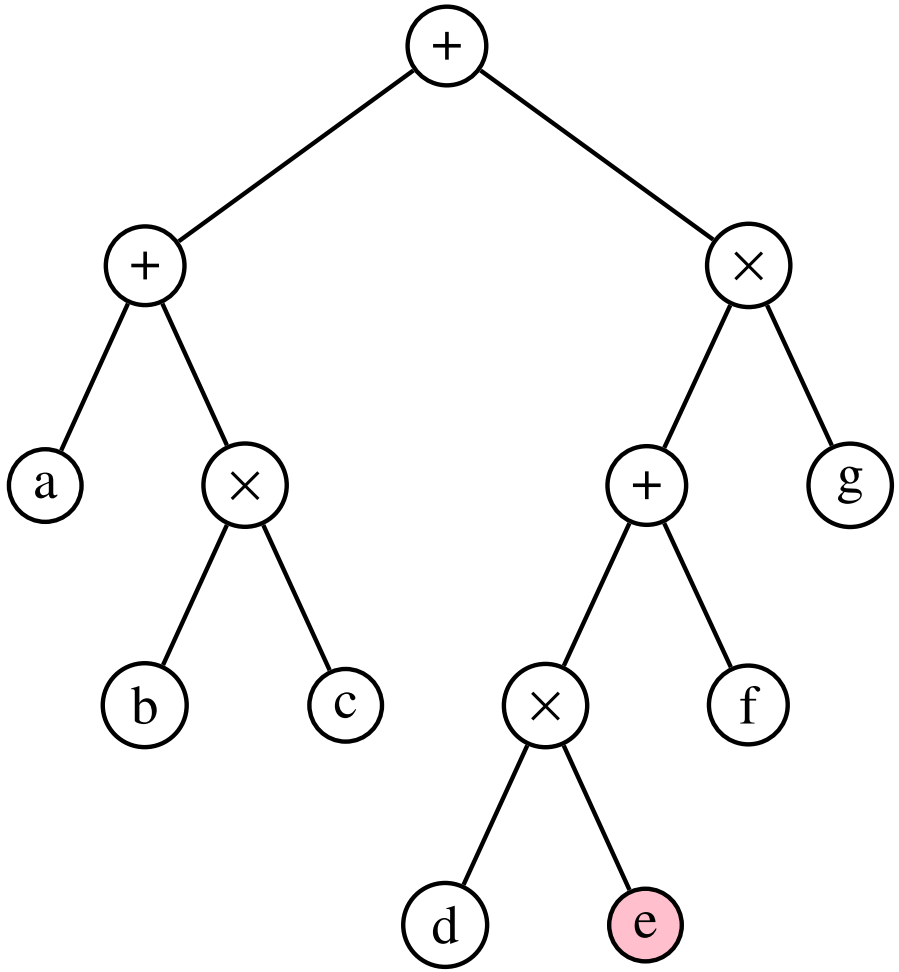
Expression: abc×+

Example: Expression Tree (Post-Order Traversal)



