

Data Structures in Java

Lecture 9: Trees and Tree Traversals.

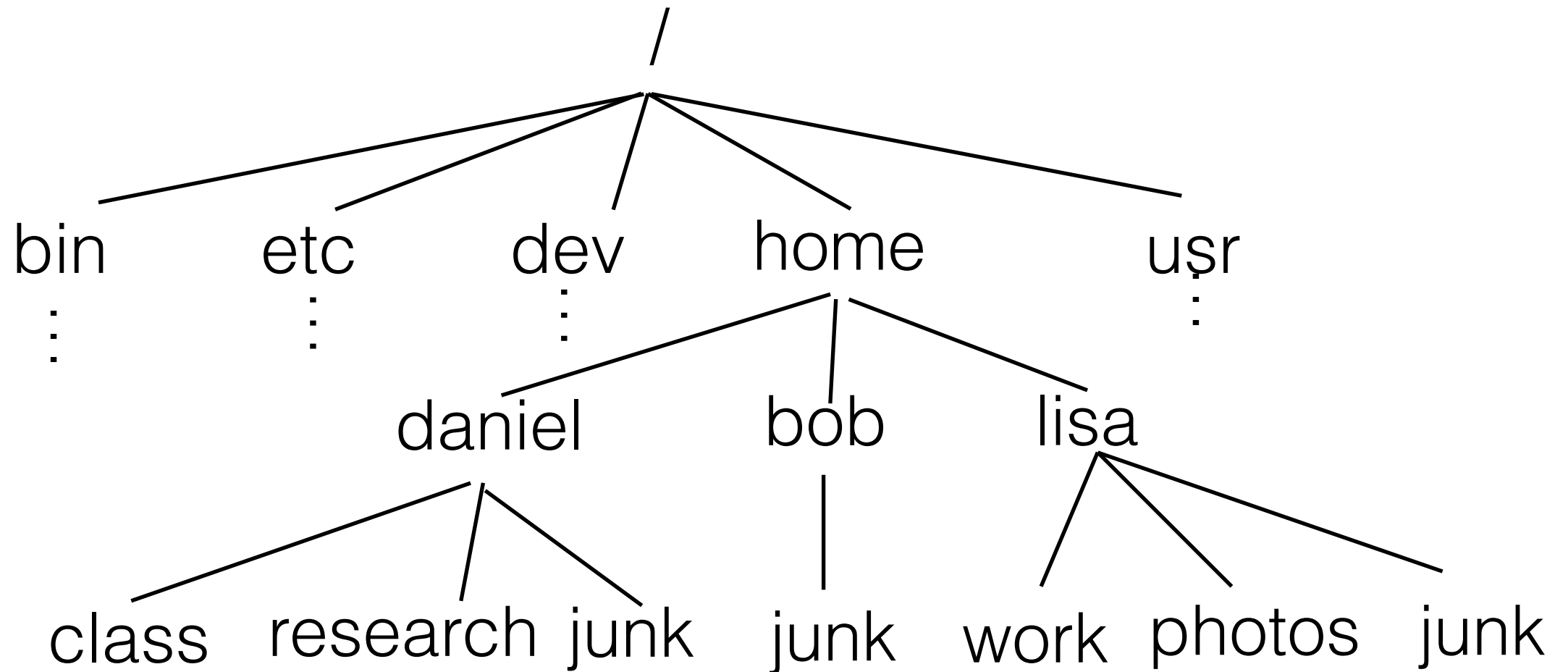
2/16/2022

Daniel Bauer

Trees in Computer Science

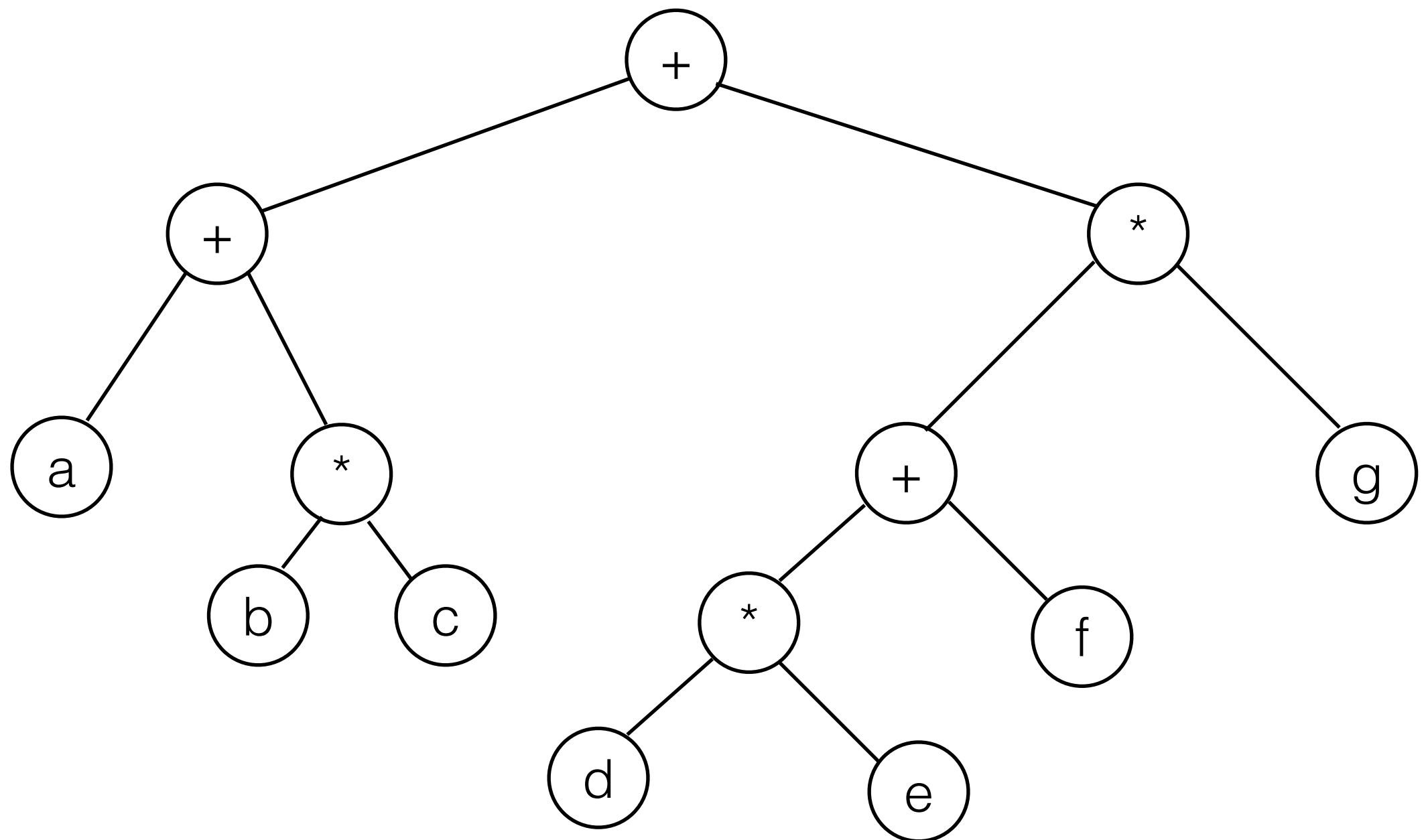
- A lot of data comes in a *hierarchical/nested structure*.
 - Mathematical expressions.
 - Program structure.
 - File systems.
 - Decision trees.
 - Natural Language Syntax, Taxonomies, Family Trees, ...

Example: Unix File System



Example: Expression Trees

$$(a + b * c) + (d * e + f) * g$$

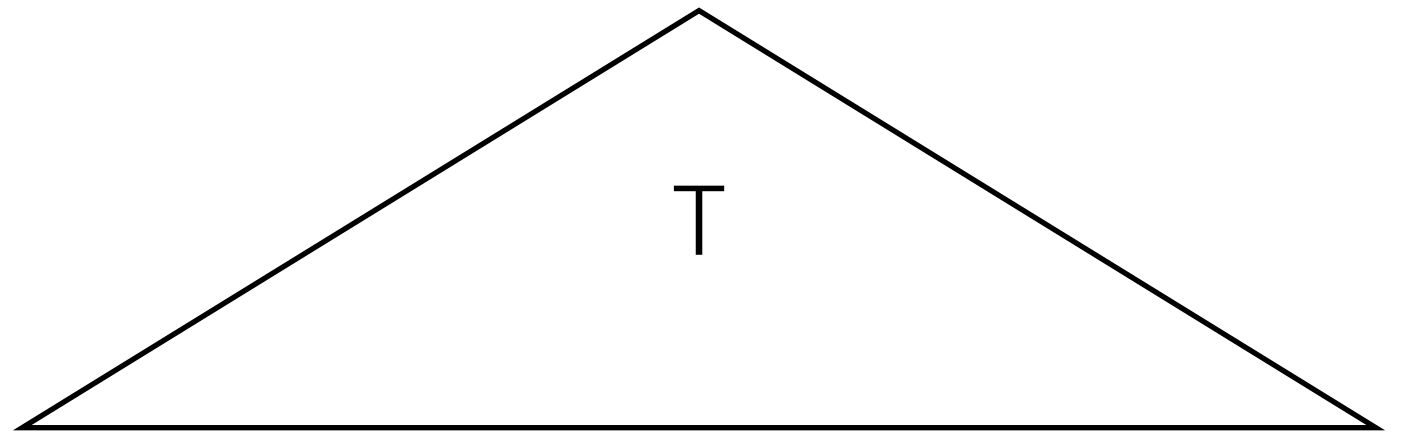


More Efficient Algorithms with Trees

- Sometimes we can represent data in a tree to speed up algorithms.
- Only need to consider part of the tree to solve certain problems:
 - Searching, Sorting,...
- Can often speed up $O(N)$ algorithms to $O(\log N)$ once data is represented as a tree.

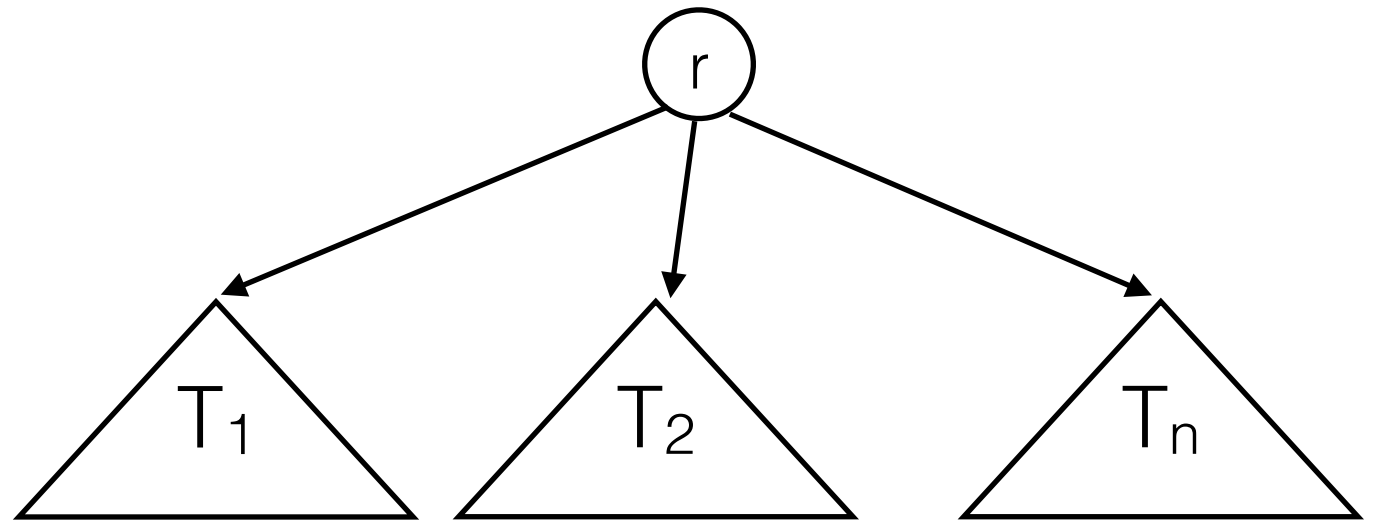
Tree ADT

- A tree T consists of
 - A root node r .
 - zero or more nonempty subtrees $T_1, T_2, \dots T_N$,
 - each connected by a directed edge from r .
- Support typical collection operations: size, get, set, add, remove, find, ...

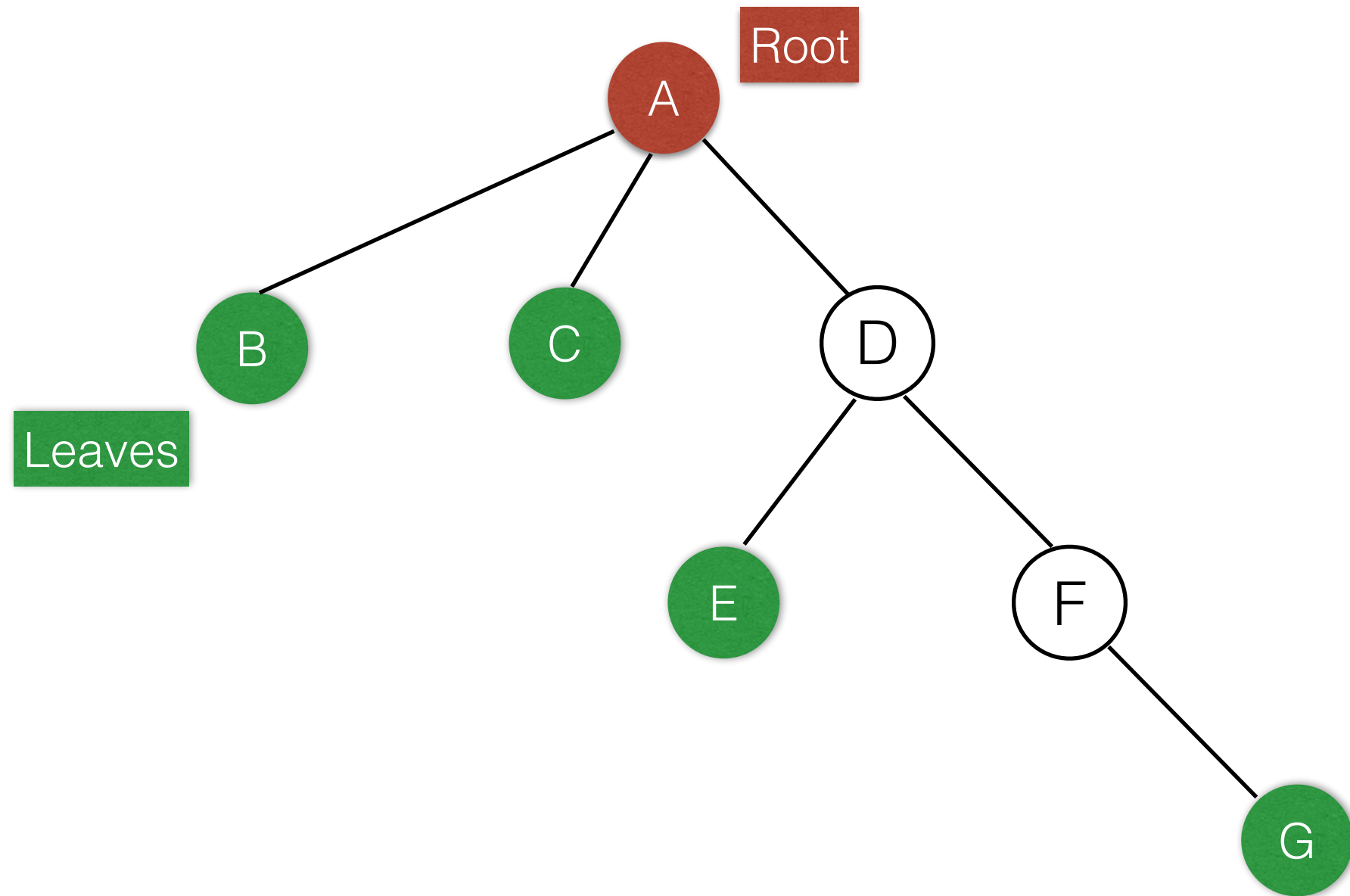


Tree ADT

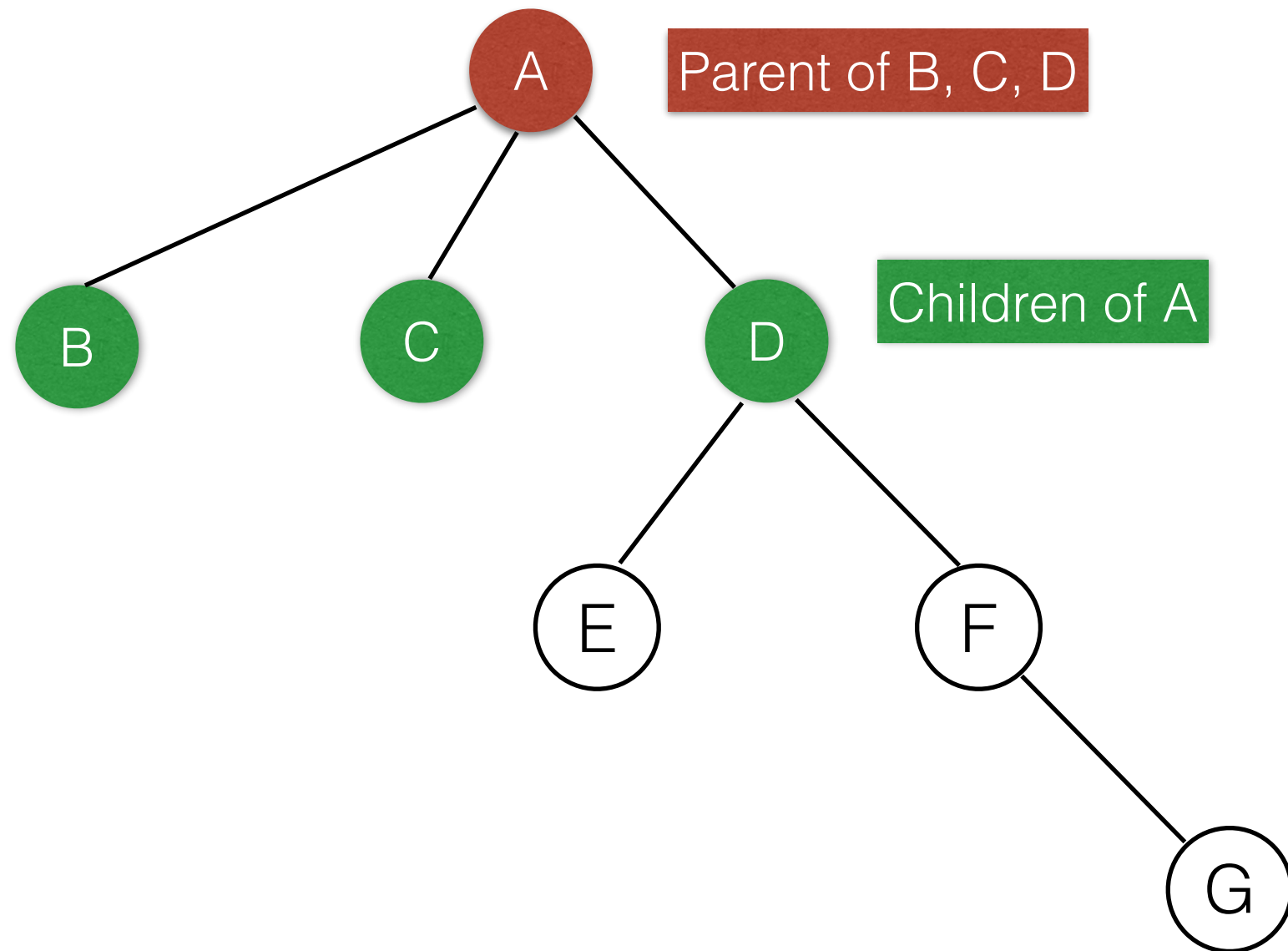
- A tree T consists of
 - A root node r .
 - zero or more nonempty subtrees T_1, T_2, \dots, T_N ,
 - each connected by a directed edge from r .
- Support typical collection operations: size, get, set, add, remove, find, ...



