Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 5.

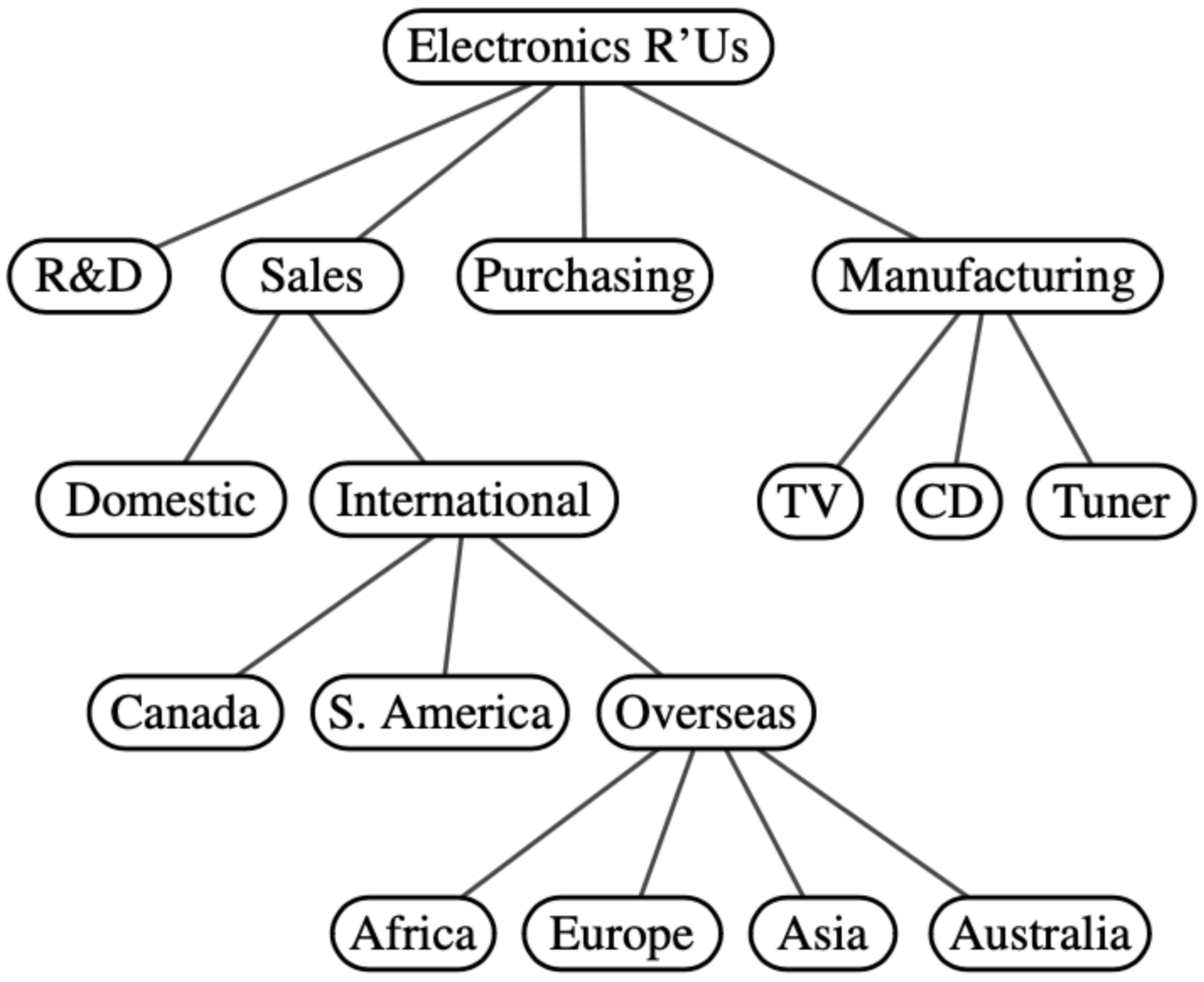
Review of Basic Data Structures - 3

M. Oguzhan Kulekci

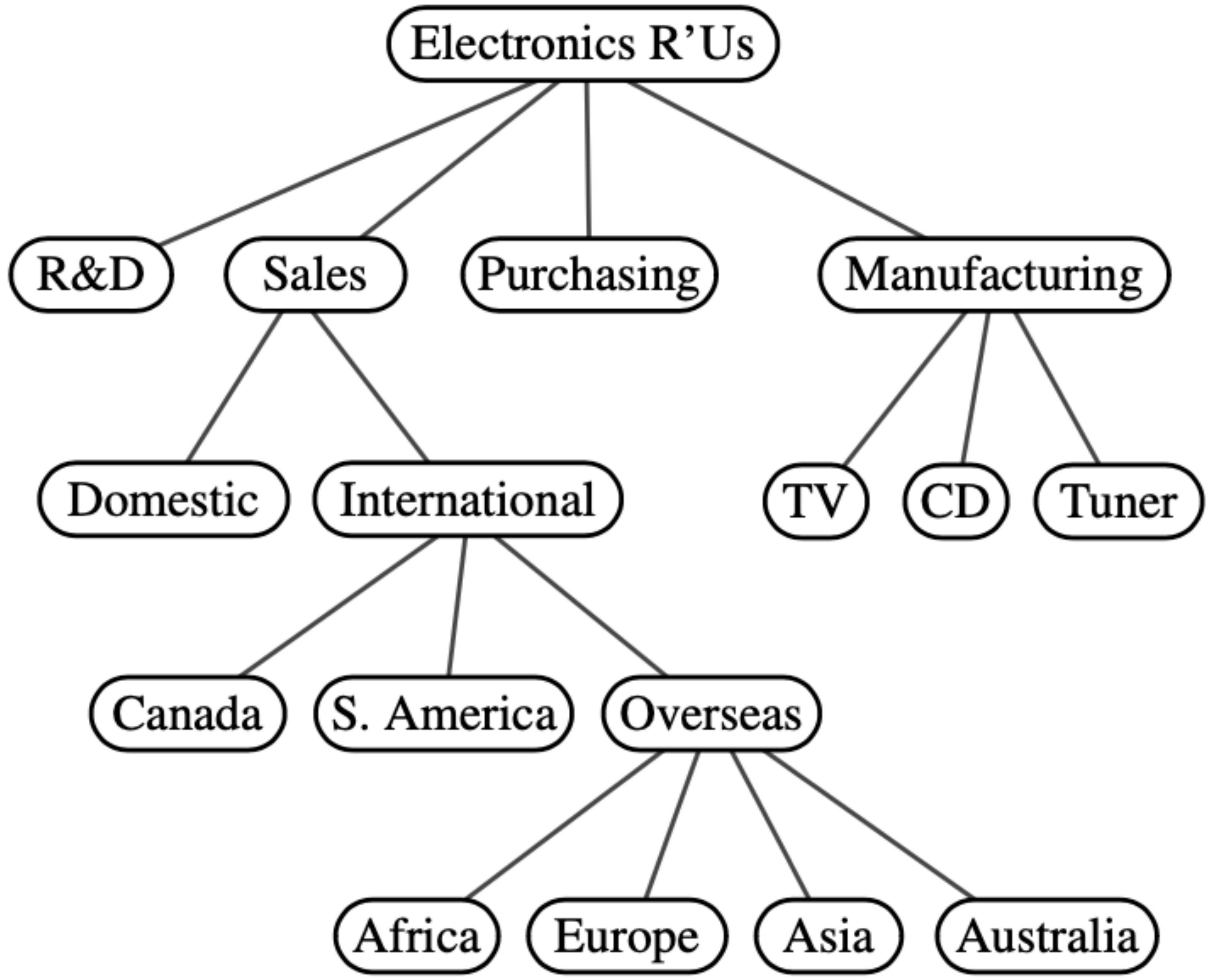
- Tree data structure
- Tree-traversals, pre-, post- in-order and breadth-first

Linear vs. Hierarchical Data Structures

- Array, linked-list, stack, queue are linear data structures, i.e., one-dimensional properties
- Trees are two-dimensional?, thus, hierarchical
- Graphs are also like trees with some differences