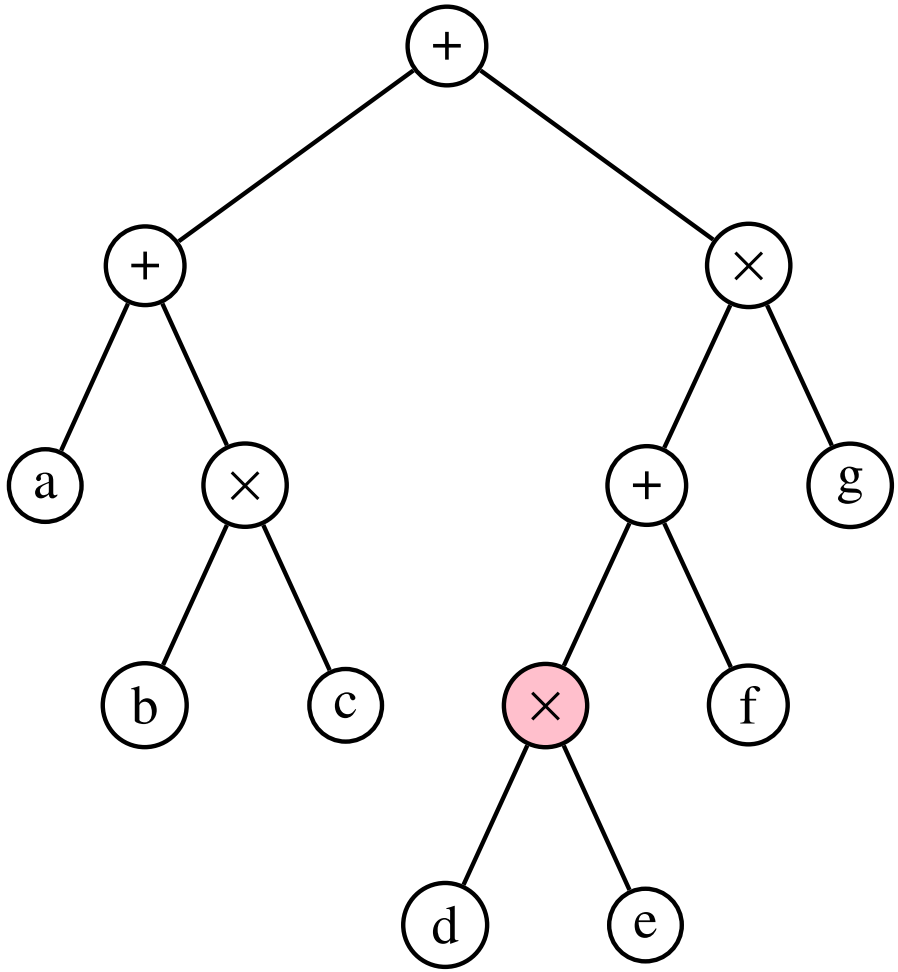
Expression: abc×+d

Example: Expression Tree (Post-Order Traversal)