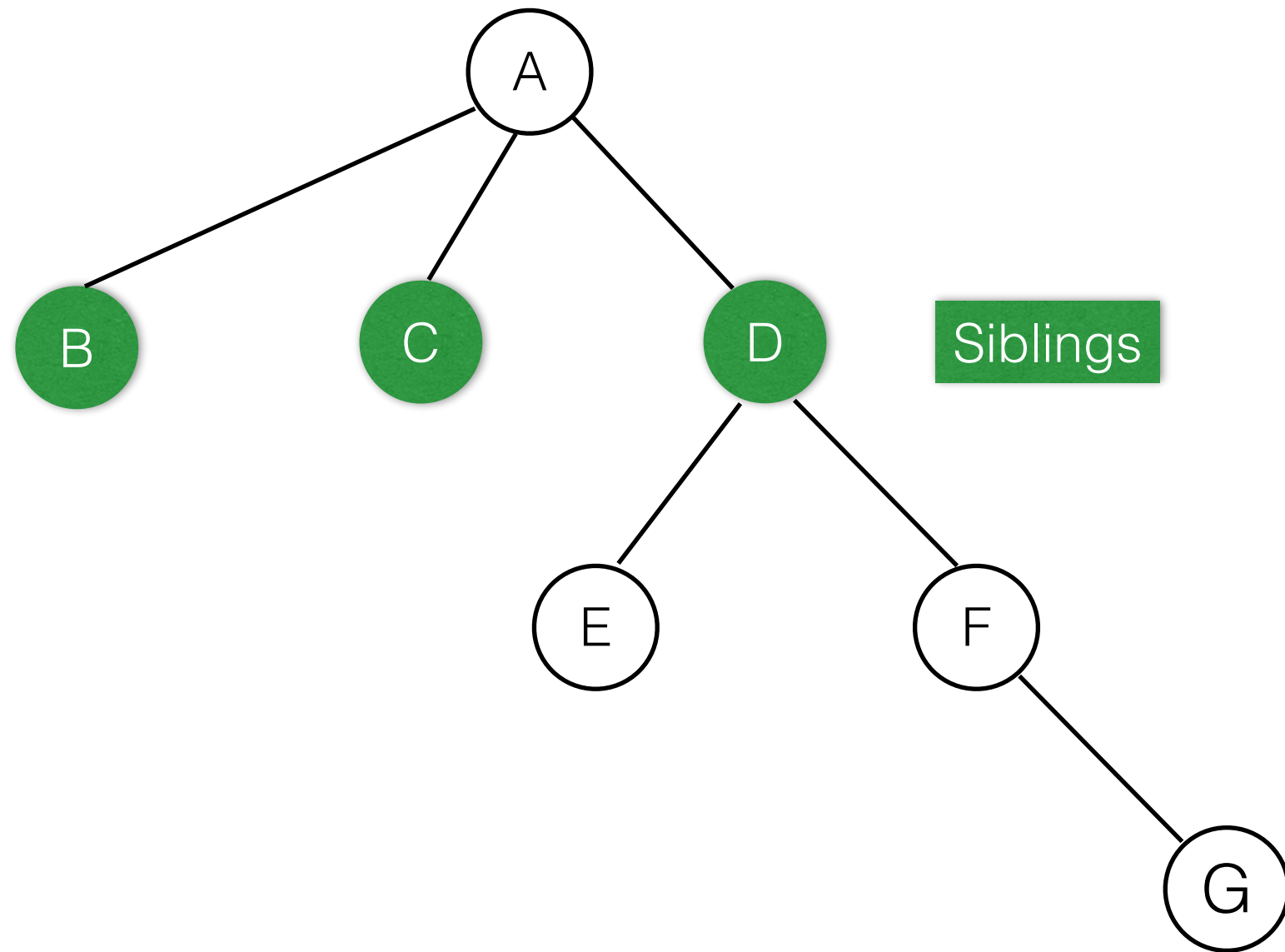
Tree Terminology



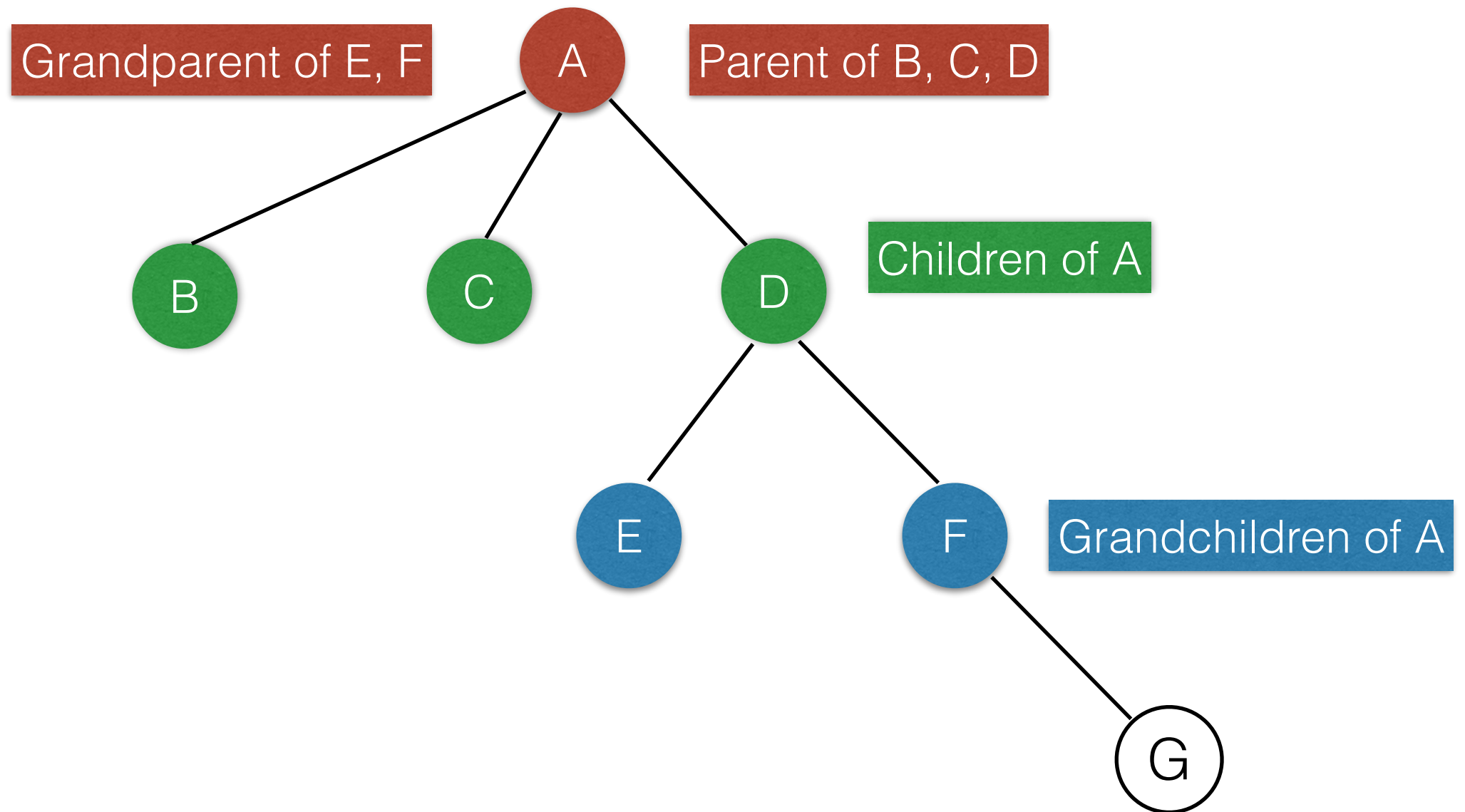
Tree Terminology



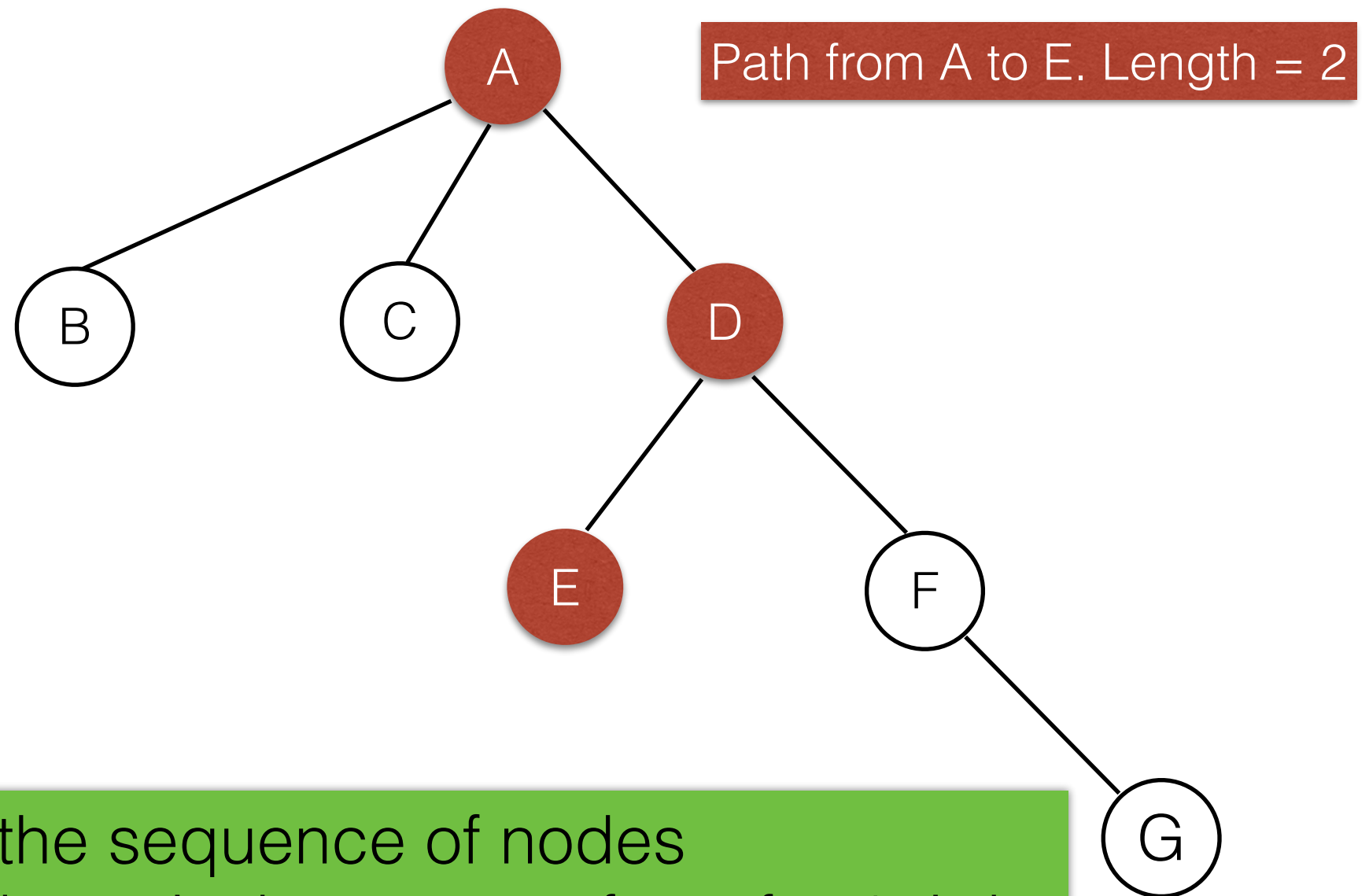
Tree Terminology



Tree Terminology



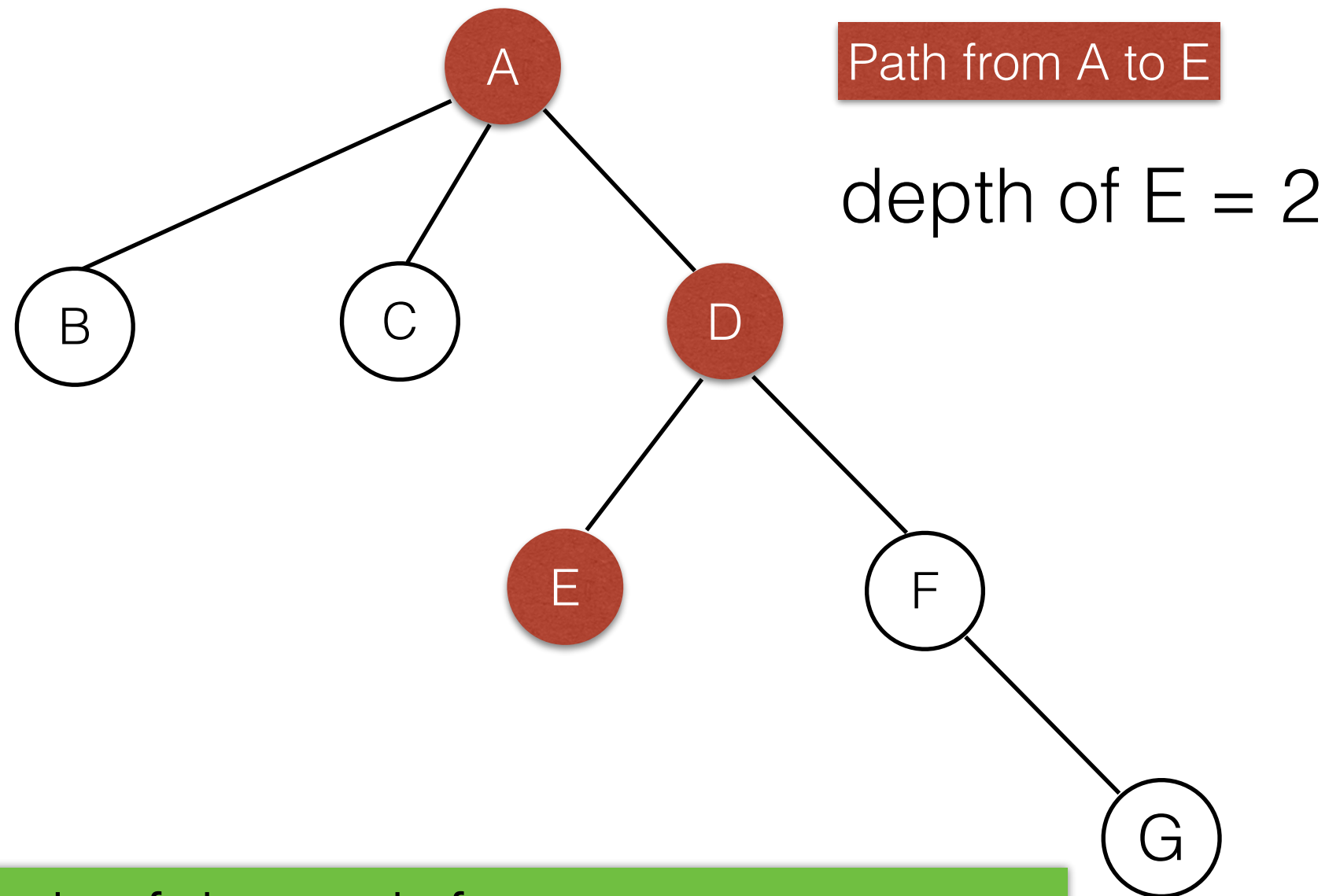
Tree Terminology



Path from n_1 to n_k : the sequence of nodes n_1, n_2, \dots, n_k , such that n_i is the parent of n_{i+1} for $1 \leq i < k$.

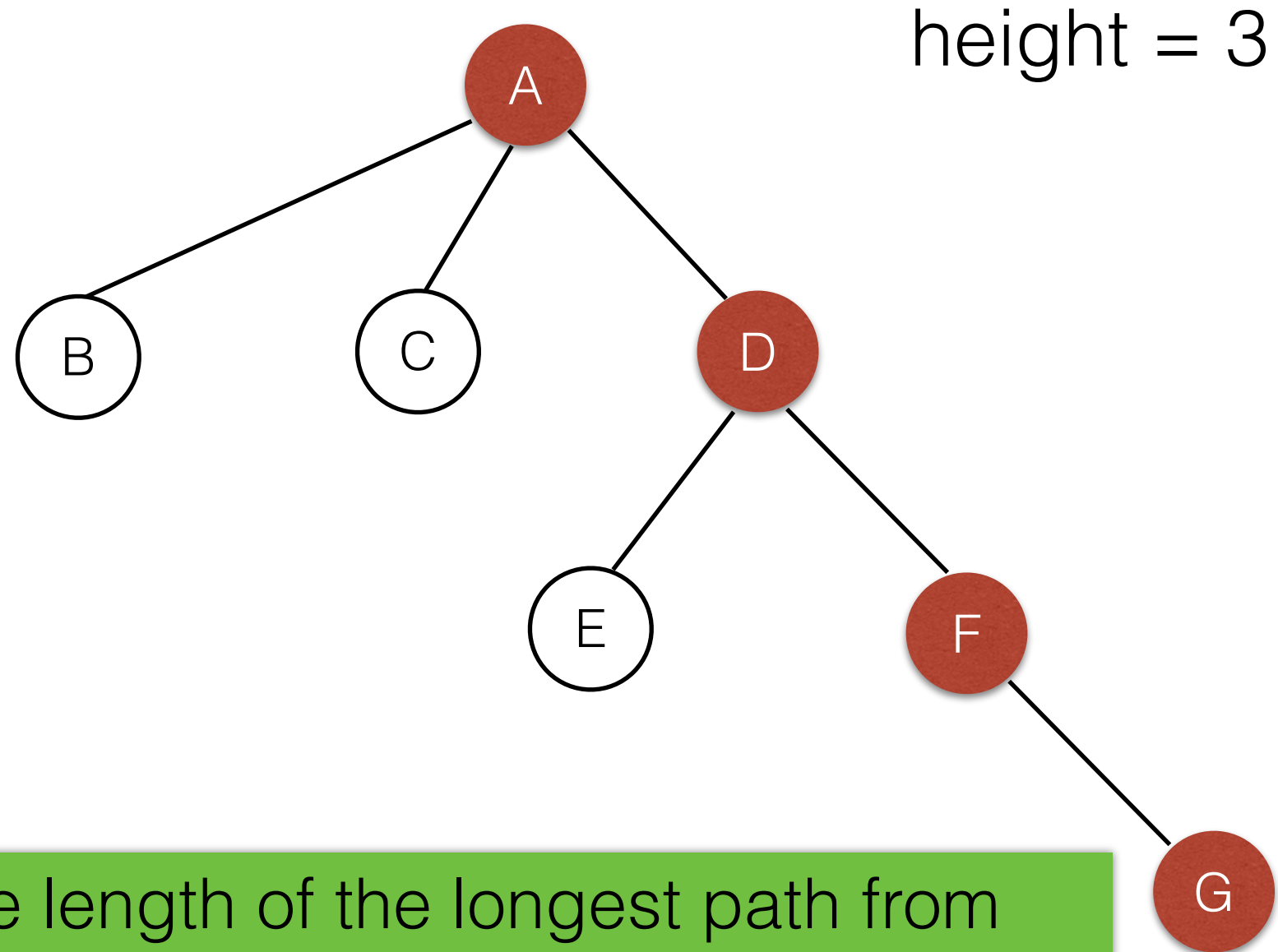
Length of a path: $k-1$ = number of edges on the path