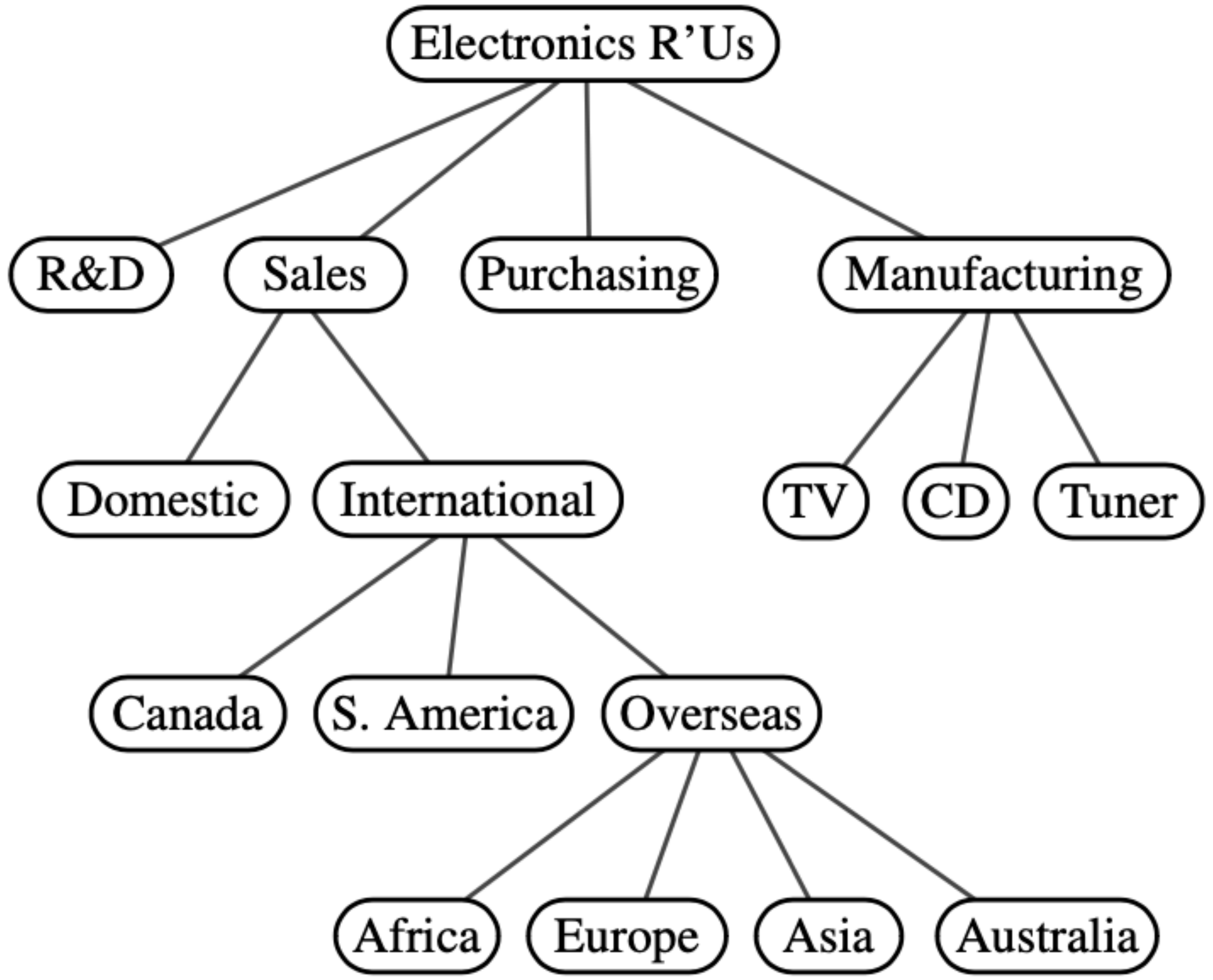
Tree Data Structure



A tree T is as a set of nodes storing elements such that the nodes have a **parent-child** relationship with the following properties:

- If T is nonempty, it has a special node, called **the root of T , that has no parent**.
- Each node v of T different from the root has **a unique parent node** w; every node with parent w is a **child** of w.

Tree Data Structure



- Siblings, internal node, external or leaf node
- Ancestor, descendant
- Subtree of T rooted at a node v
- Edge, path
- Ordered tree, n-ary tree

Tree Data Structure

T.root(): Return the position of the root of tree T, or None if T is empty.

T.is_root(p): Return True if position p is the root of Tree T.

p.element(): Return the element stored at position p.

T.parent(p): Return the position of the parent of position p, or None if p is the root of T.

T.num_children(p): Return the number of children of position p.

T.children(p): Generate an iteration of the children of position p.

T.is_leaf(p): Return True if position p does not have any children.