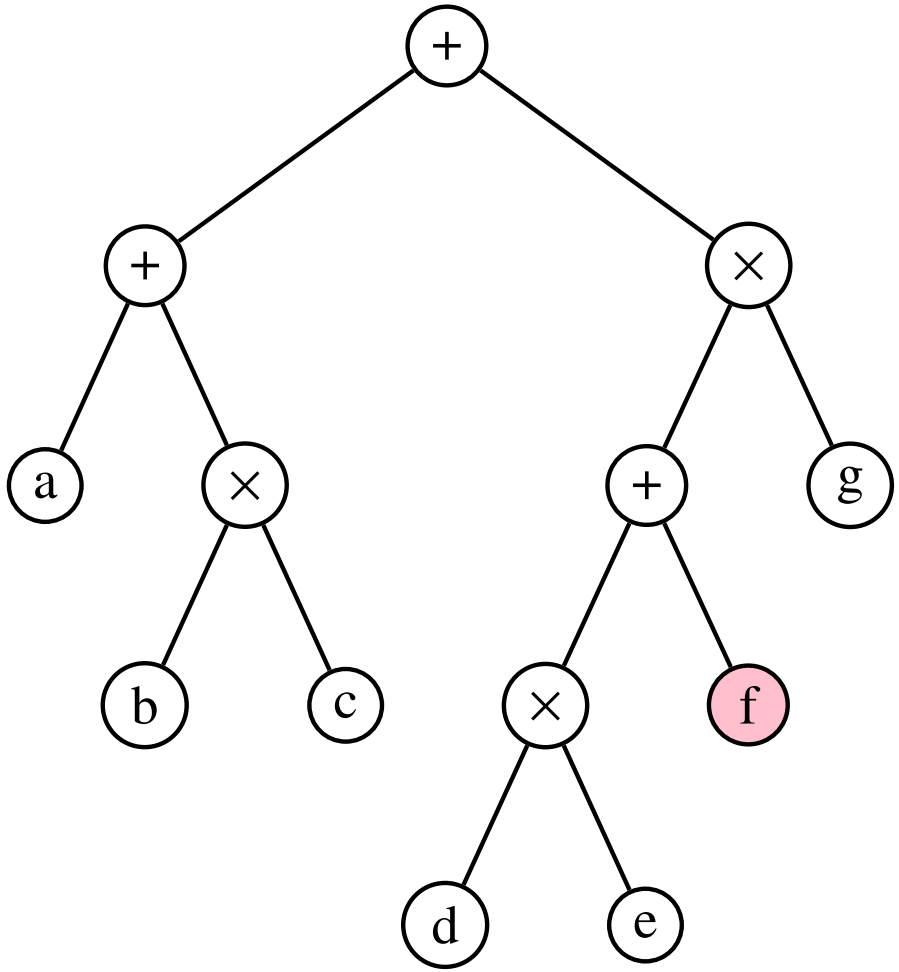
Expression: abc×+de

Example: Expression Tree (Post-Order Traversal)



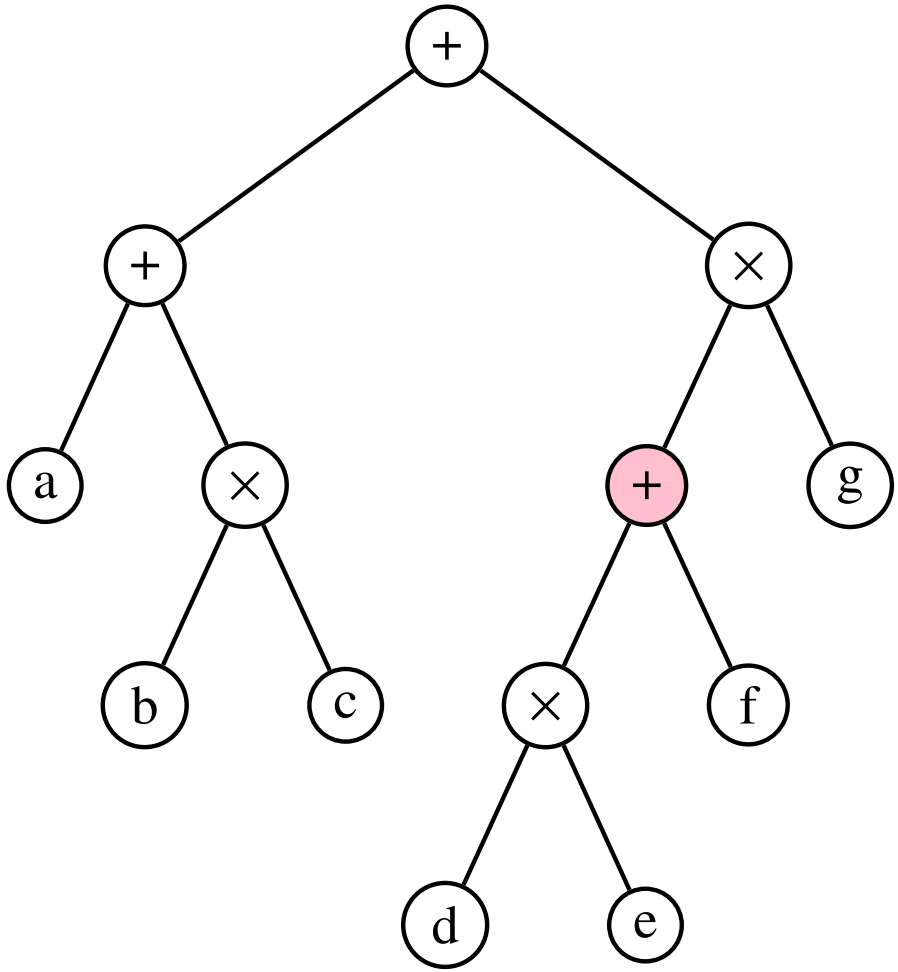
Expression: abc×+de×

Example: Expression Tree (Post-Order Traversal)