Tree Terminology



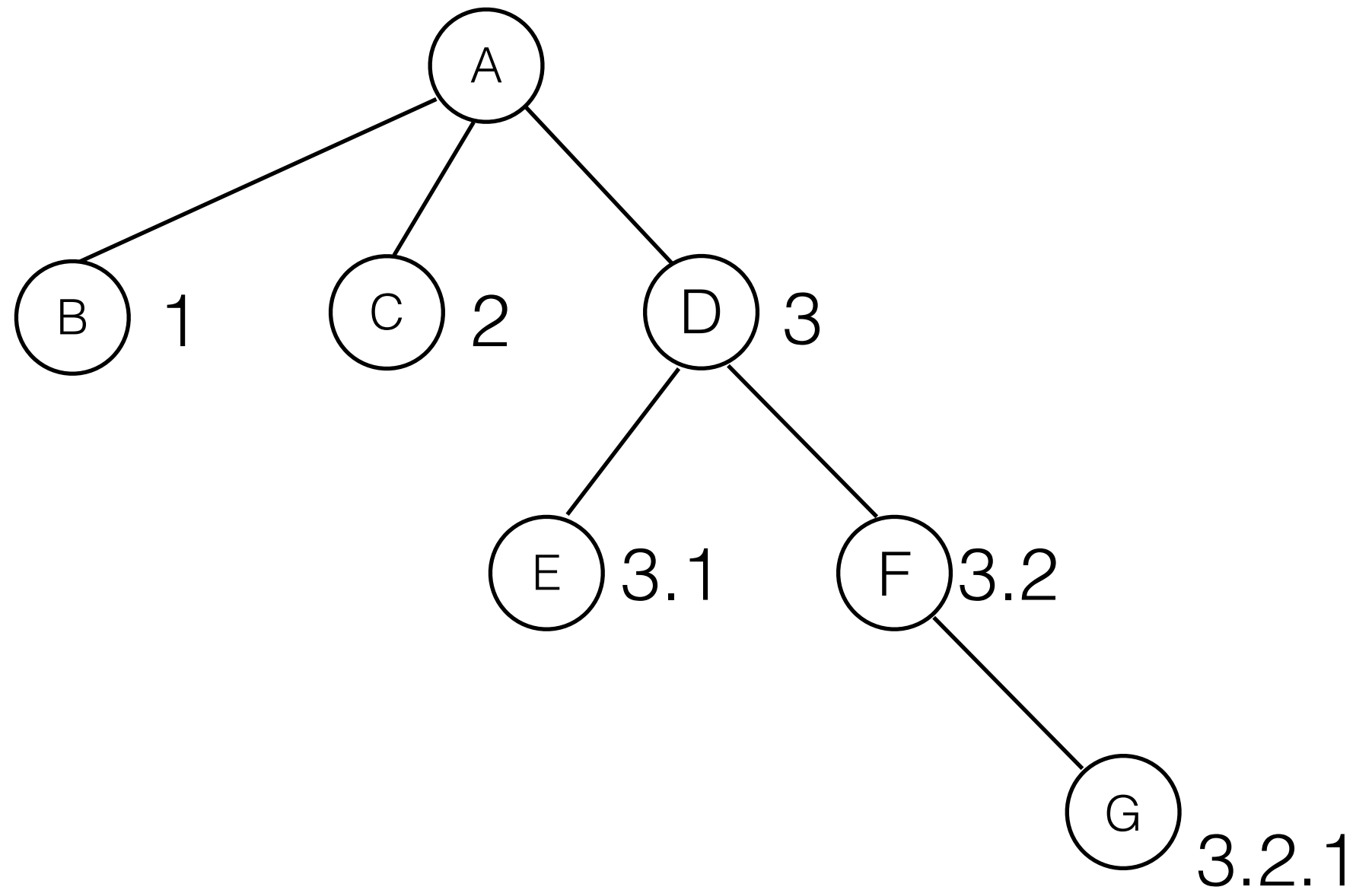
Depth of n_k : the length of the path from root to n_k .

Tree Terminology



Height of tree T : the length of the longest path from root to a leaf.

Tree Addresses

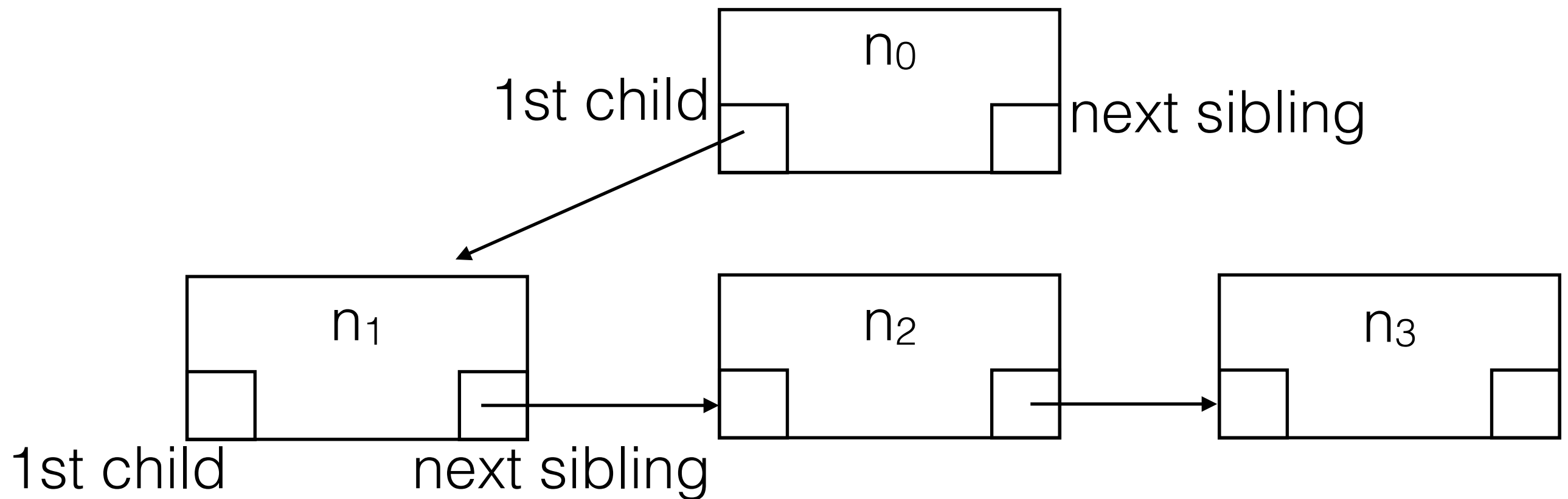


root has an empty address

address of the i -th child of node with address n is $n.i$

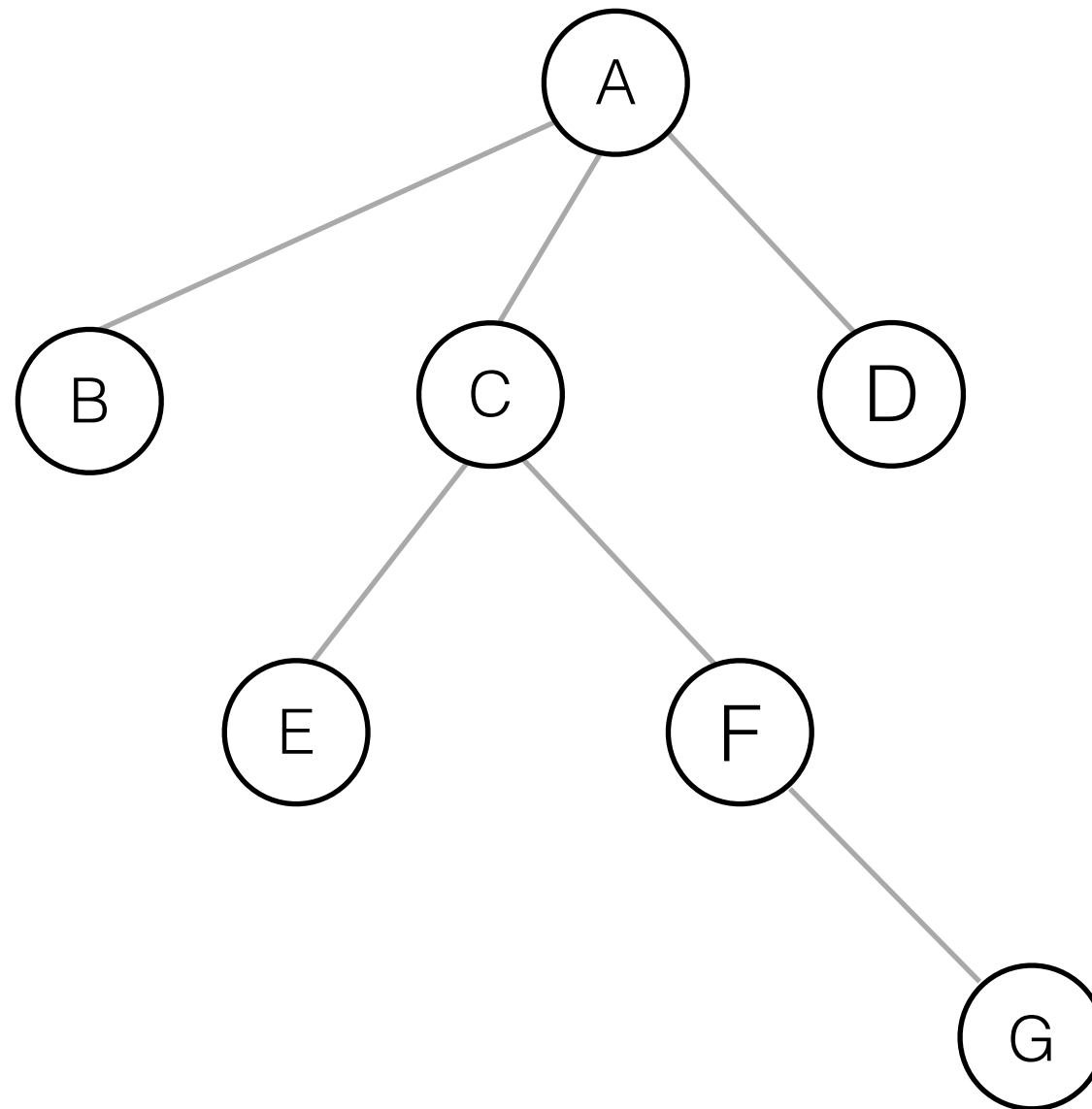
Representing Trees

- Option 1: Organize siblings as a linked list.

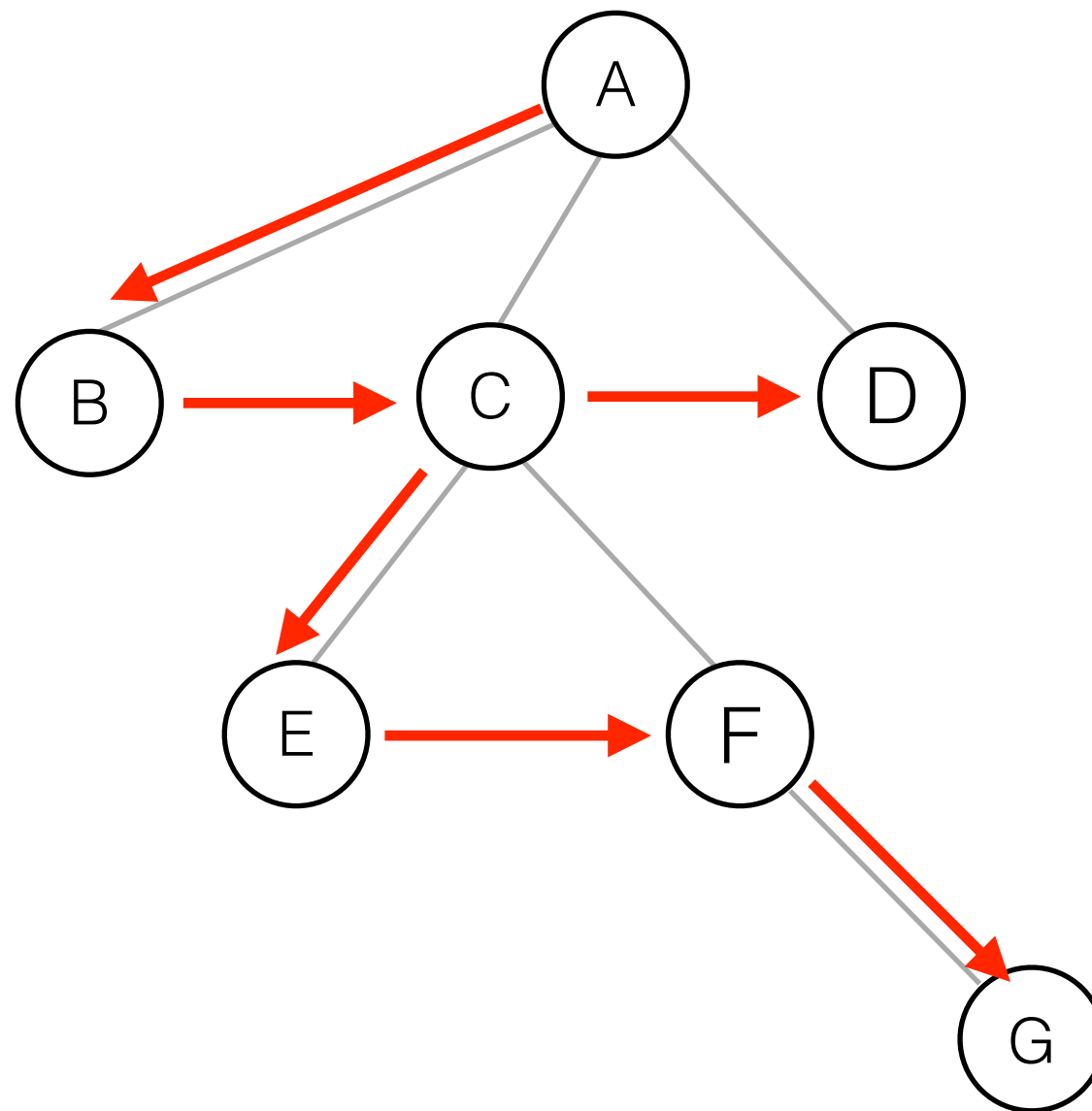


- Problem: Takes longer to find a node from the root.

Siblings as Linked List



Siblings as Linked List

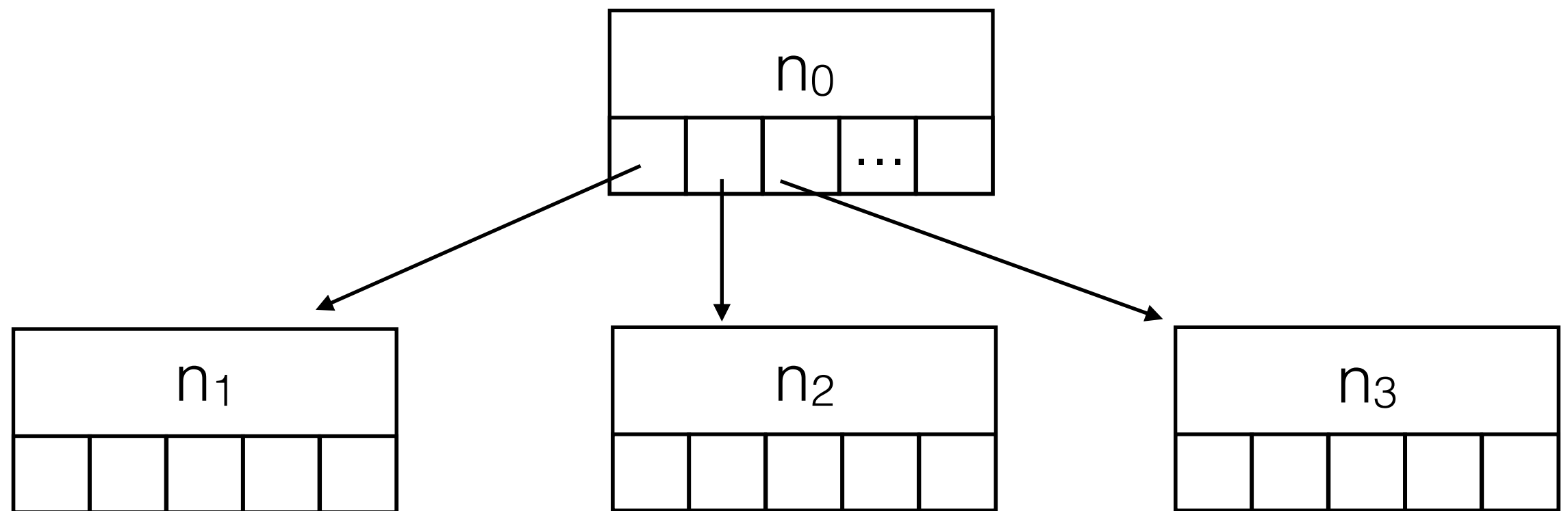


Implementing Siblings as Linked List

```
public class LinkedSiblingTree<E> {  
    private TreeNode<E> root;  
  
    private static class TreeNode<E> {  
        E element;  
        TreeNode<E> firstChild;  
        TreeNode<E> nextSibling;  
        ...  
    }  
    ...  
}
```

Representing Trees

- Option 2: Use an list to keep track of children.



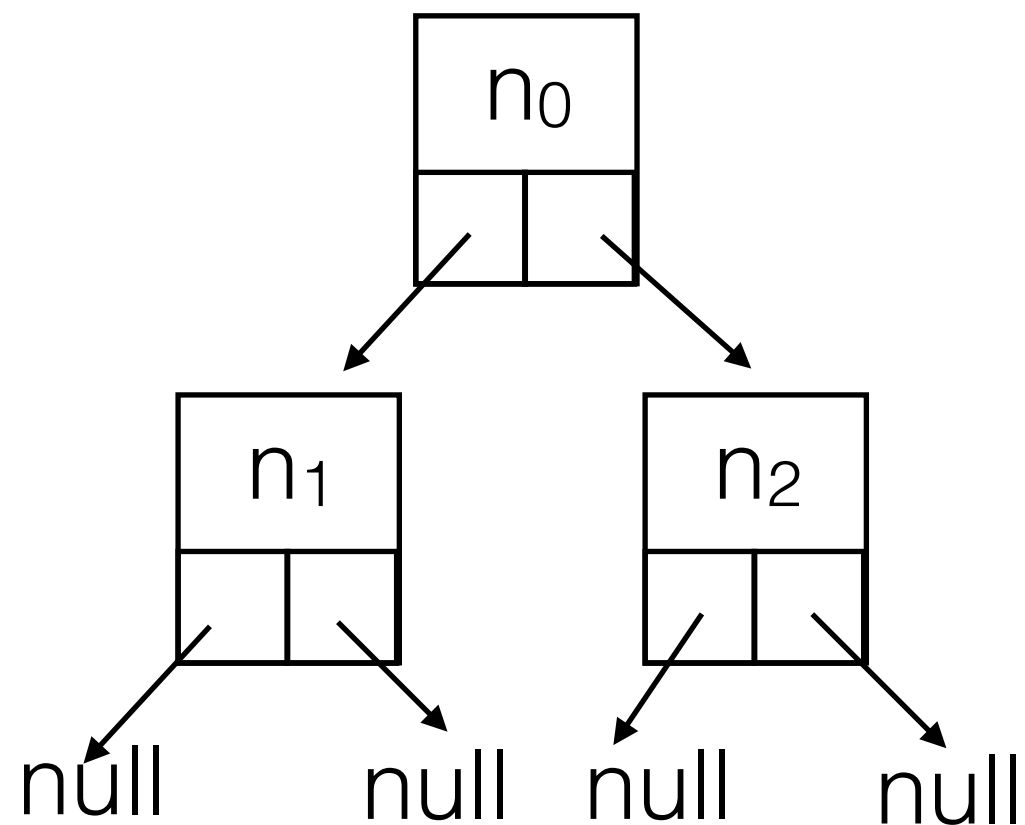
- Only reasonable for small or constant number of children

List of Children

```
public class AdjacencyListTree<E> {  
    private TreeNode<E> root;  
  
    private static class TreeNode<E> {  
        E element;  
        List<TreeNode<E>> children;  
        ...  
    }  
    ...  
}
```

Representing Trees

- Option 3: Every node has **fixed** number of references to children.