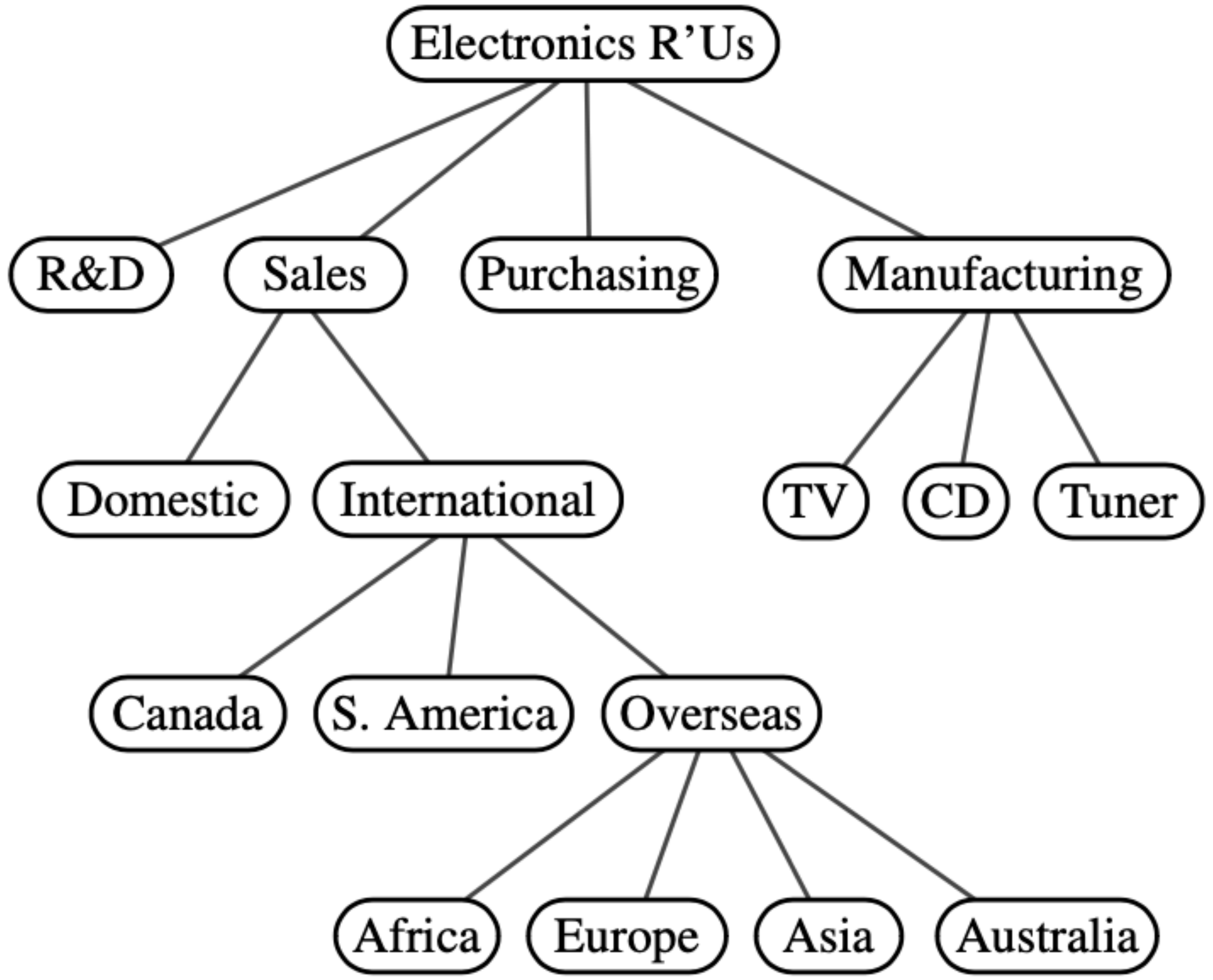
len(T): Return the number of positions (and hence elements) that are contained in tree T.

T.is_empty(): Return True if tree T does not contain any positions.

T.positions(): Generate an iteration of all *positions* of tree T.

iter(T): Generate an iteration of all *elements* stored within tree T.