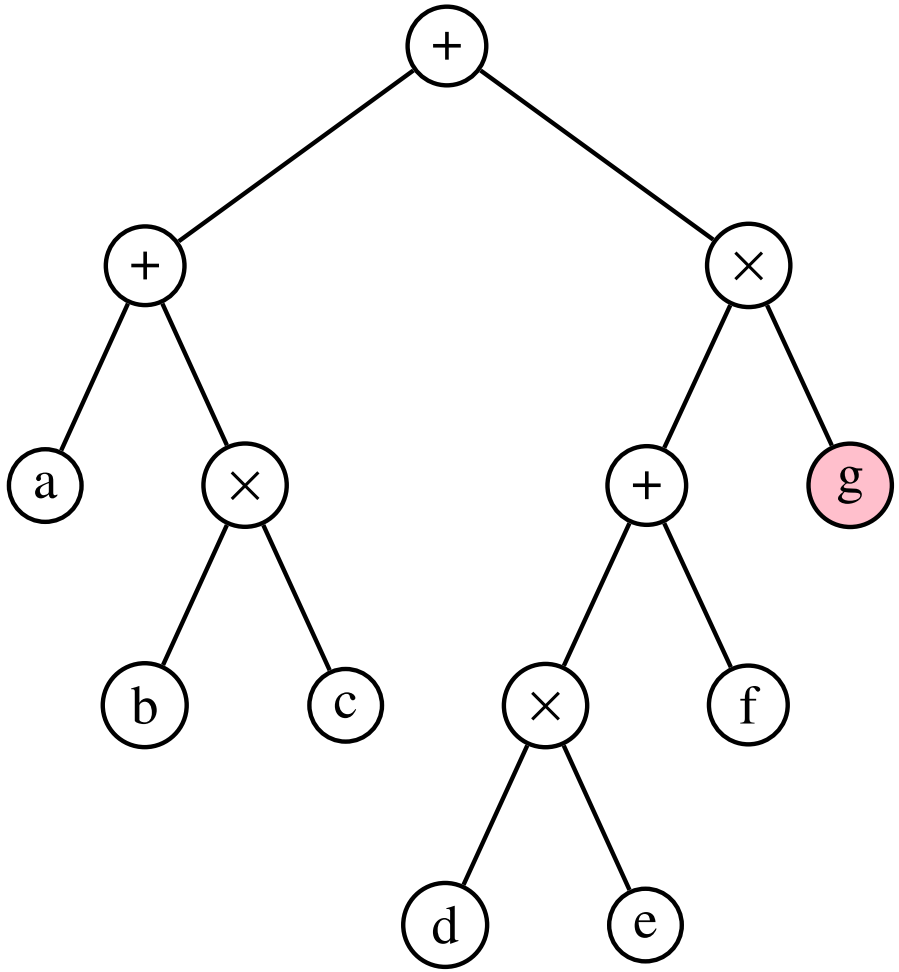
Expression: abc×+de×f

Example: Expression Tree (Post-Order Traversal)