- More memory efficient than a complete list of children, but only reasonable for small or constant number of children

Binary Trees

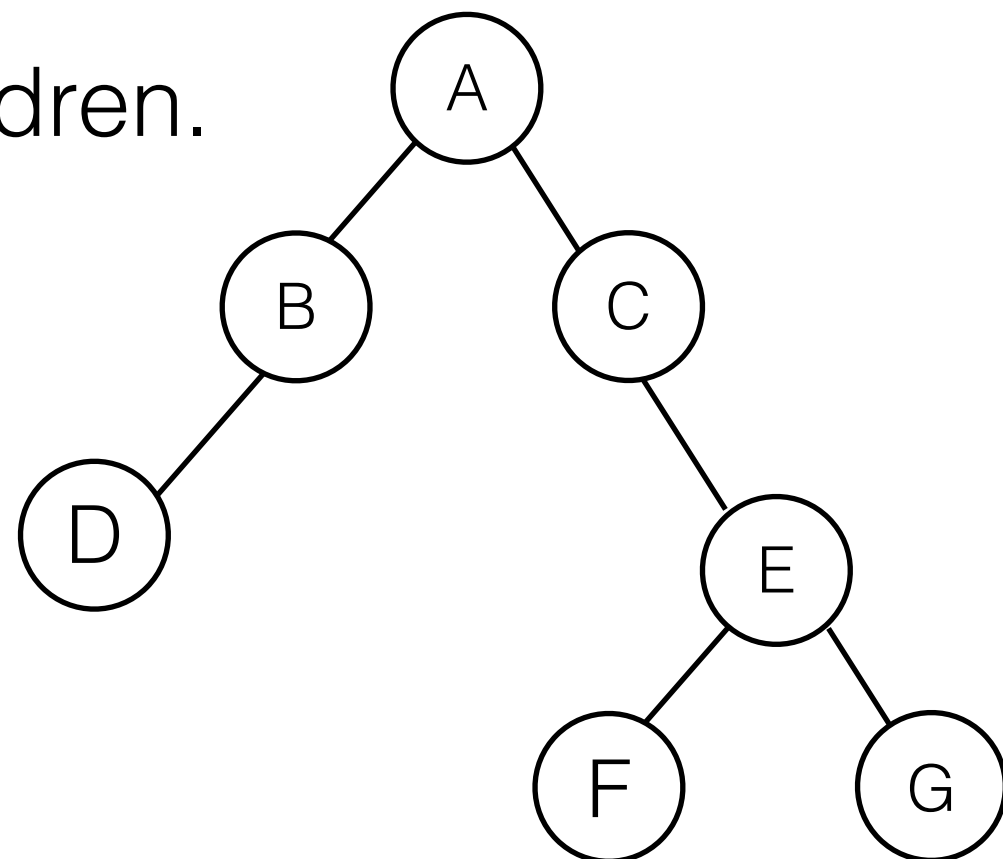
- For binary trees, the number of children is at most two.
- Binary trees are very common in data structures and algorithms.
- Binary tree algorithms are convenient to analyze.

Implementing Binary Trees

```
public class BinaryTree<T> {  
  
    // The BinaryTree is essentially just a wrapper around the  
    // linked structure of BinaryNodes, rooted in root.  
    private BinaryNode<T> root;  
  
    /**  
     * Represent a binary subtree.  
     */  
    private static class BinaryNode<T>{  
        public T data;  
        public BinaryNode<T> left;  
        public BinaryNode<T> right;  
        ...  
    }  
    ...  
}
```


Full Binary Trees

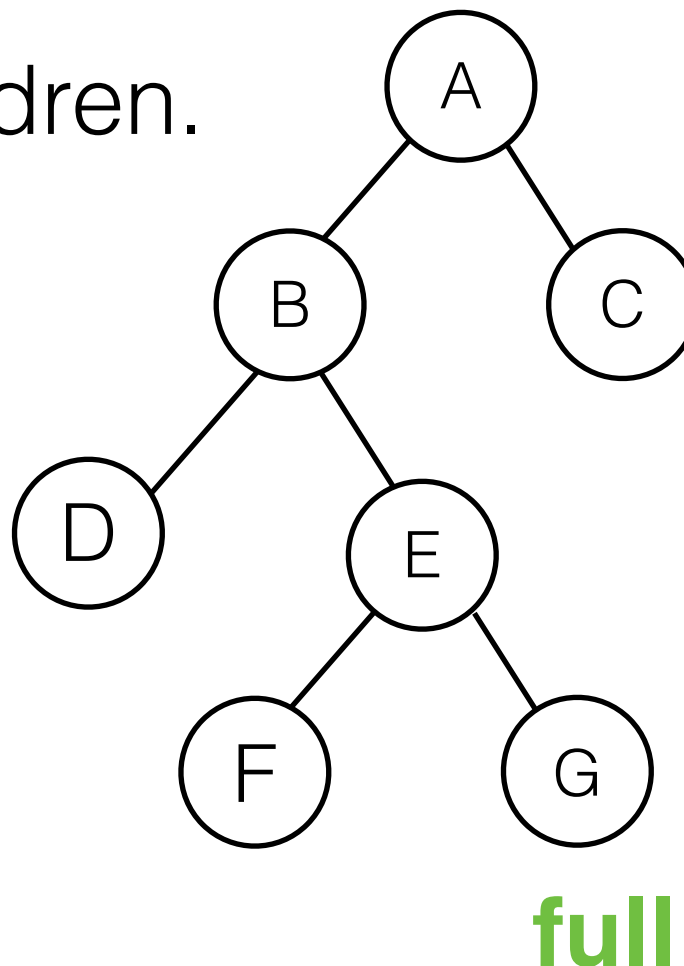
- In a full binary tree every node
 - is either a leaf.
 - or has exactly two children.



not full

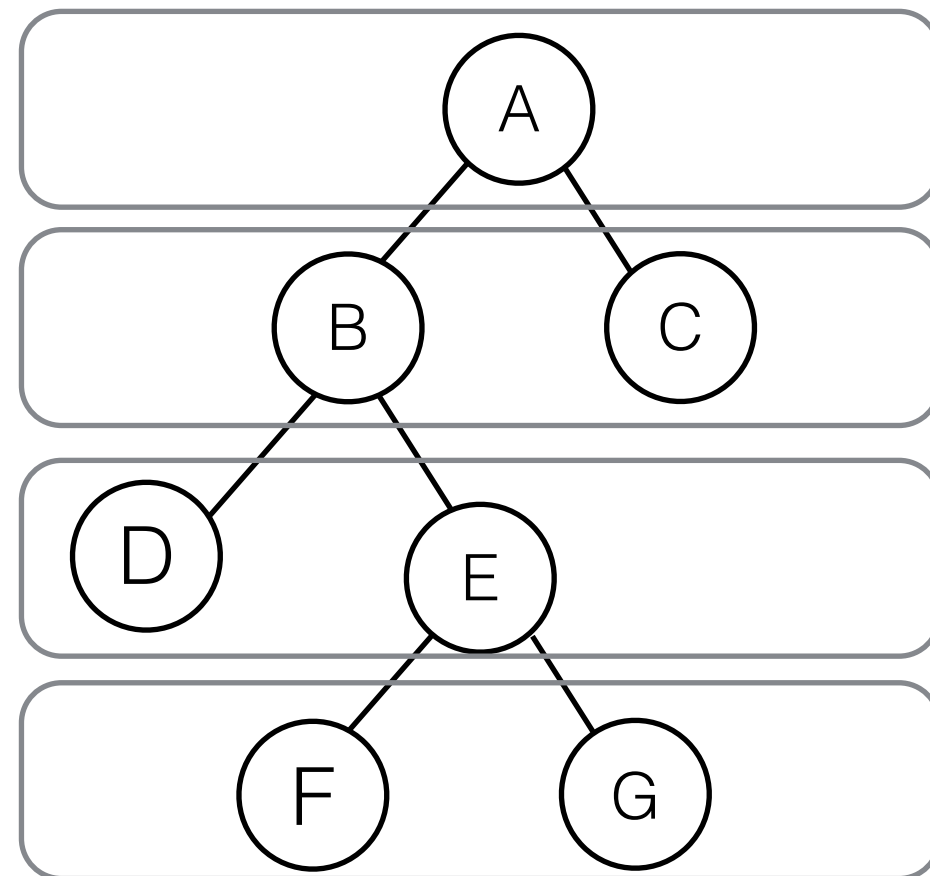
Full Binary Trees

- In a full binary tree every node
 - is either a leaf.
 - or has exactly two children.



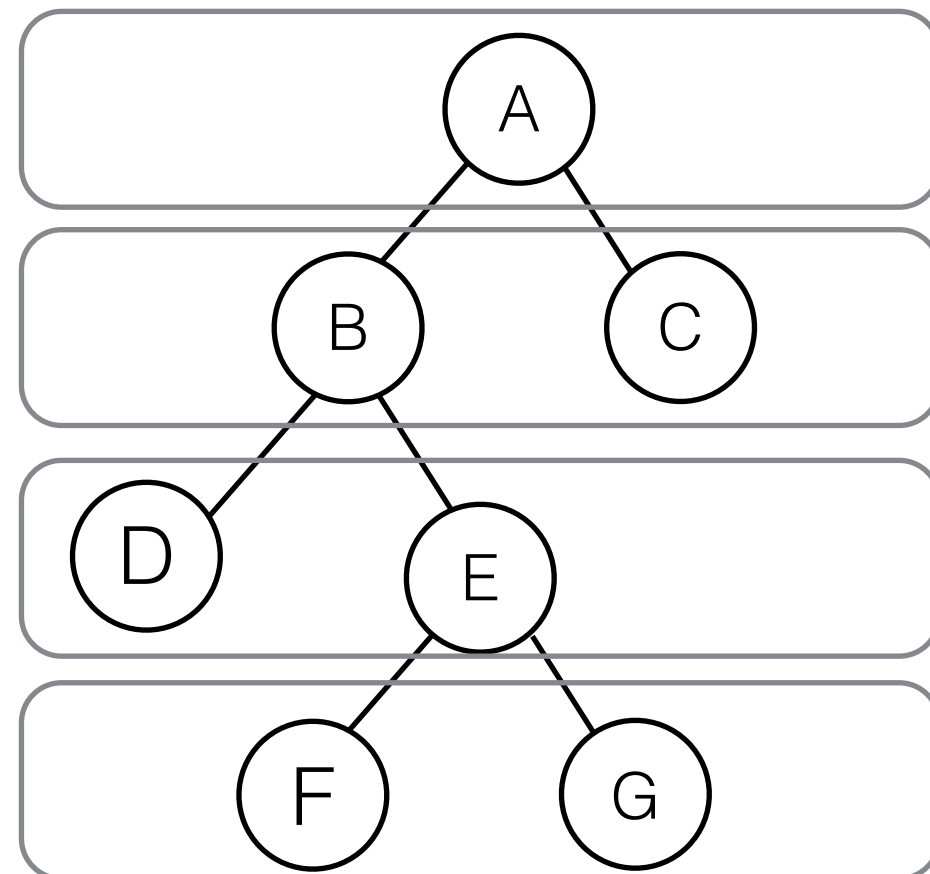
Complete Binary Trees

- A complete binary tree is a binary tree in which all levels (except possibly the last) are completely filled.



Complete Binary Trees

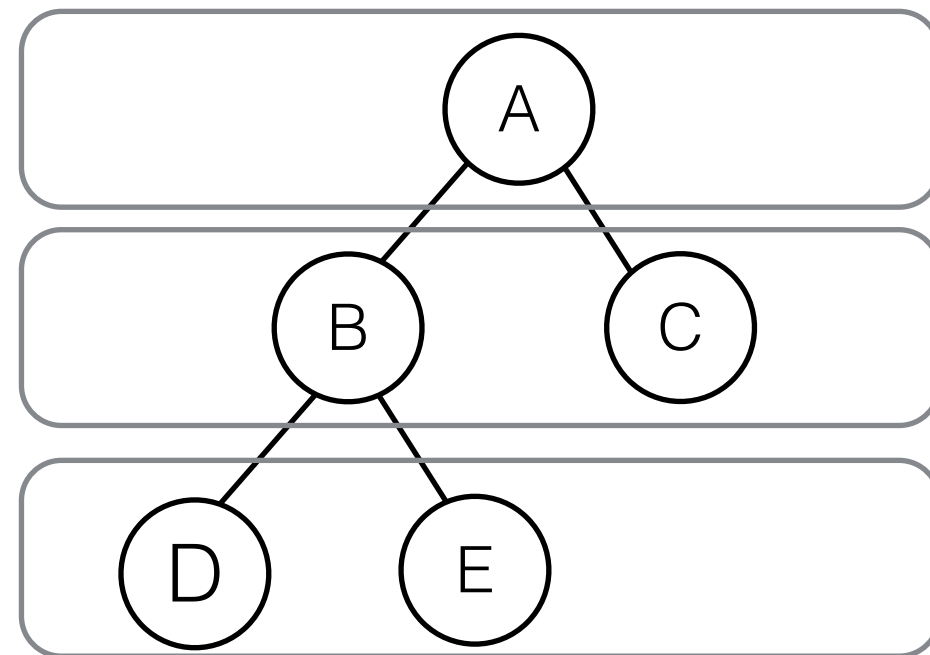
- A complete binary tree is a binary tree in which all levels (except possibly the last) are completely filled.



full, but not complete

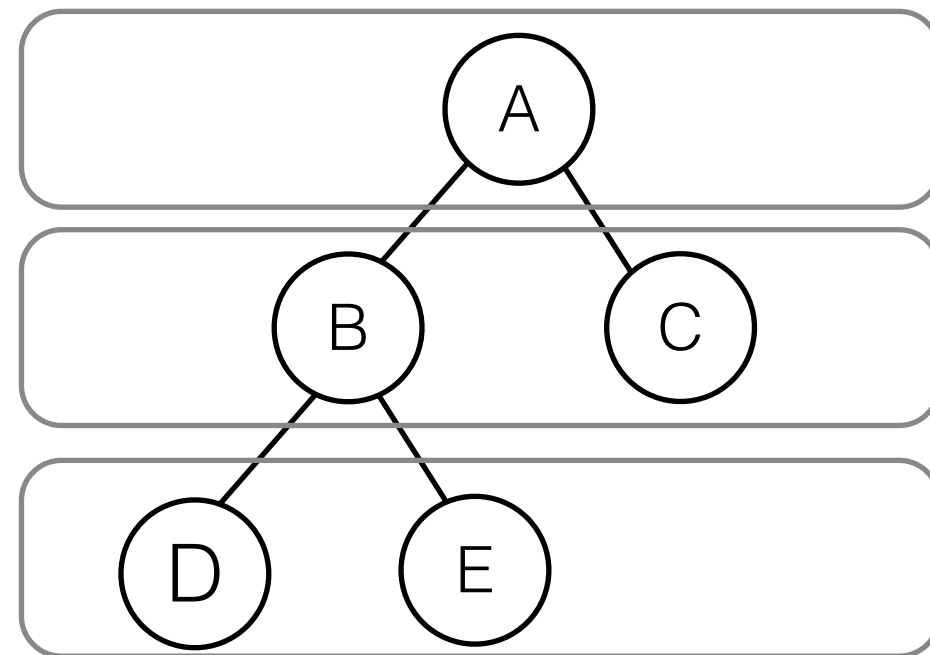
Complete Binary Trees

- A complete binary tree is a binary tree in which all levels (except possibly the last) are completely filled and every node is as far left as possible.



Complete Binary Trees

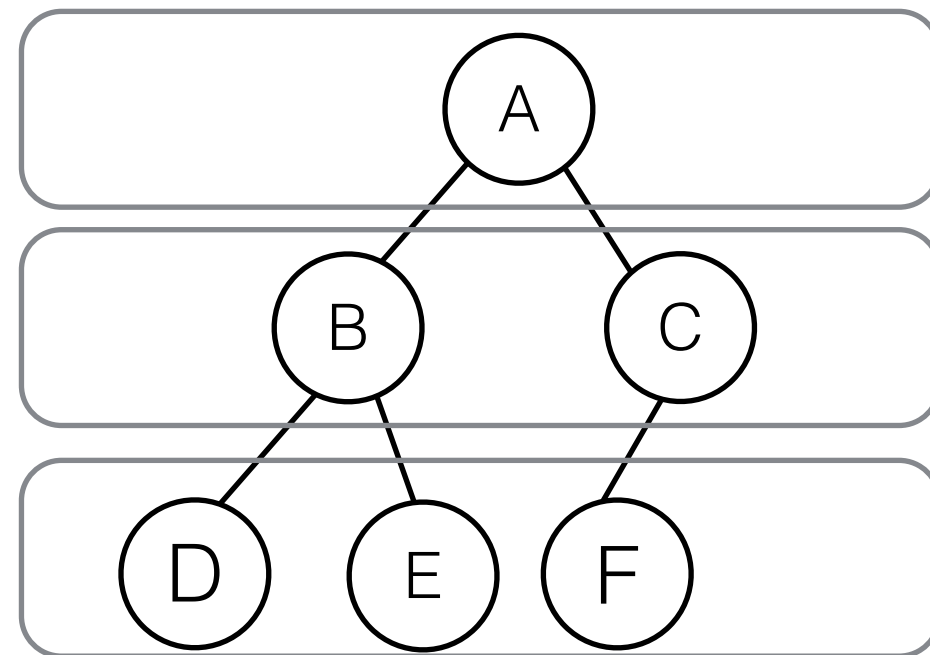
- A complete binary tree is a binary tree in which all levels (except possibly the last) are completely filled and every node is as far left as possible.



complete

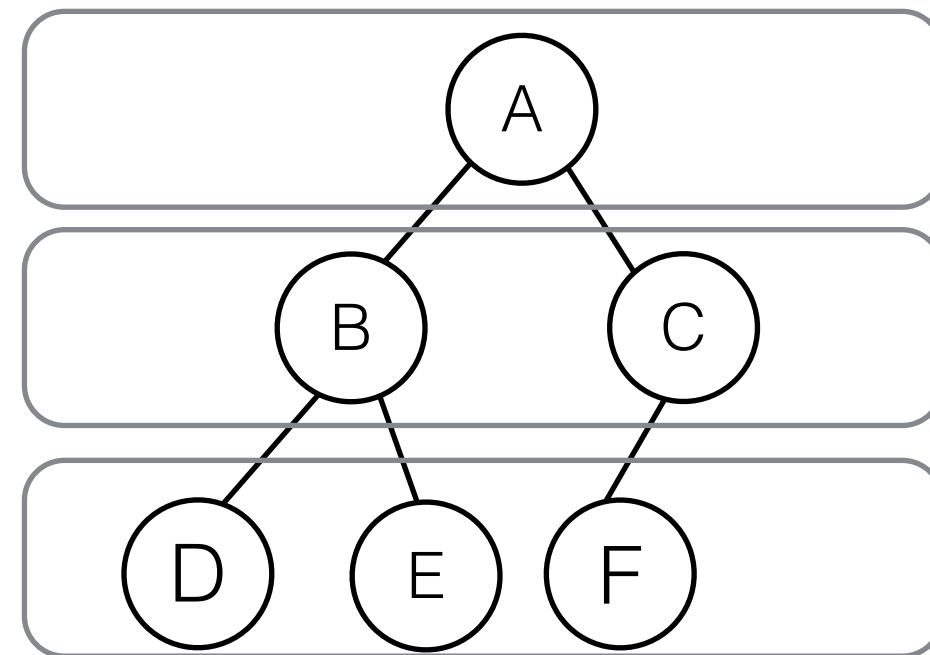
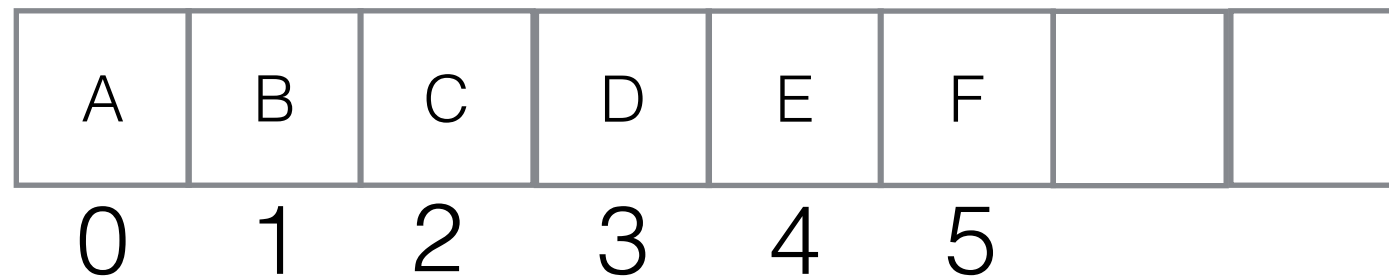
Complete Binary Trees

- A complete binary tree is a binary tree in which all levels (except possibly the last) are completely filled and every node is as far left as possible.