Tree Data Structure

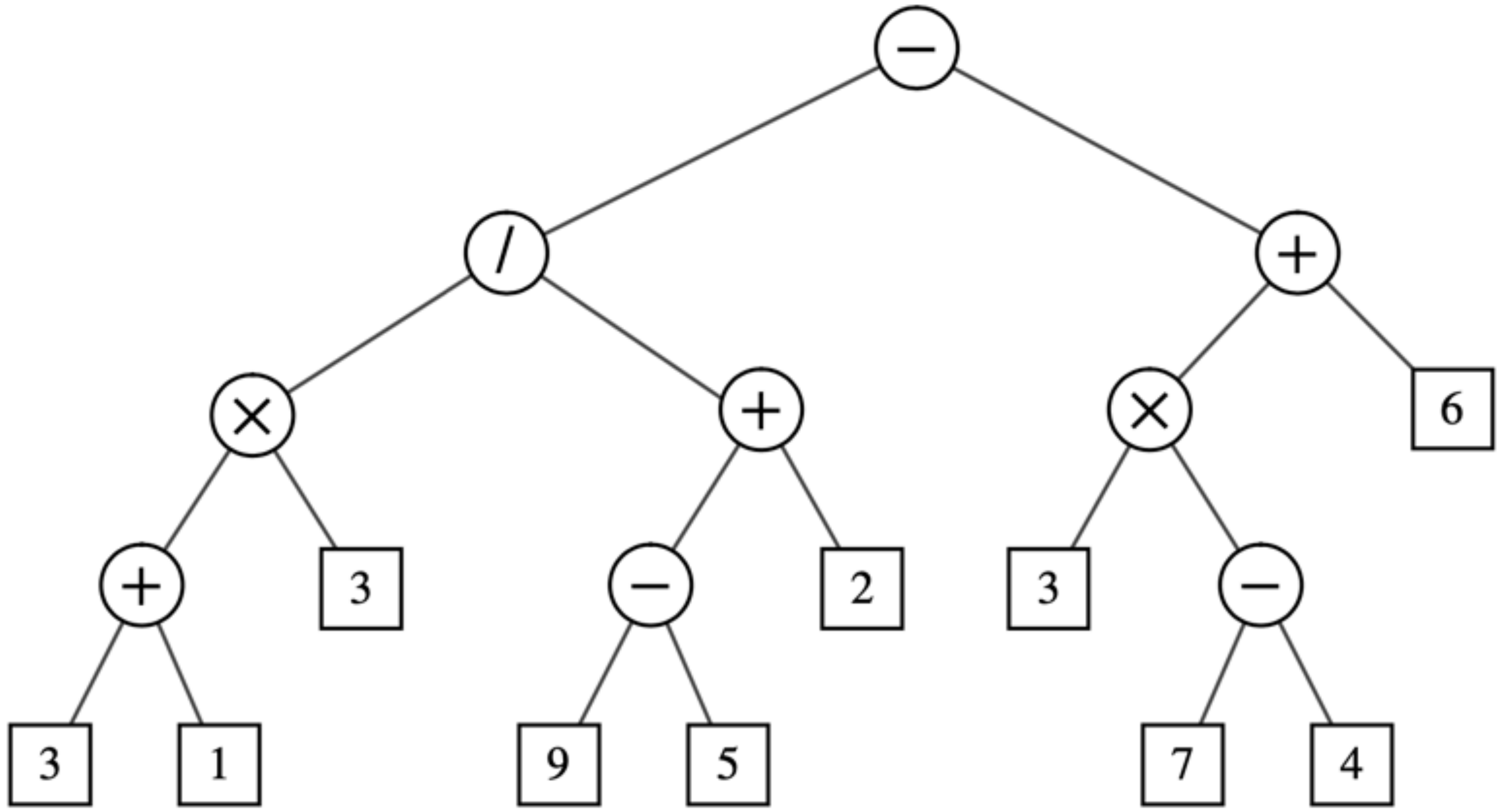


- **Depth of a node:** Number of ancestors to the root (excluding itself' e.g., root has depth 0)
- **Height of a node:** Number of descendants on the longest path to a leaf (excluding itself, e.g. leaves have height 0)

The height of a nonempty tree T is equal to the maximum of the depths of its leaf positions.

Binary Tree