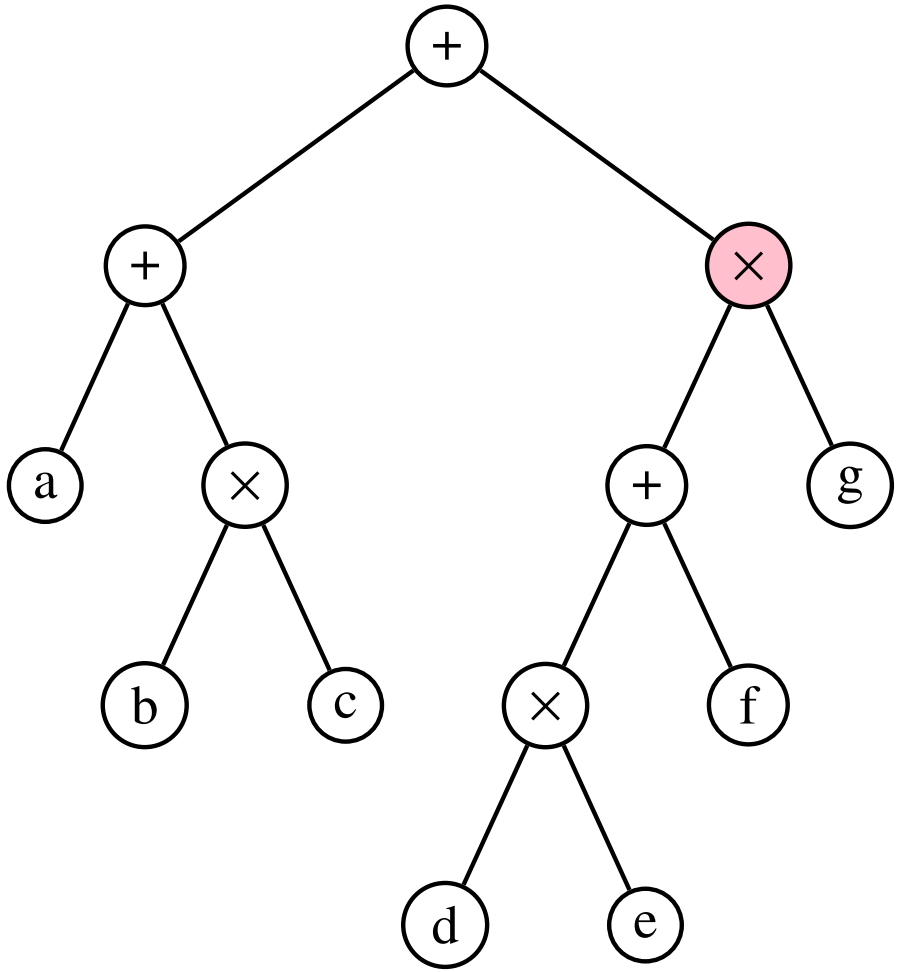
Expression: abc×+de×f+

Example: Expression Tree (Post-Order Traversal)



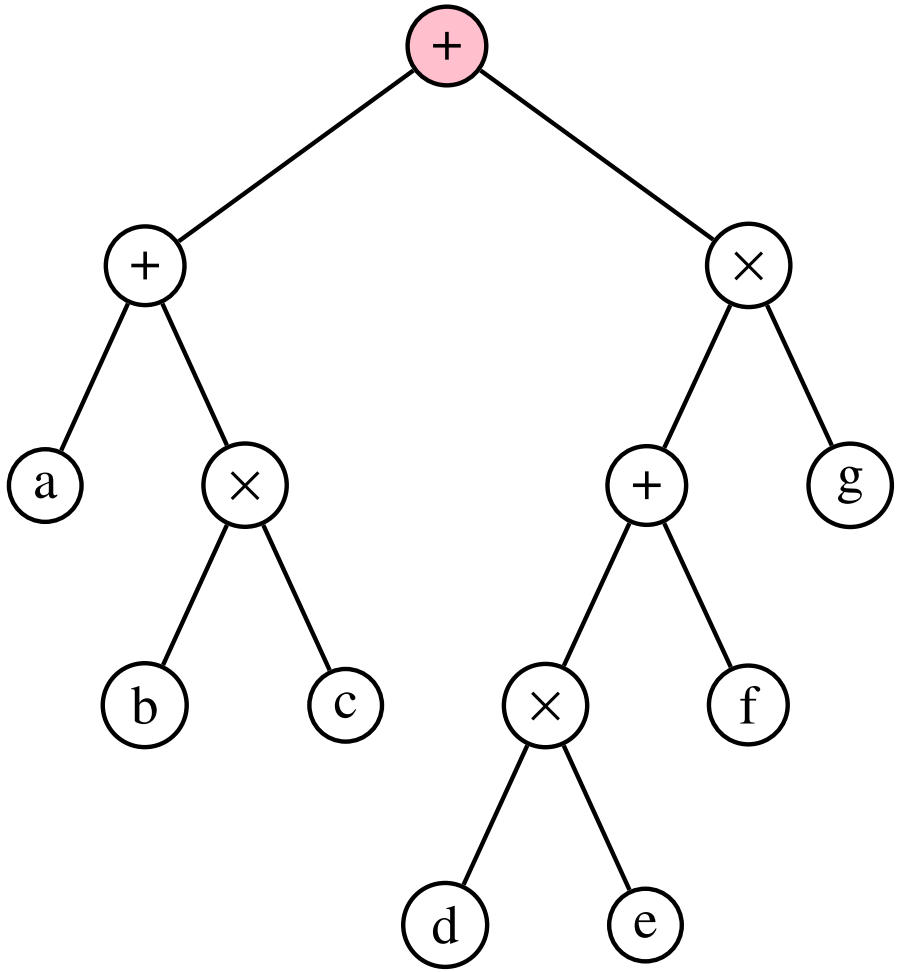
Expression: abc×+de×f+g

Example: Expression Tree (Post-Order Traversal)