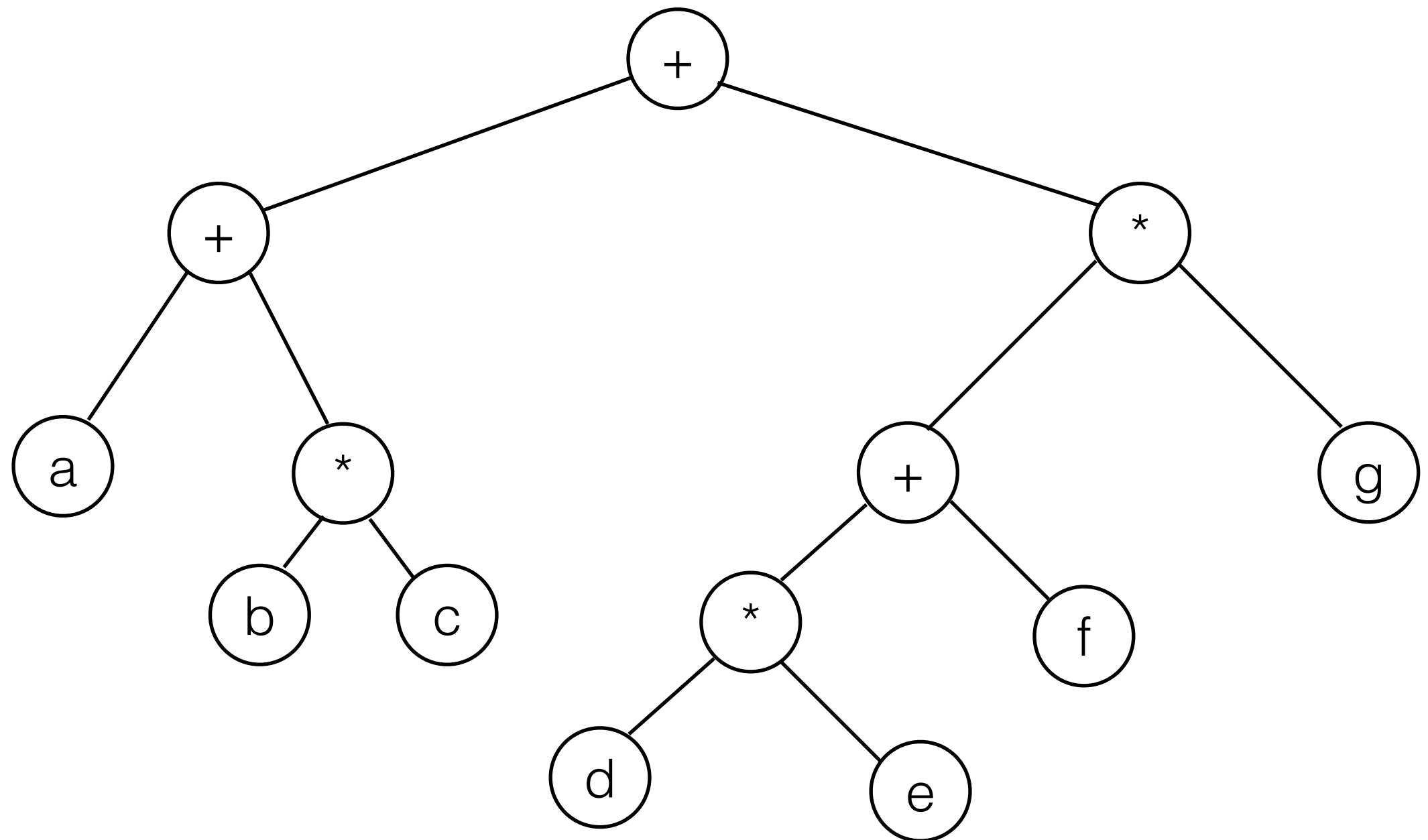
complete but not full

Storing Complete Binary Trees in Arrays



Structure of the tree only depends on the number of nodes.

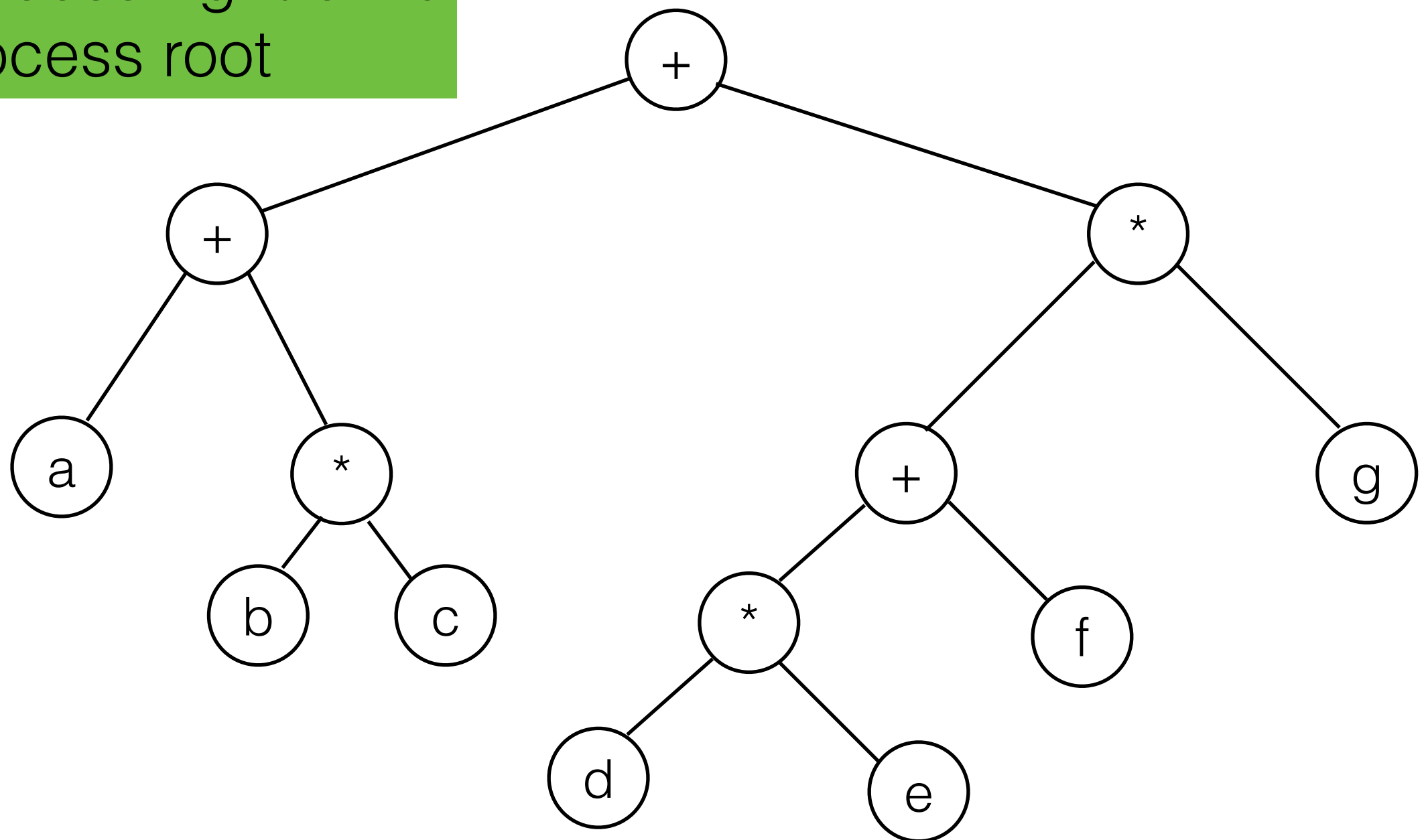
Example Binary Tree: Expression Trees



Tree Traversals: Post-order

1. Process left child
2. Process right child
3. Process root

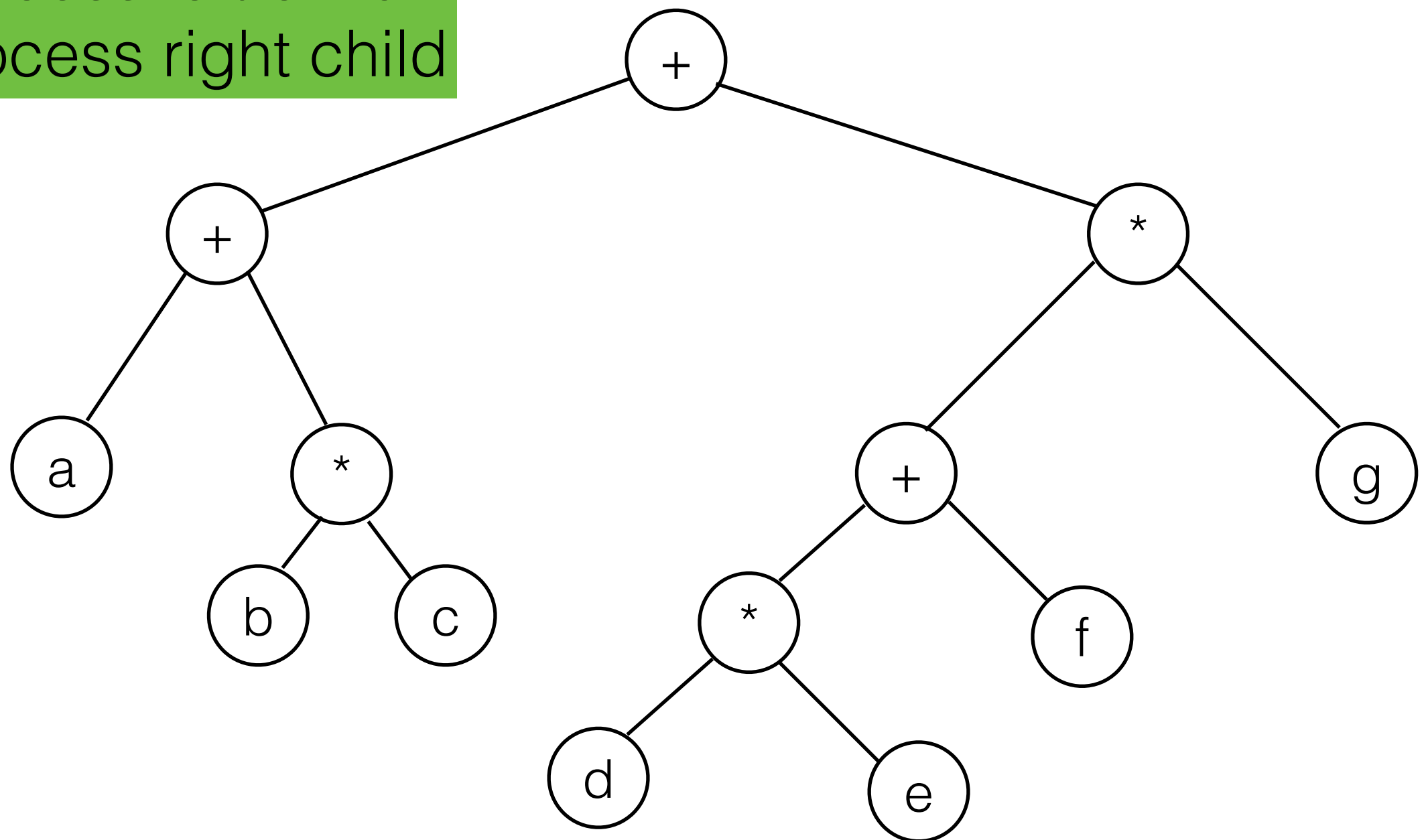
a b c * + d e * f + g *



Tree Traversals: Pre-order

1. Process root
2. Process left child
3. Process right child

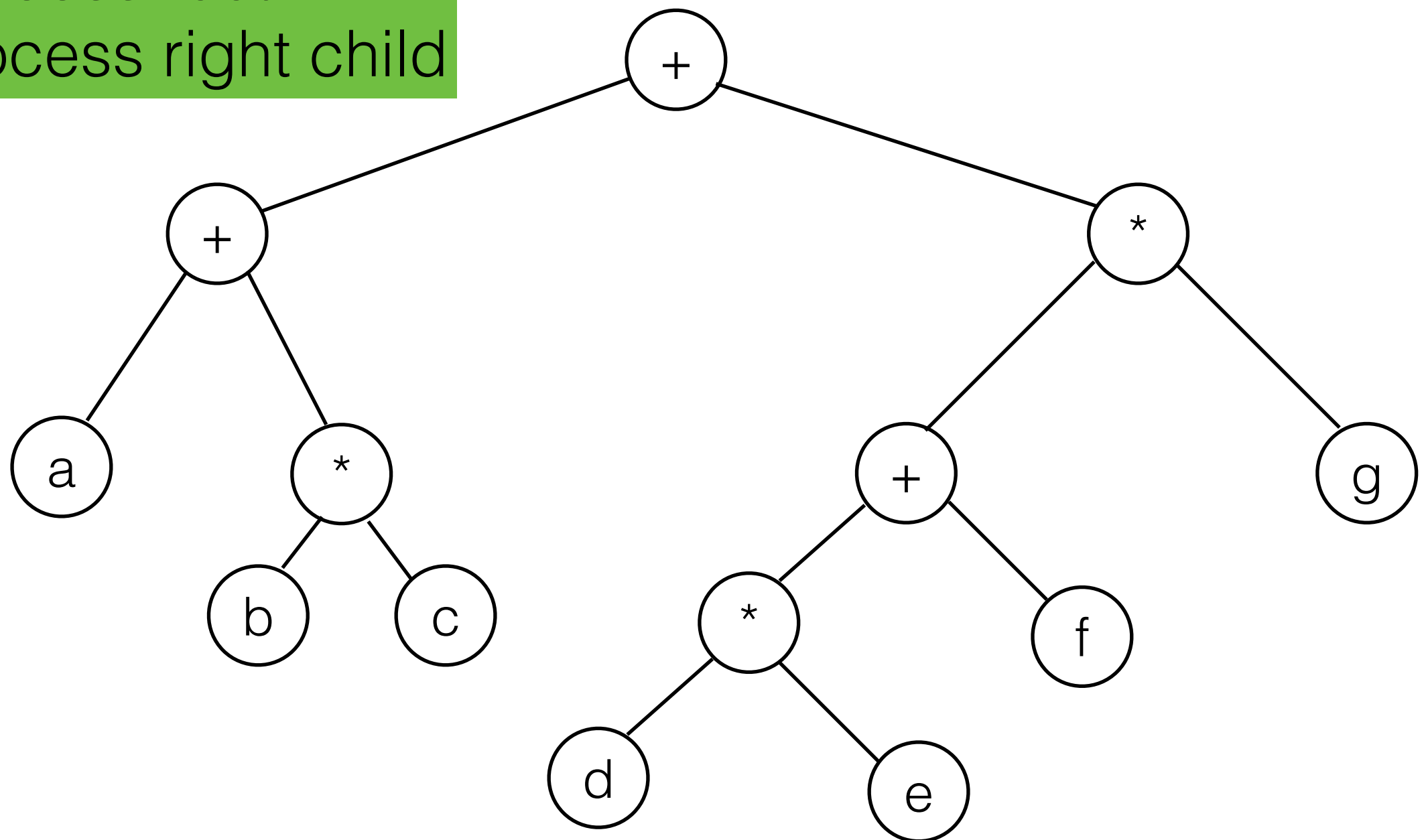
$++a*bc*+*defg$



Tree Traversals: In-order

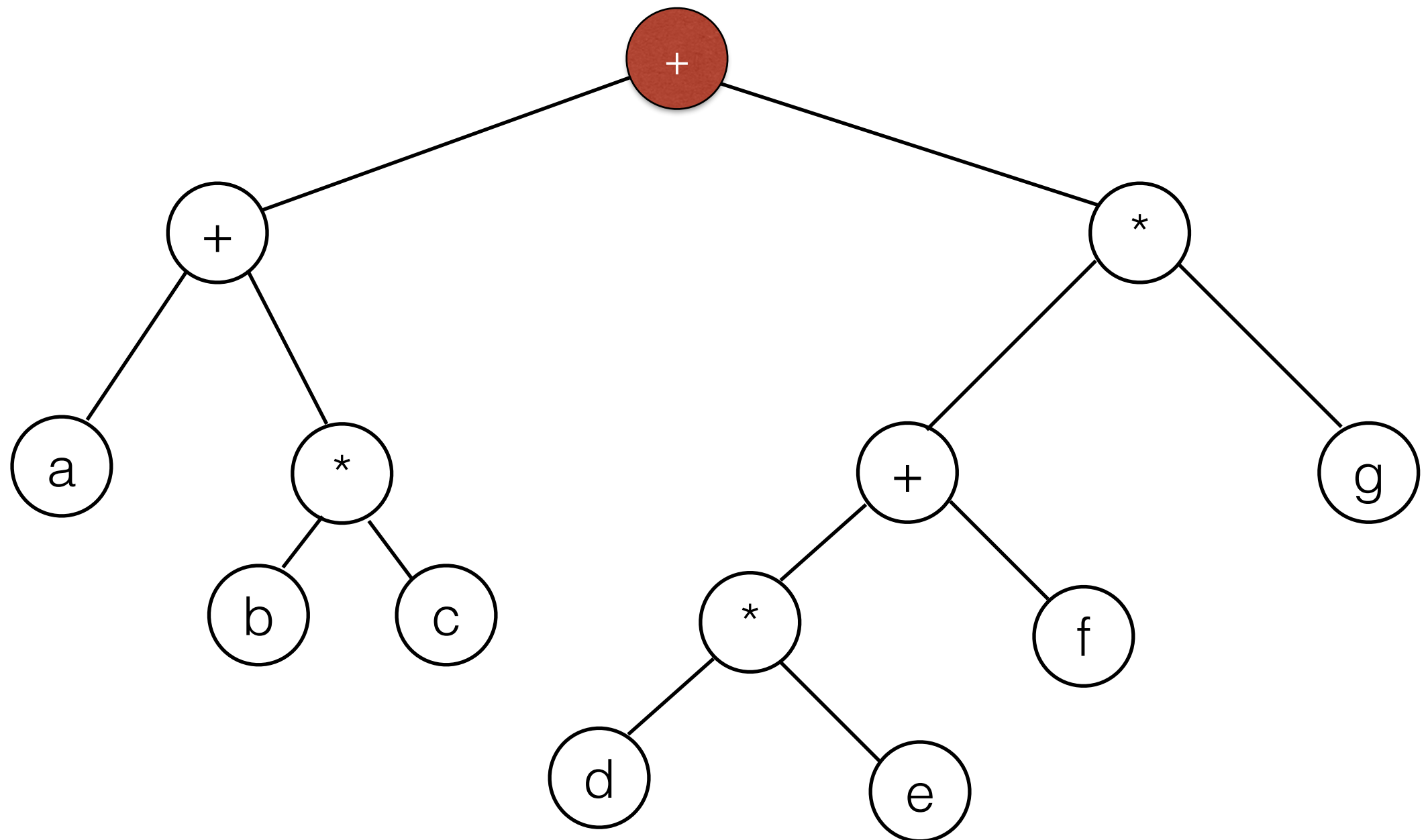
1. Process left child
2. Process root
3. Process right child