- Each node has at most 2 children
- Each node (other than root) is either left or right child of another node.
- Proper binary tree: Each node has either 0 or 2 children
- There are many properties of binary trees, please refer to the textbooks…



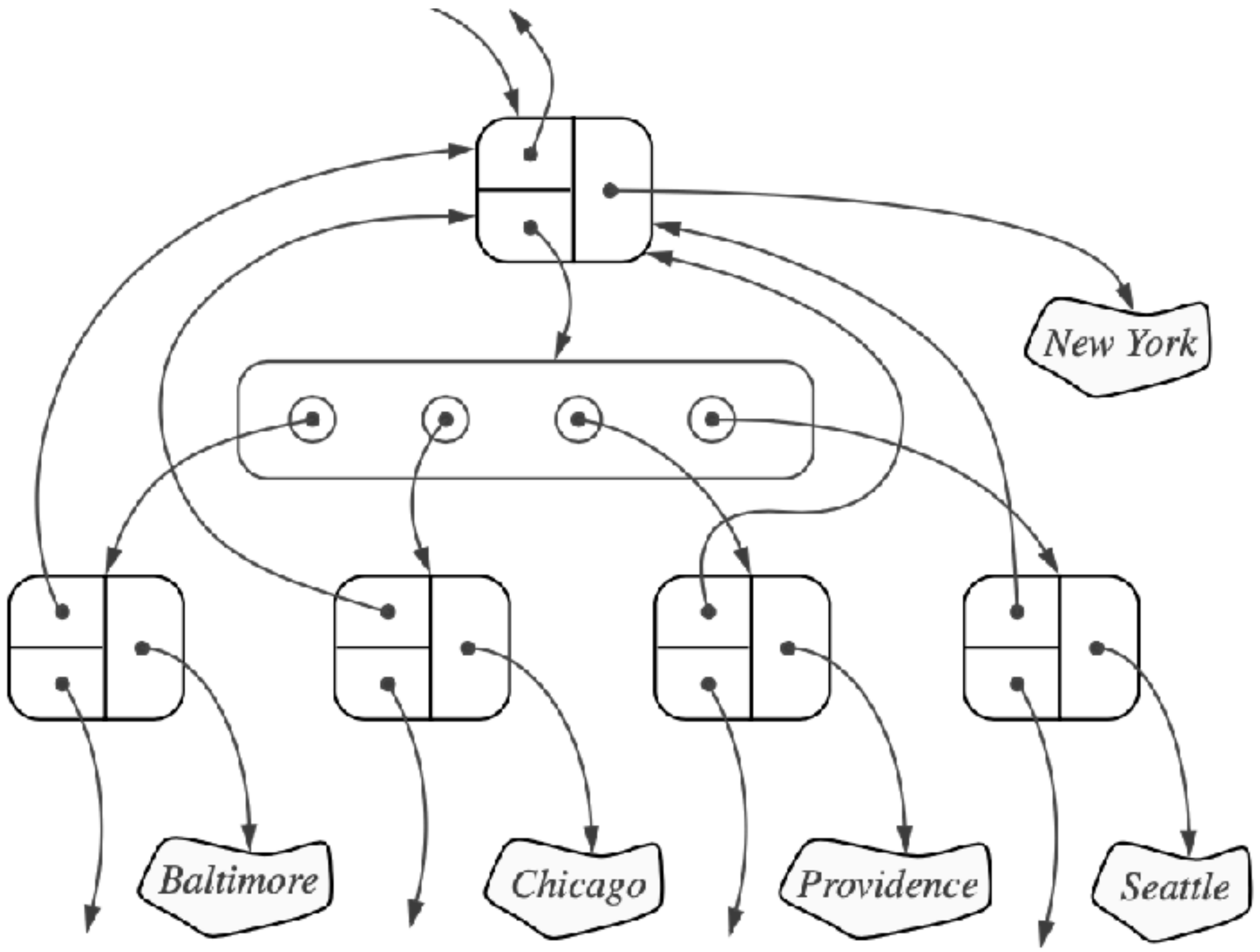
T.left(p): Return the position that represents the left child of p, or None if p has no left child.