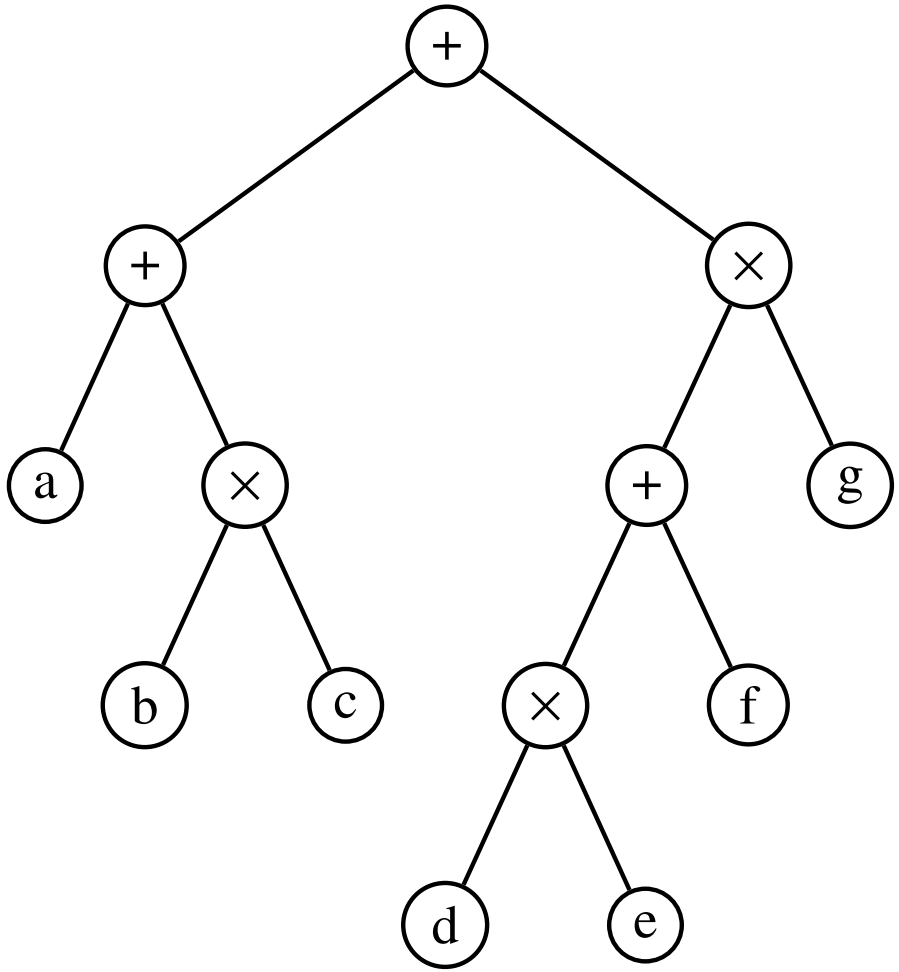
Expression: abc×+de×f+g×

Example: Expression Tree (Post-Order Traversal)



Expression: abc×+de×f+g×+

Example: Expression Tree (Post-Order Traversal)



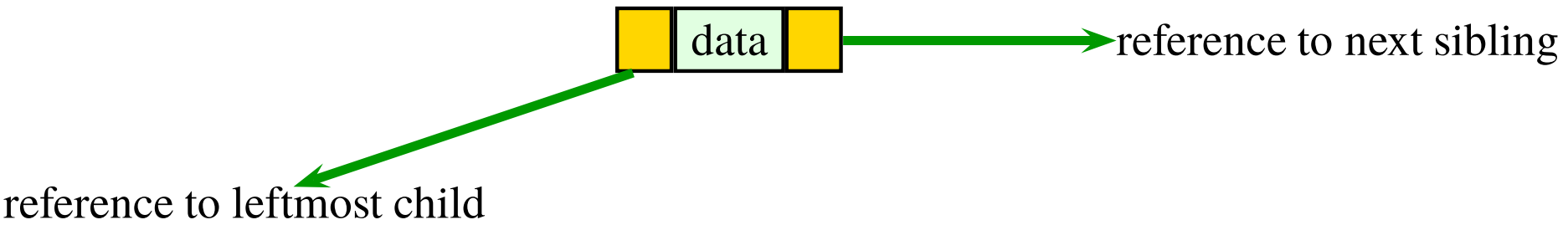
Expression: abc×+de×f+g×+ ⟶ **Postfix Notation**

Representing Non-Binary Trees

Question: How can we represent non-binary trees?