$$(a + b * c) + (d * e + f) * g$$



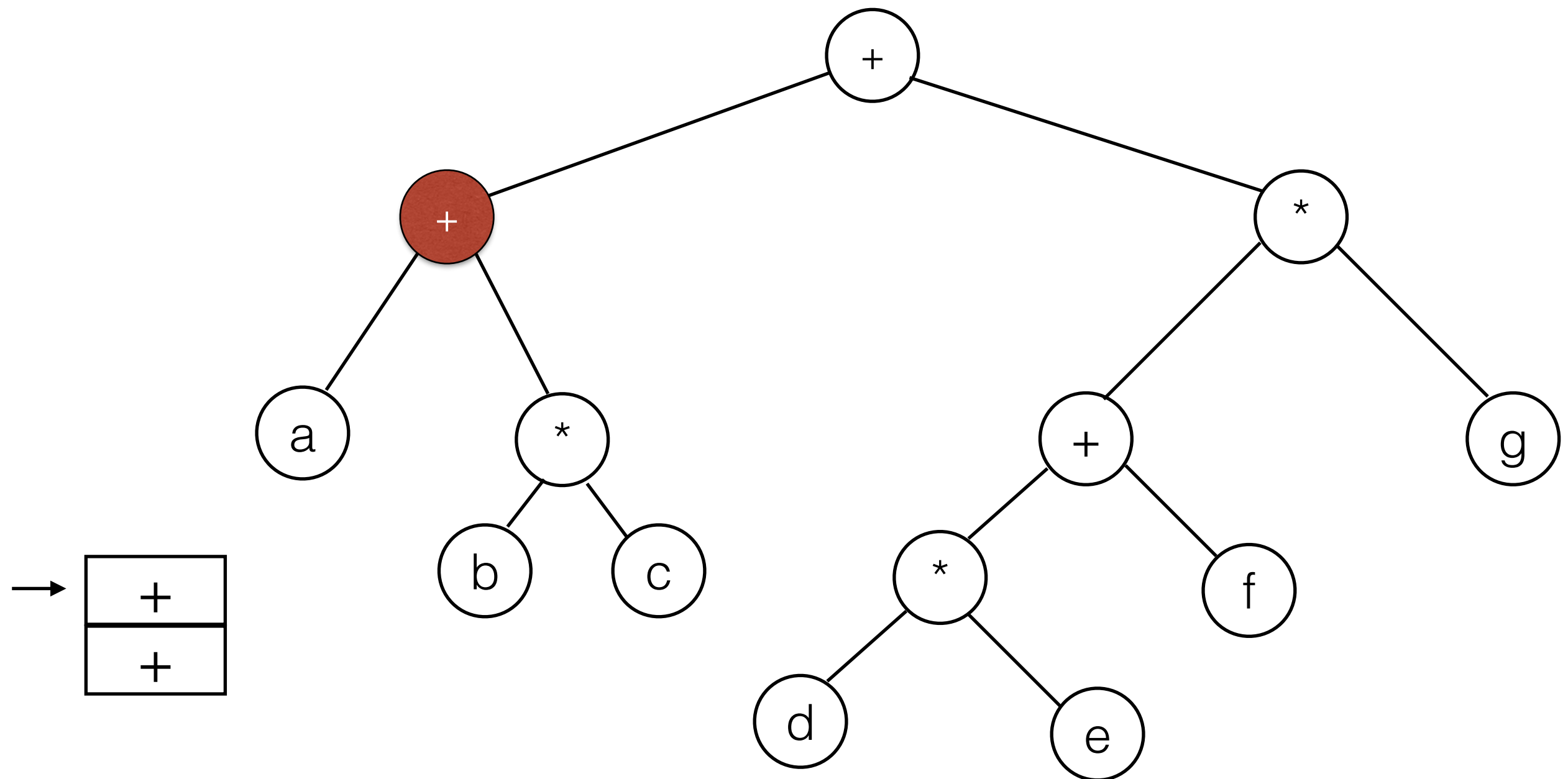
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



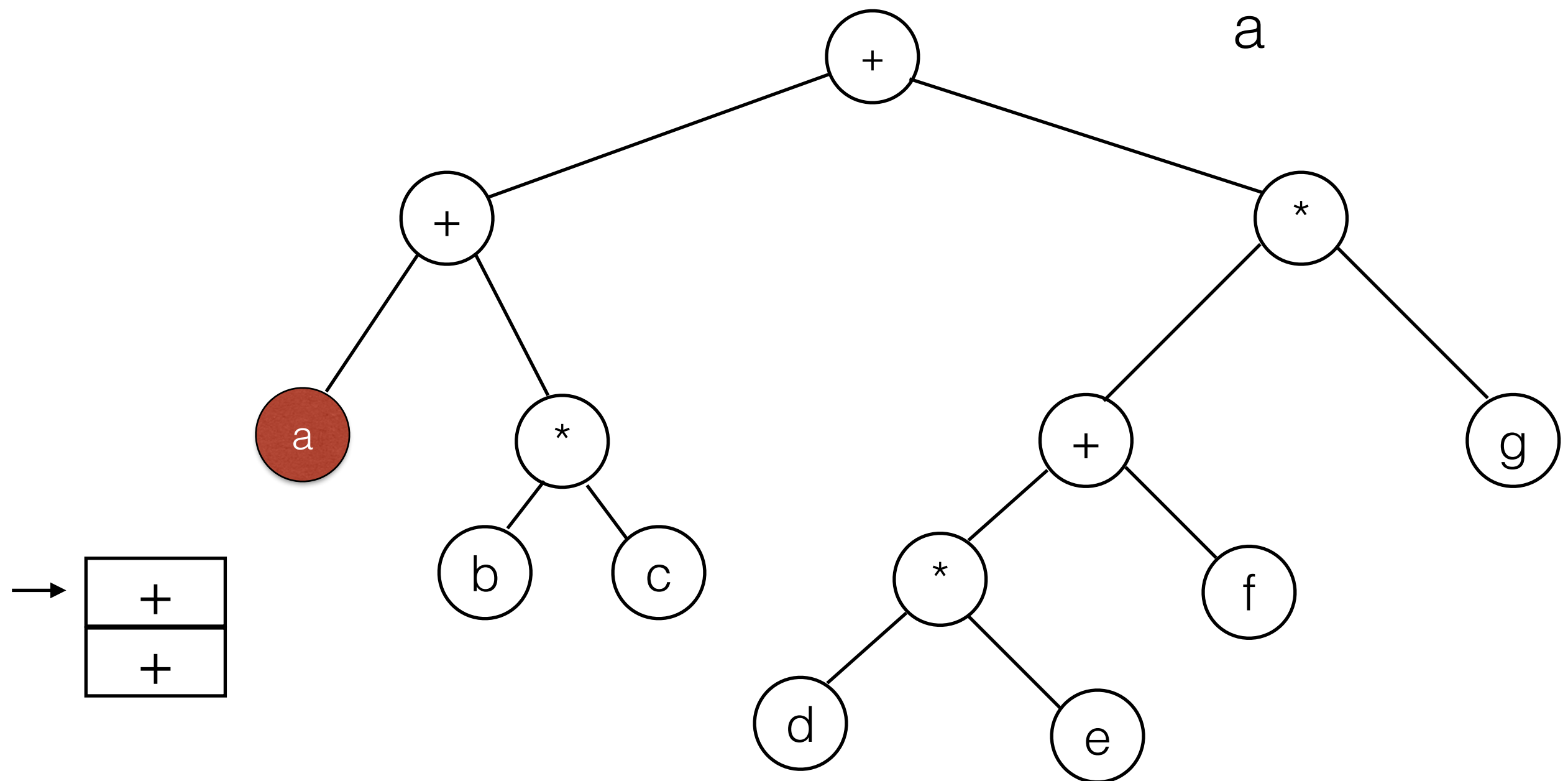
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



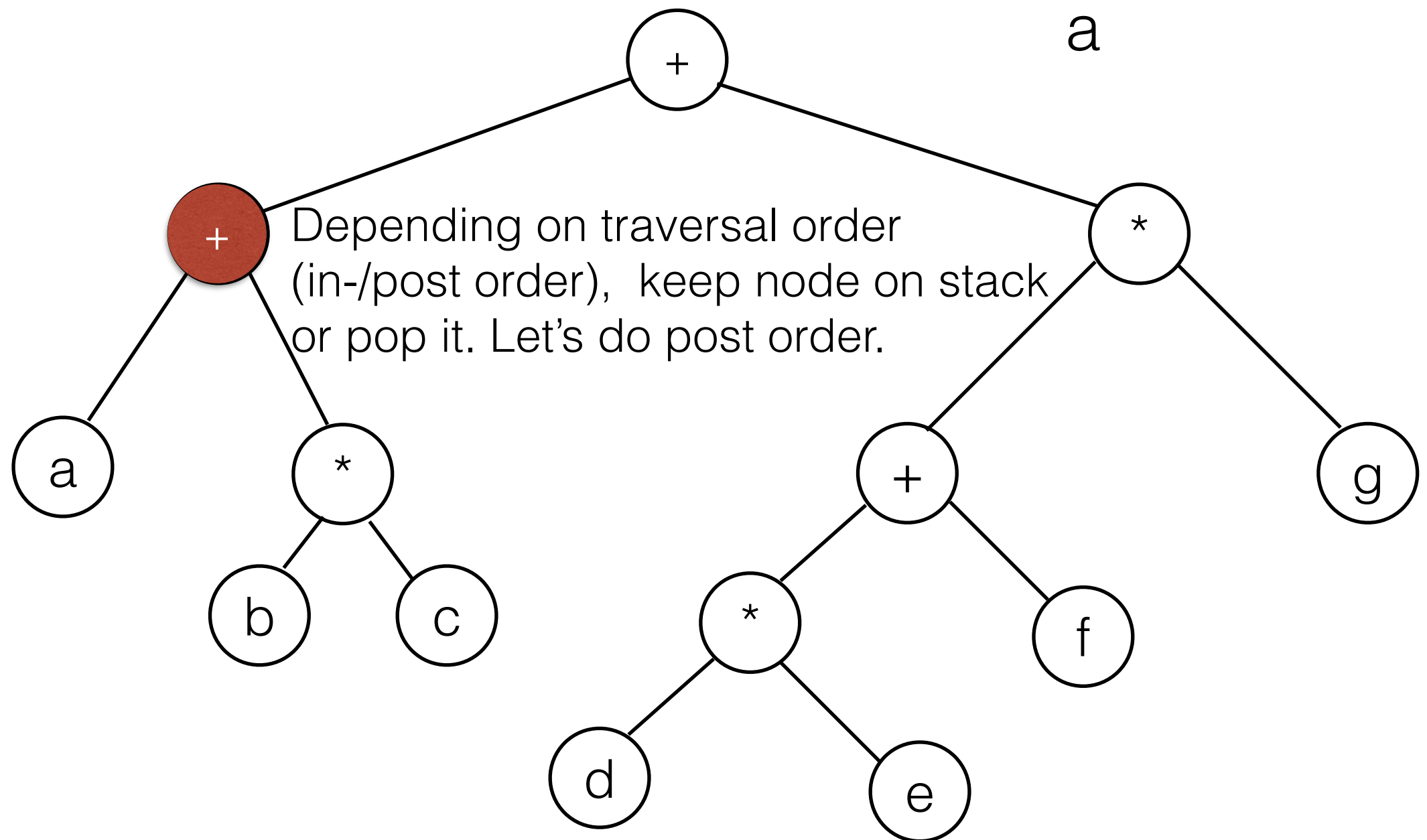
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



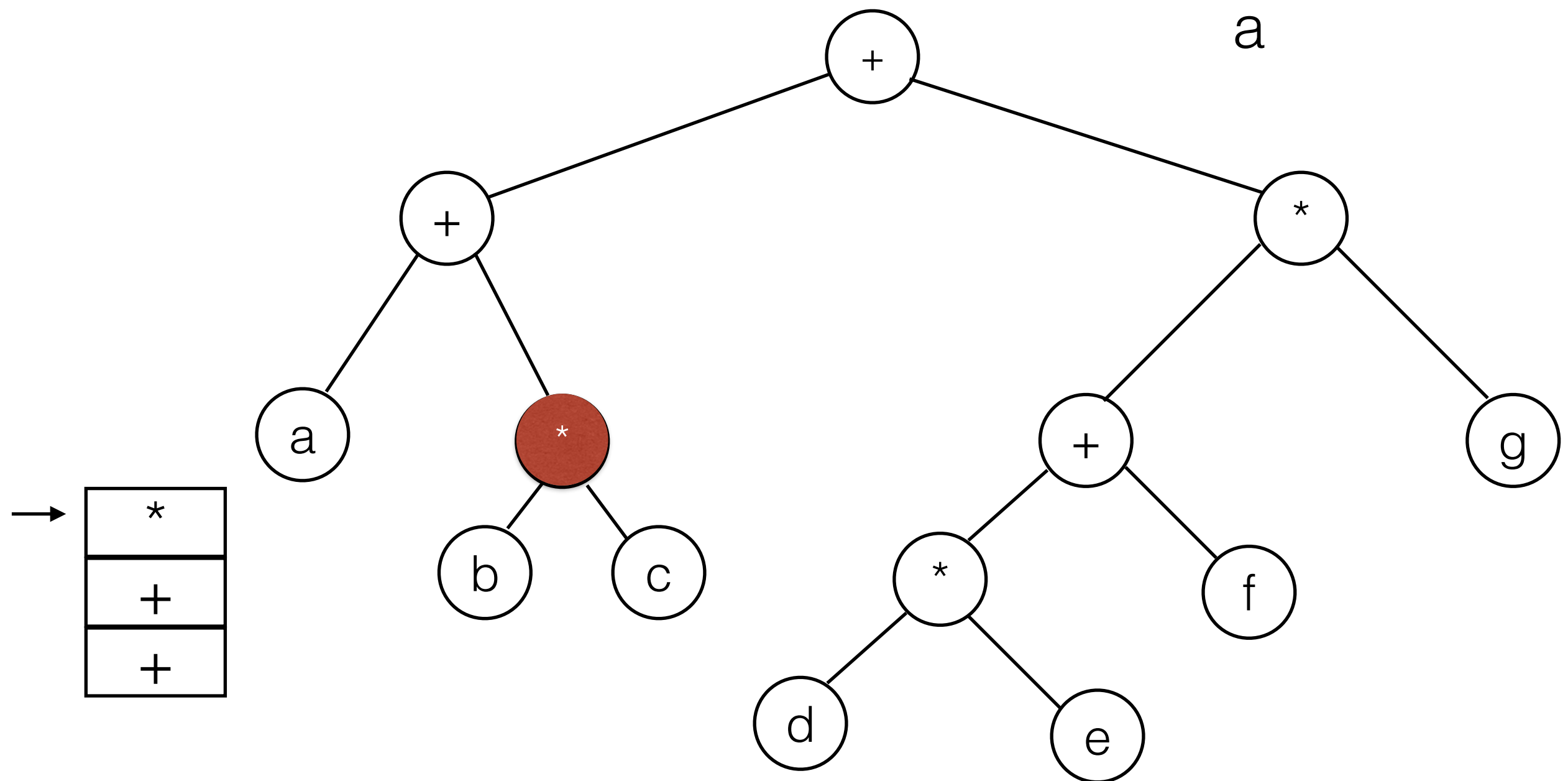
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



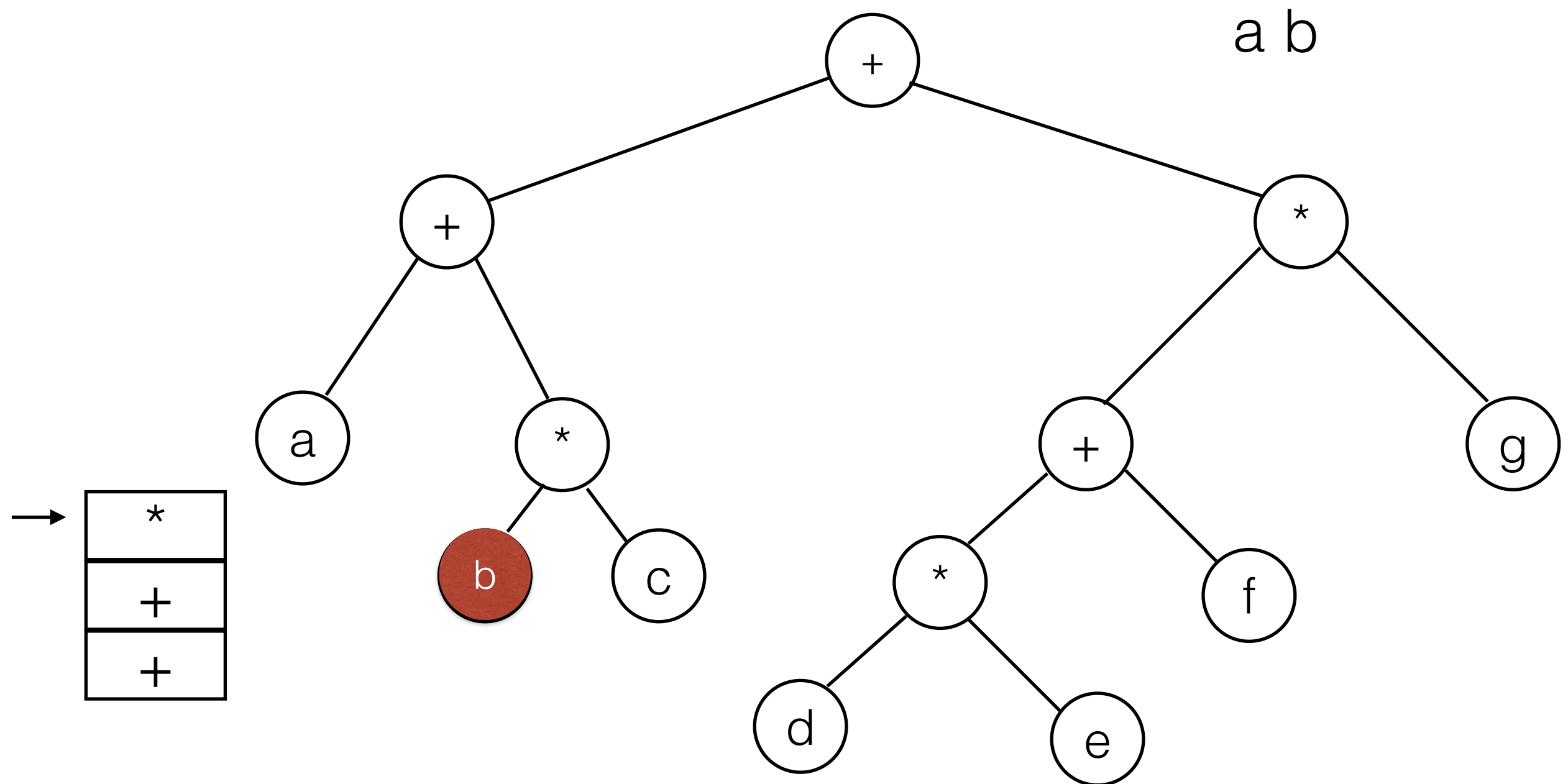
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