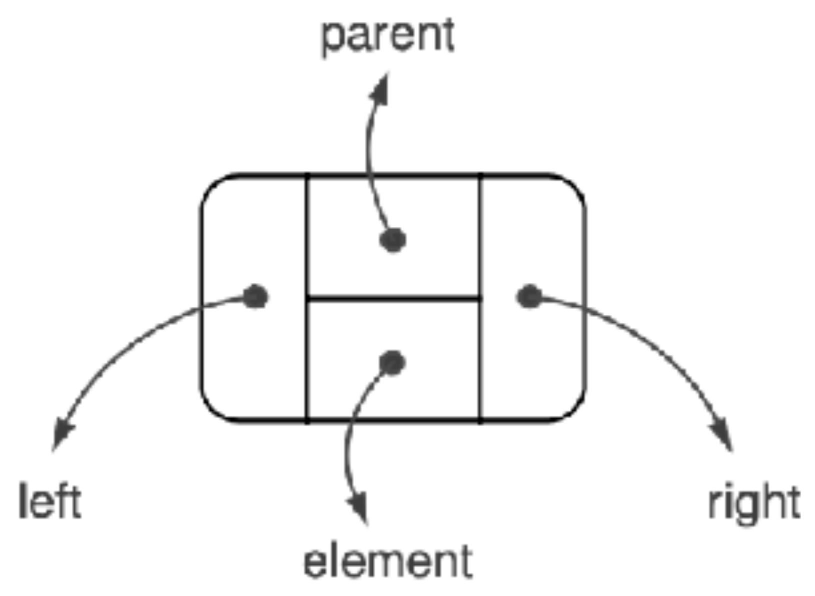
T.right(p): Return the position that represents the right child of p, or None if p has no right child.

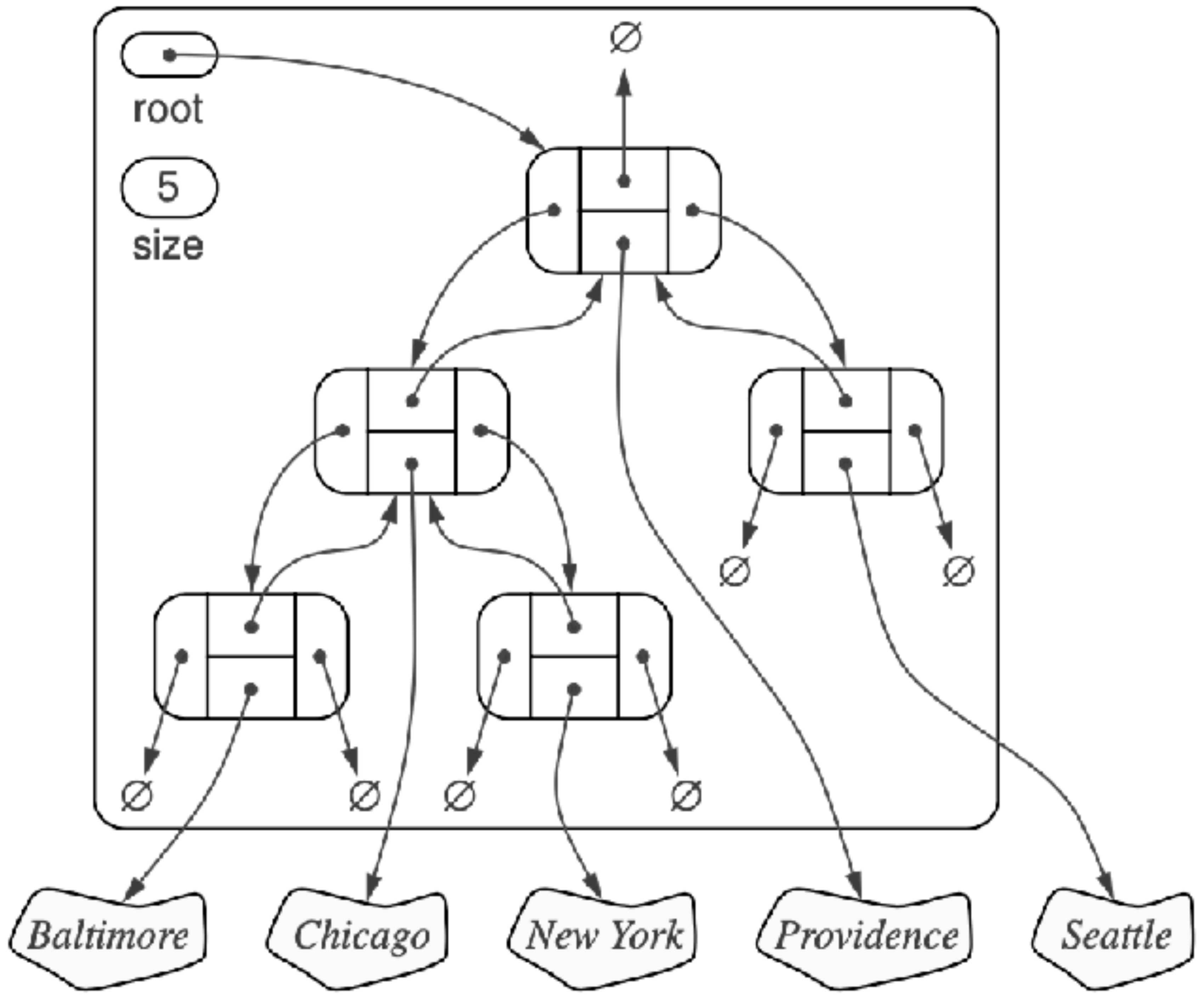
T.sibling(p): Return the position that represents the sibling of p, or None if p has no sibling.