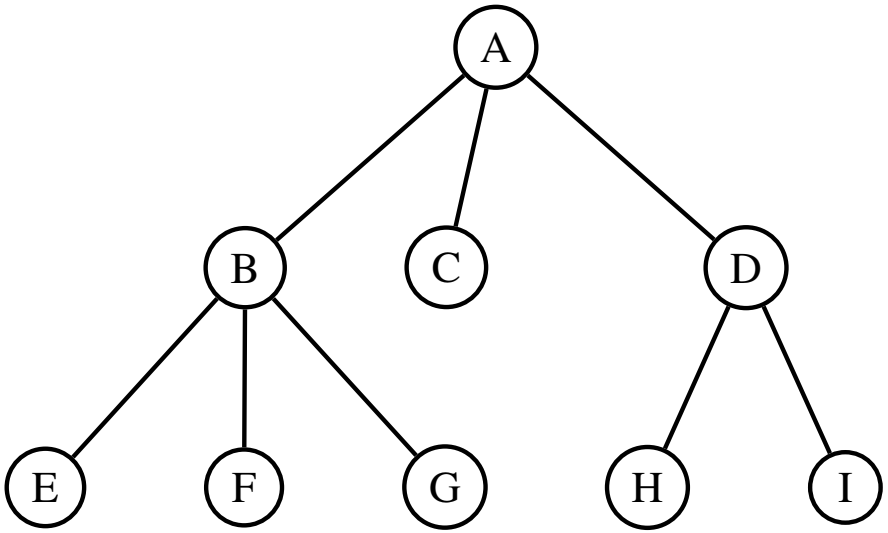
Representing Non-Binary Trees

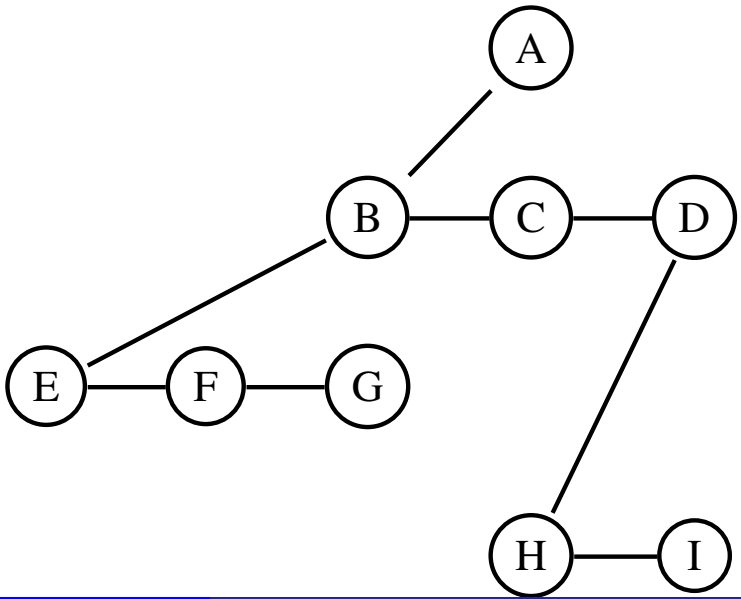
Question: How can we represent non-binary trees?
Answer: You can design a tree node with more than one reference in order to have more than two children (number of references is max number of children).
OR
You can represent a general tree using a binary tree.



Representing Non-Binary Trees

Example: Represent the following tree using a binary tree





Next Topic ...

In this topic we have discussed some of the basic properties of trees and in particular binary trees. In the next topic we will discuss searching algorithms. Binary trees are used to implement a very efficient searching algorithm.

- Binary Search in an array
- Binary Search Trees
- Hashing