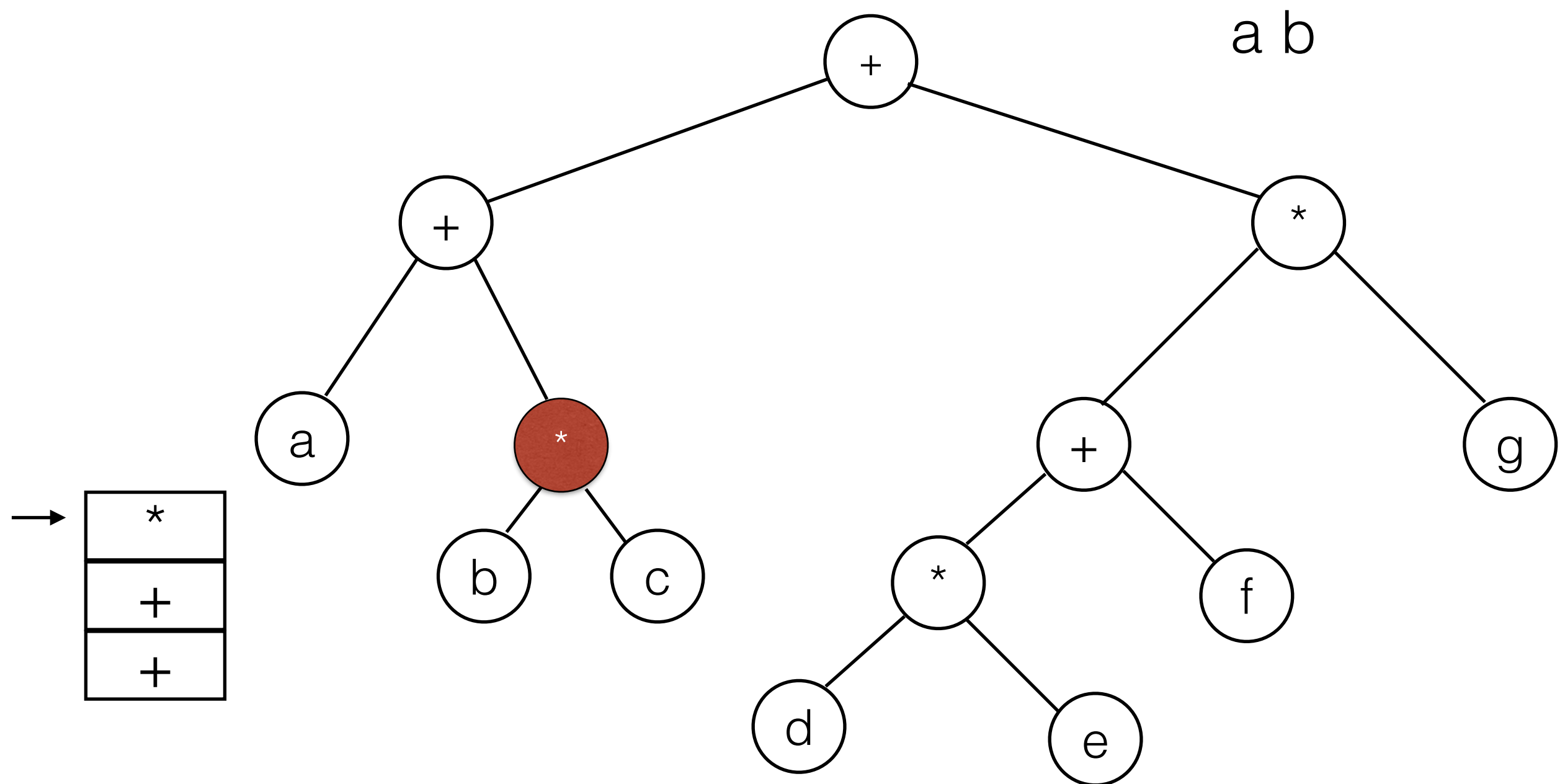
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



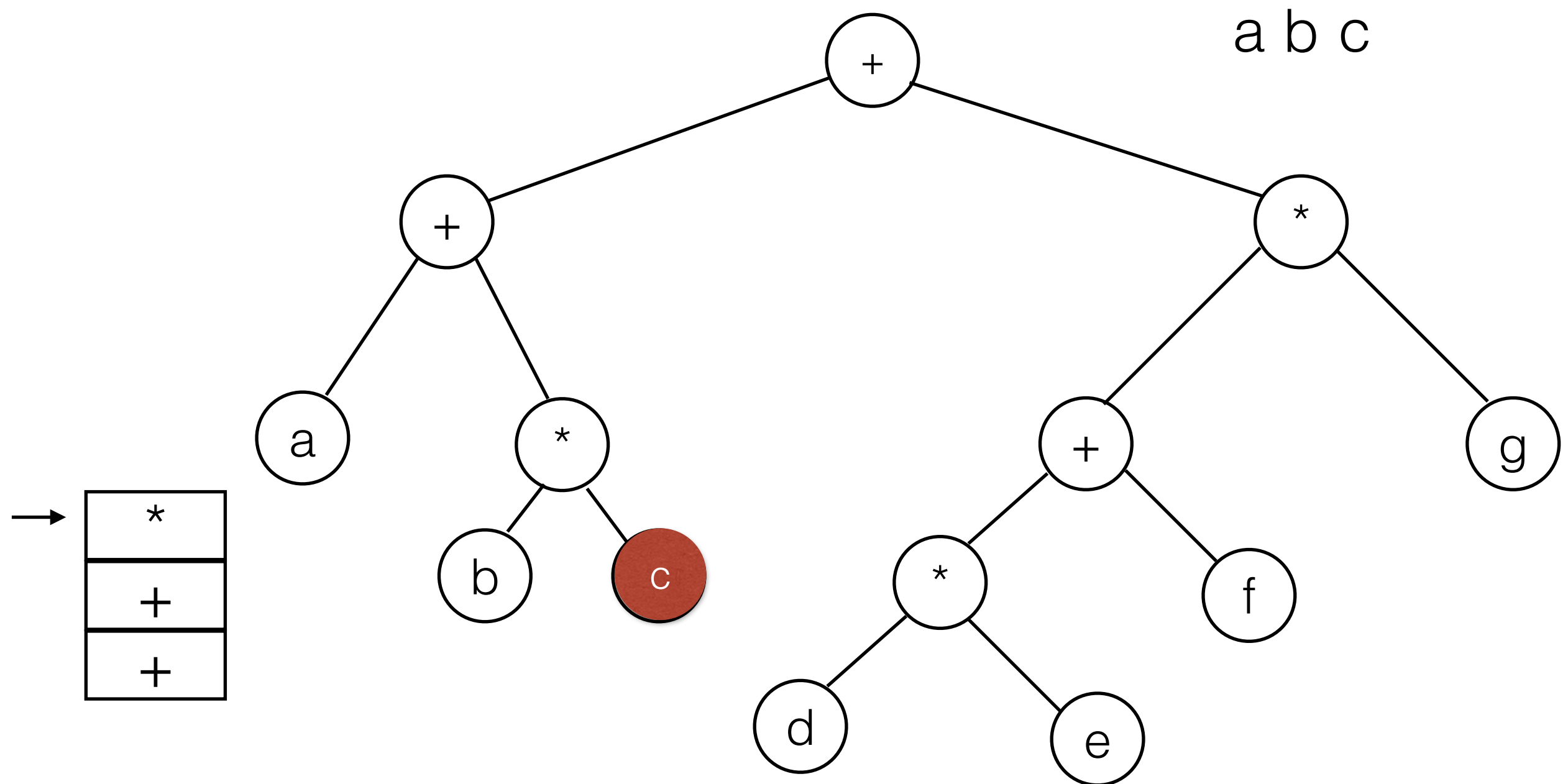
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



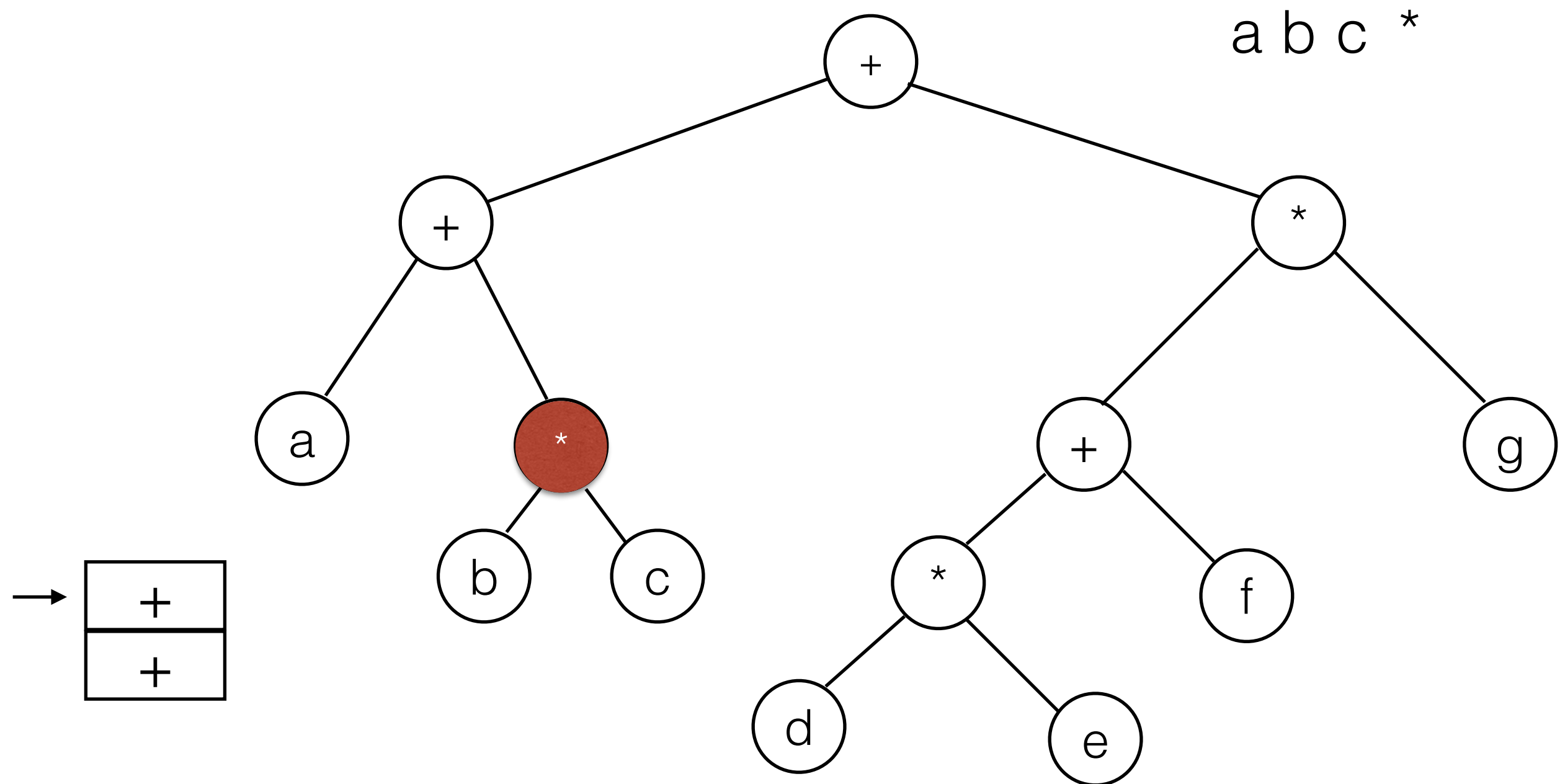
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



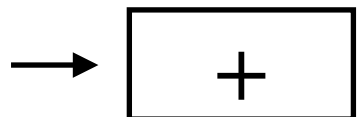
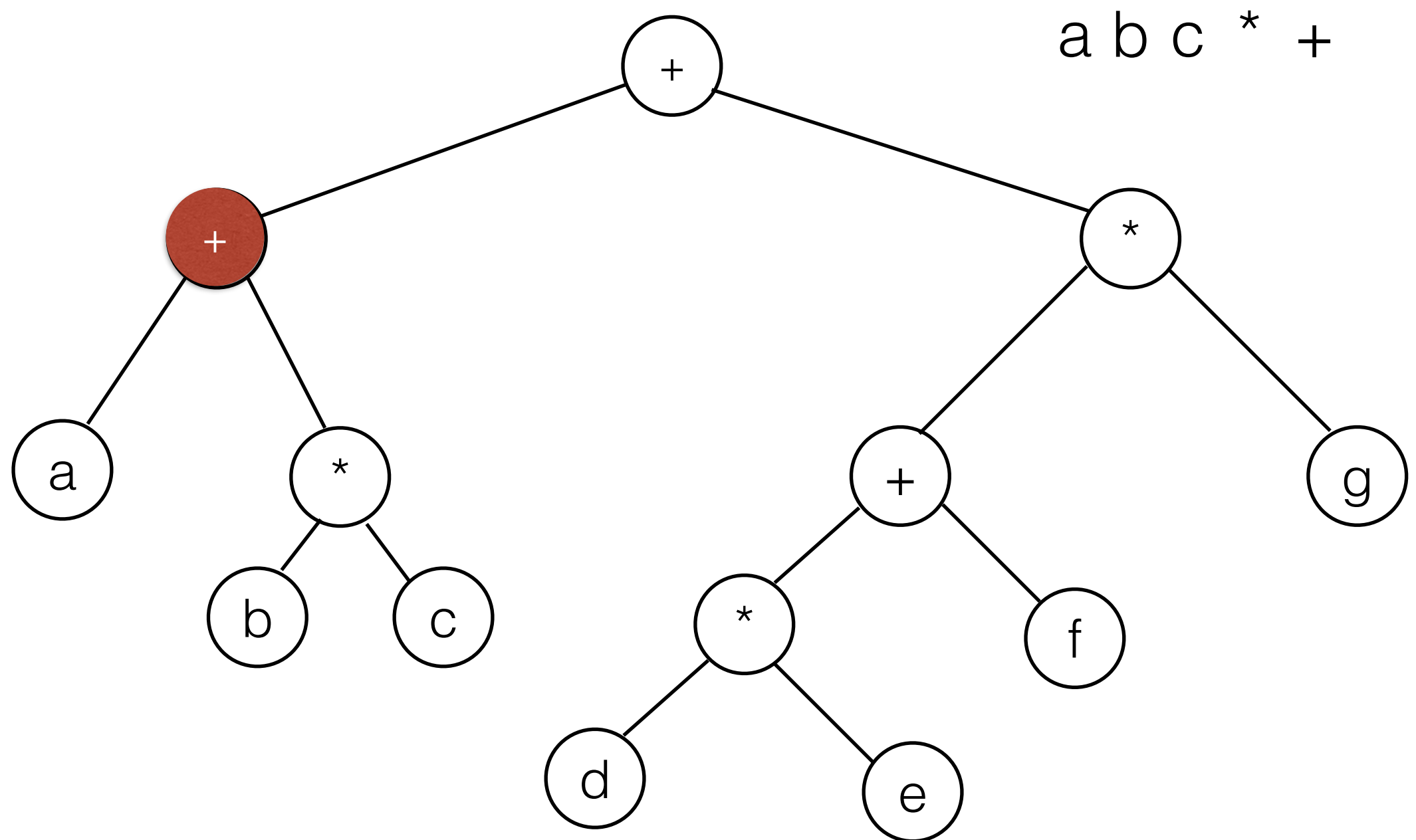
Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



Tree Traversals and Stacks

- Keep nodes that still need to be processed on a stack.



Tree Traversal using Recursion

- We often use recursion to traverse trees (making use of Java's method call stack implicitly).

```
public void printTree(int indent ) {  
    for (i=0;i<indent;i++)  
        System.out.print(" ");  
  
    System.out.println( data);           // data  
    if( left != null )  
        left.printTree(indent + 1);    // Left  
    if( right != null )  
        right.printTree(indent + 1);  // Right  
}
```

Bare-bones Implementation of a Binary Tree

- Public methods in `BinaryTree` usually call recursive helper methods.
 - Two options: implement in `BinaryTree` or in `BinaryNode`
- (sample code)

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

5 27 2 3 * / +

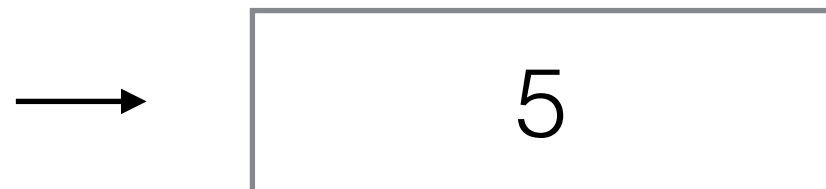
- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

push(5)

5 27 2 3 * / +



- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

push(27)

5 27 2 3 * / +



- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

push(2)

5 27 2 3 * / +



2
27
5

- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

5 27 2 3 * / +

push(3)



3
2
27
5

- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

pop() -> 3
pop() -> 2
push(2*3)

5 27 2 3 * / +



- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

pop() -> 6
pop() -> 27
push(27/6)

5 27 2 3 * / +



- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

Recall: Evaluating Postfix Expressions

$$5 + 27 / (2 * 3)$$

pop() -> 4.5
pop() -> 5
push(5 + 4.5)

5 27 2 3 * / +

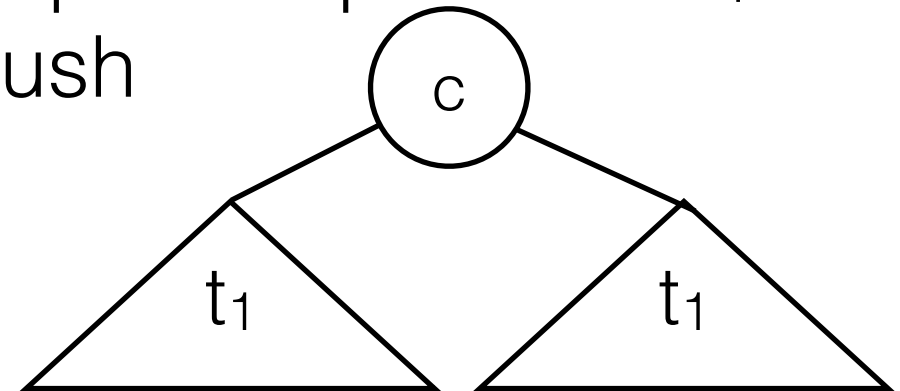
→ 9

- for c in input
 - if c is an operand, push it
 - if c is an operator x:
 - pop the top 2 operands a_1 and a_2
 - push $a_3 = a_2 \times a_1$
- pop the result.

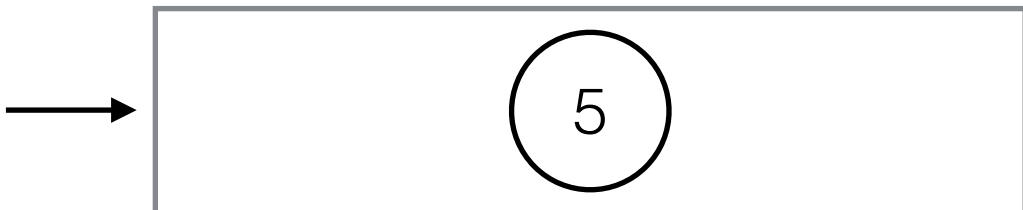
Constructing Expression Trees using a Stack

5 27 2 3 * / +

- for c in input
 - if c is an operand, push a tree
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push

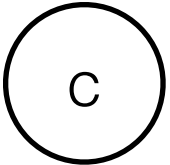
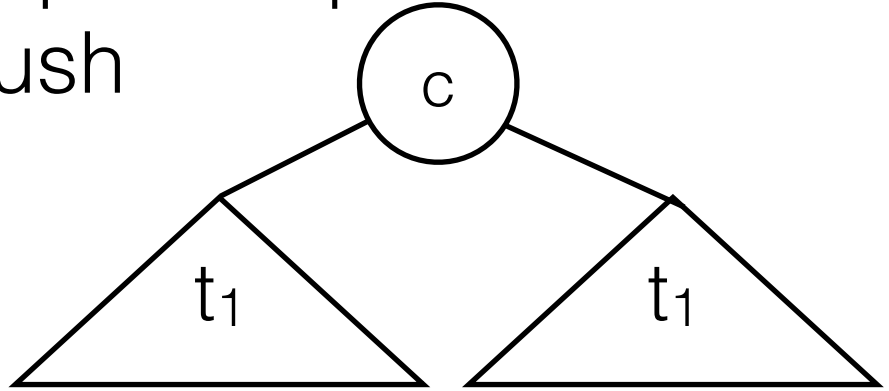


- pop the result.



Constructing Expression Trees using a Stack

5 **27** 2 3 * / +

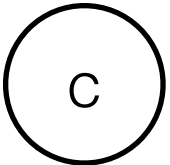
- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push 

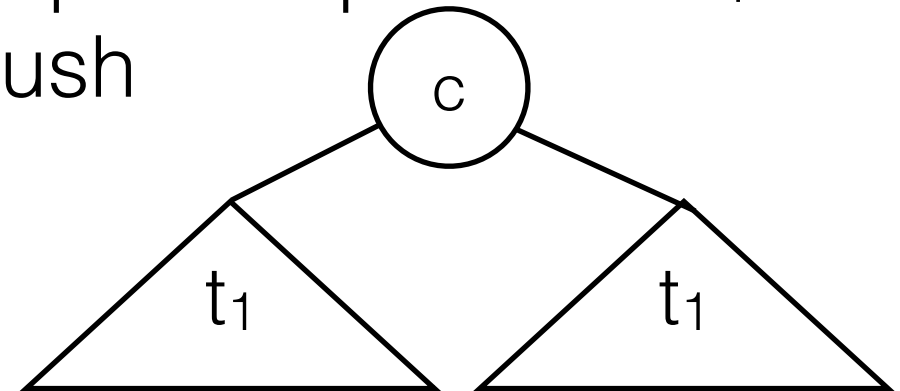
- pop the result.



Constructing Expression Trees using a Stack

5 27 2 3 * / +

- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push

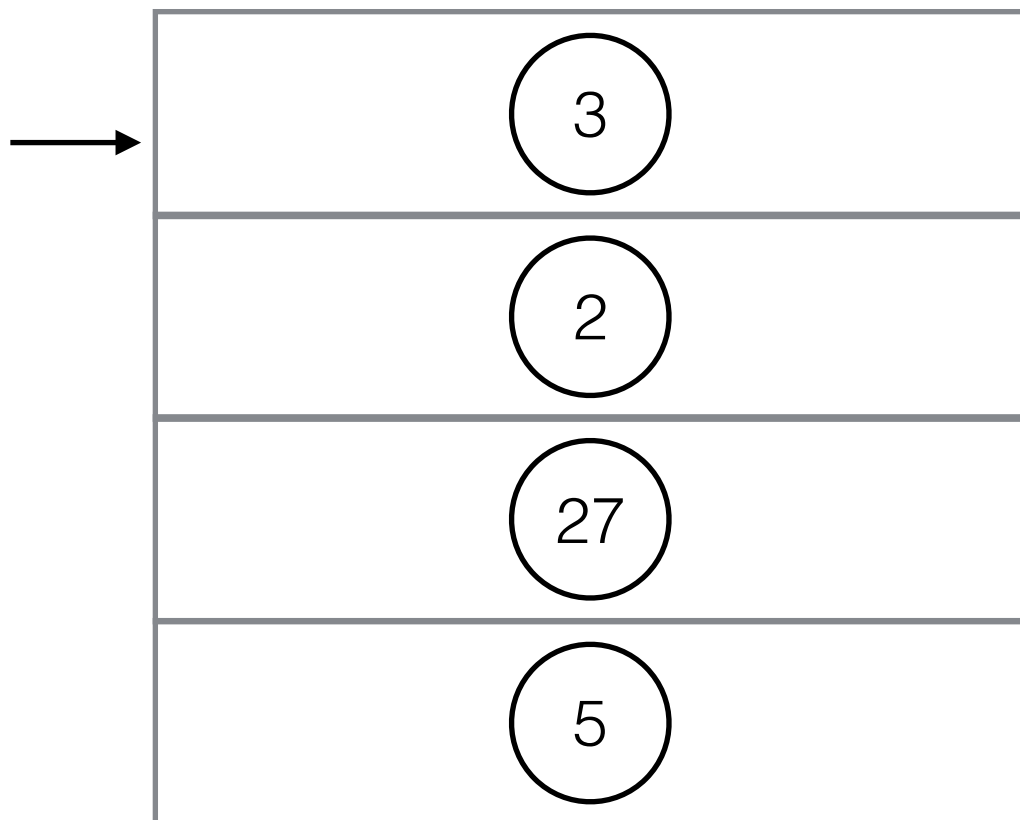


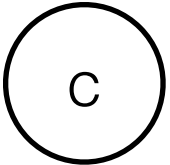
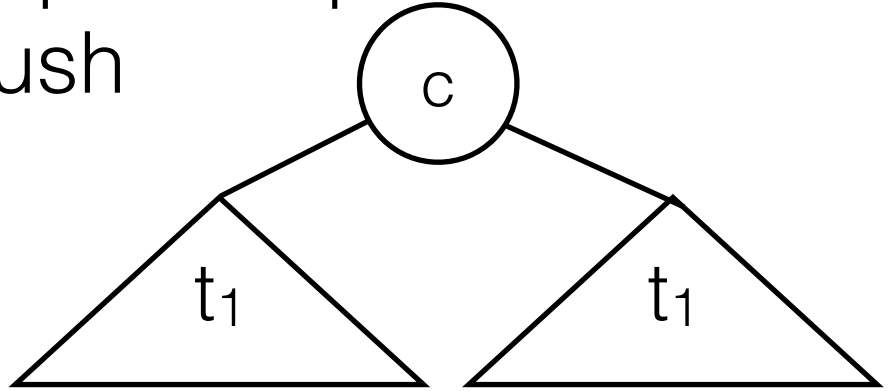
- pop the result.



Constructing Expression Trees using a Stack

5 27 2 **3** * / +

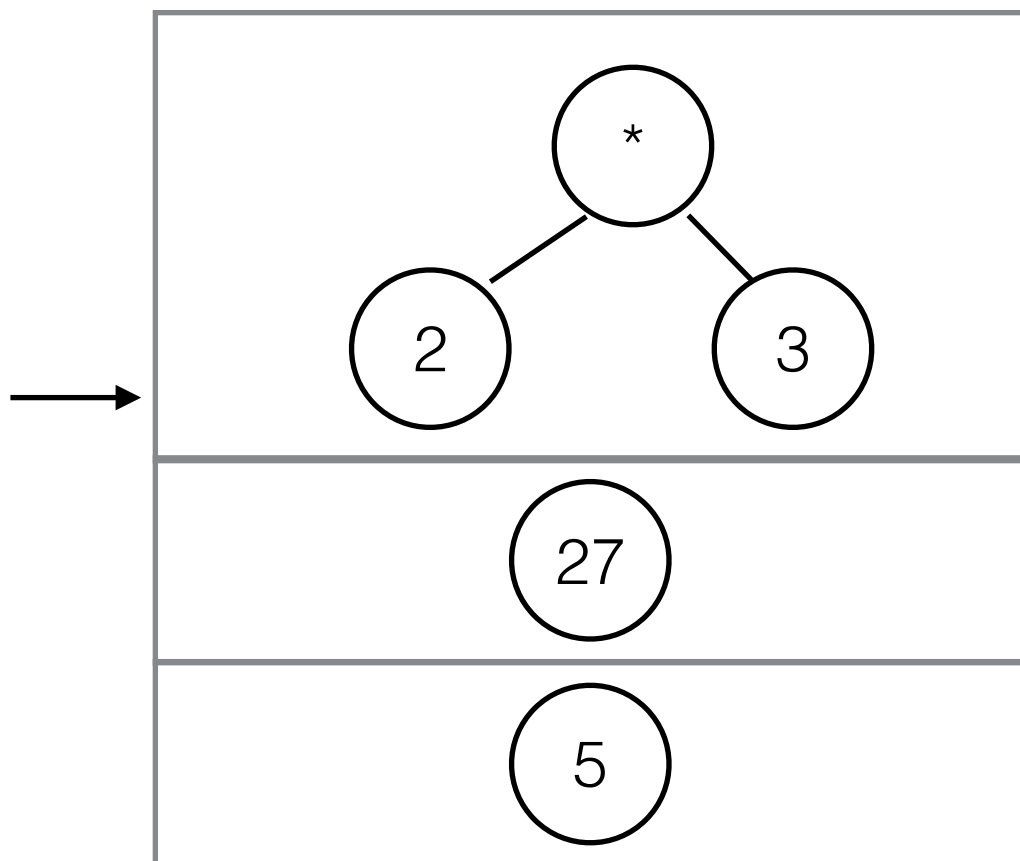


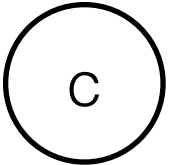
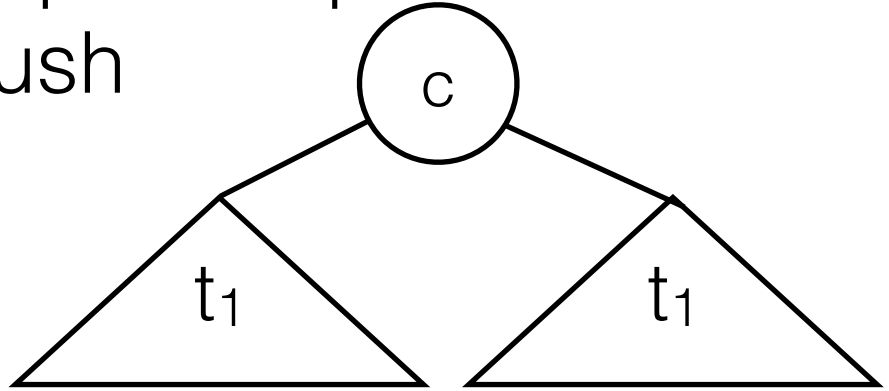
- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push 

- pop the result.

Constructing Expression Trees using a Stack

5 27 2 3 * / +

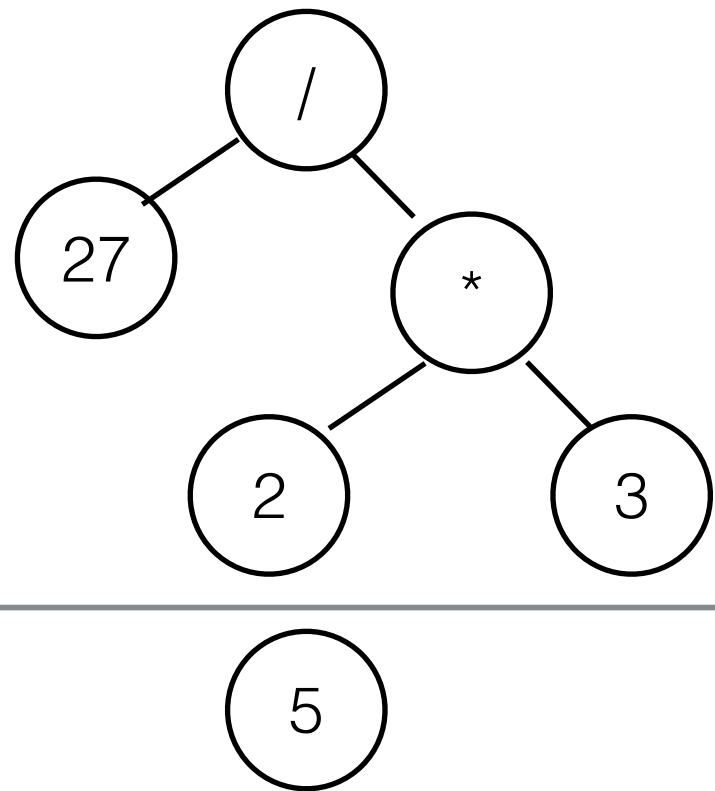


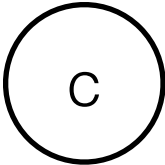
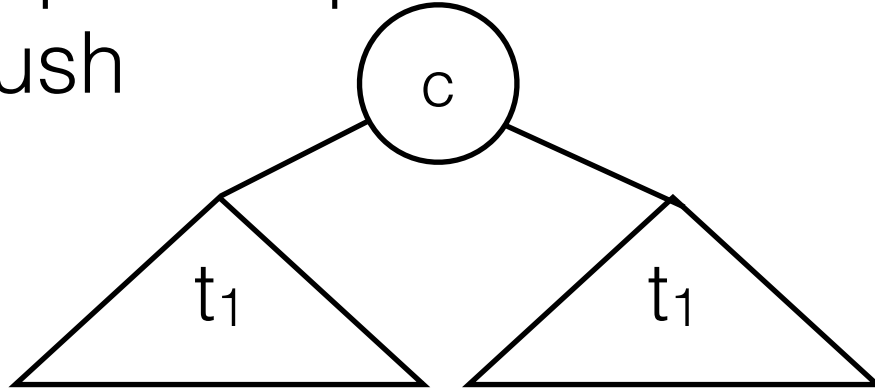
- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push 

- pop the result.

Constructing Expression Trees using a Stack

5 27 2 3 * / +

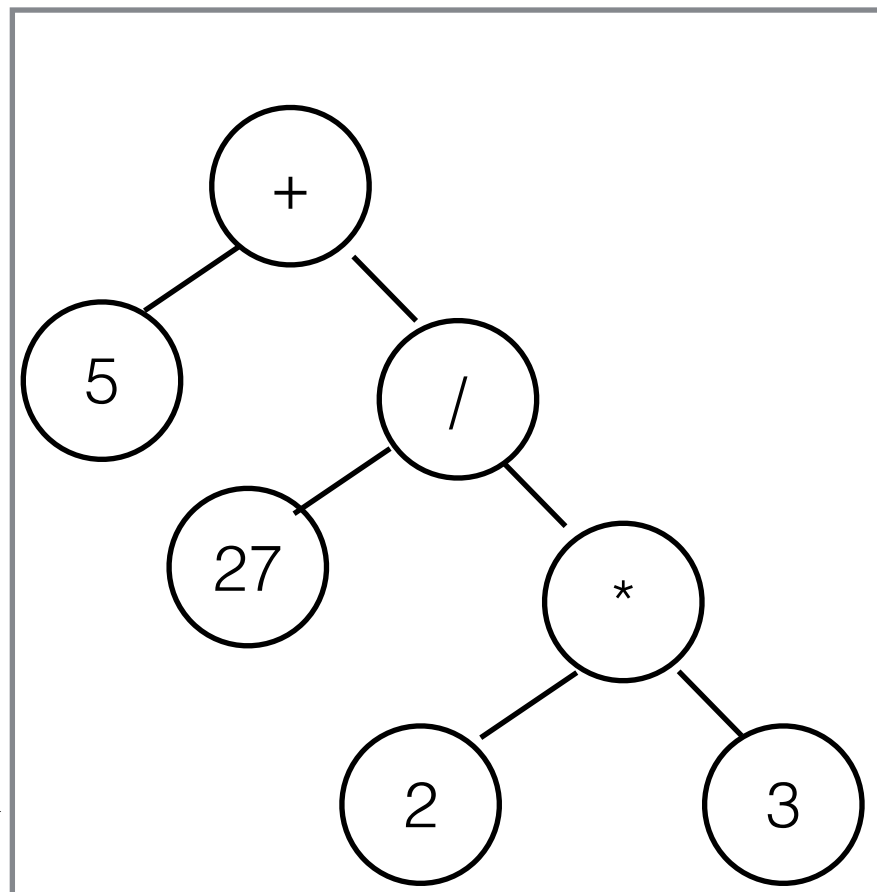


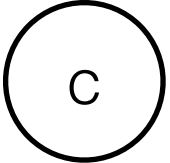
- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push 

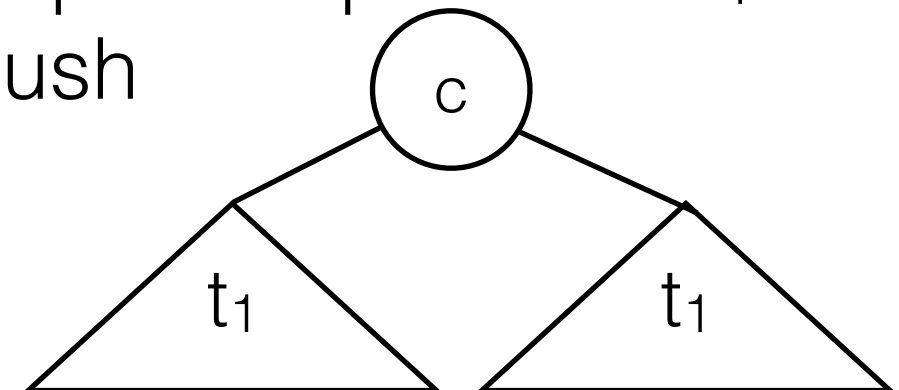
- pop the result.

Constructing Expression Trees using a Stack

5 27 2 3 * / +



- for c in input
 - if c is an operand, push a tree 
 - if c is an operator:
 - pop the top 2 trees t_1 and t_2
 - push



- pop the result.