Implementing Trees

Linked-List based





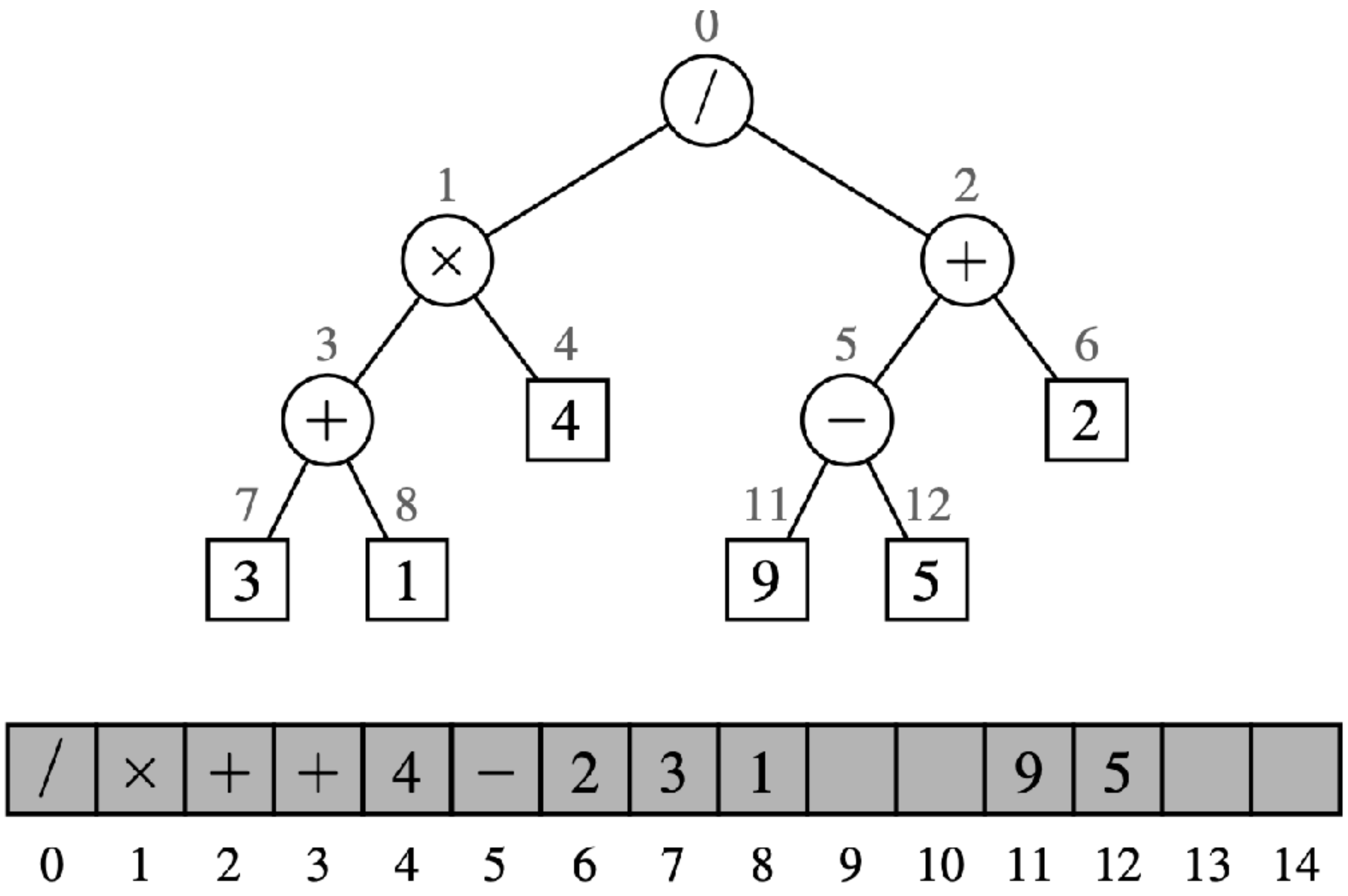


Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p+1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

Implementing Trees

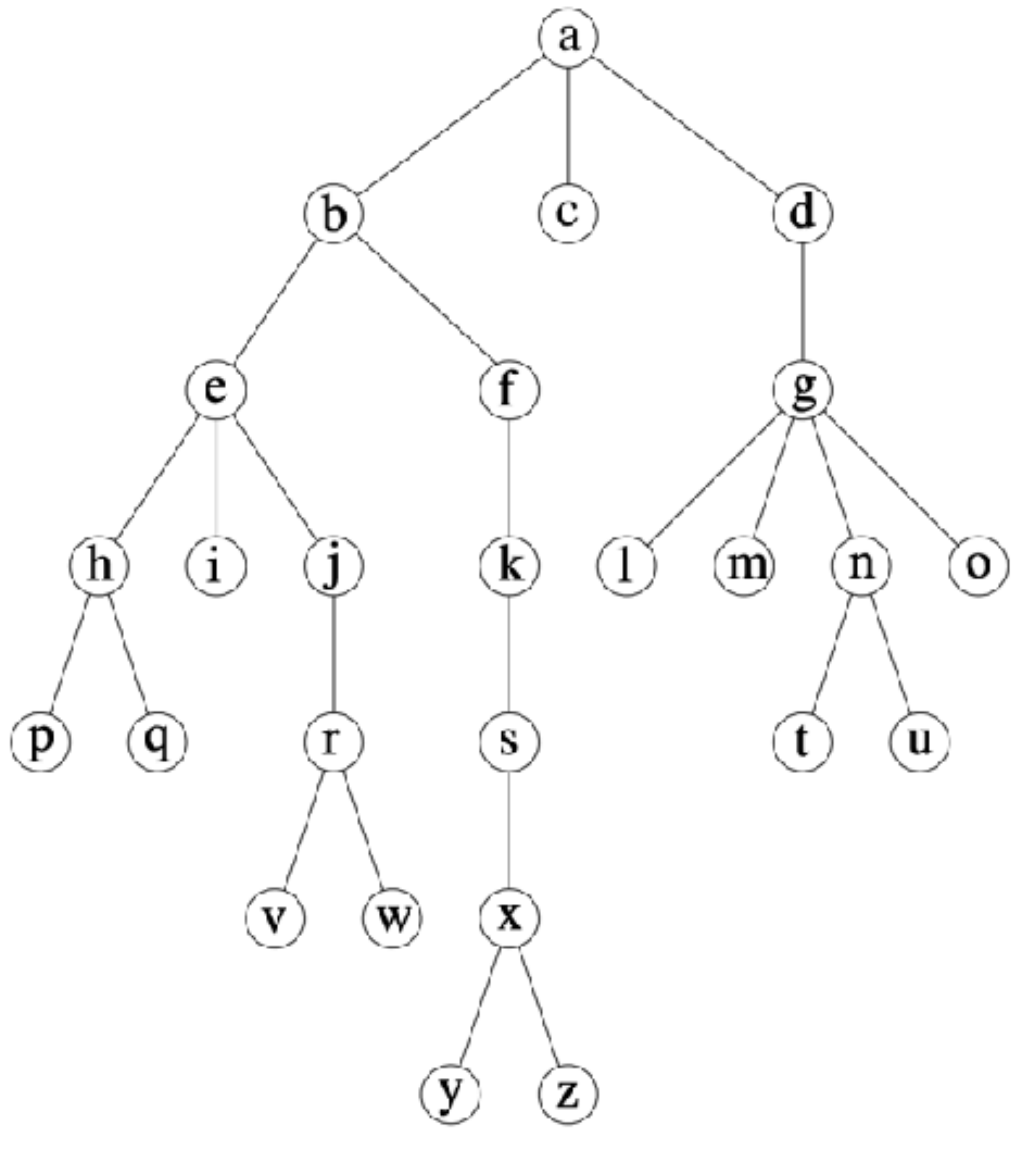
Array based (binary tree)

- Root at position $p = 0$.
- Left child of p is $2 \cdot p + 1$
- Right child of p is $2 \cdot p + 2$



Implementing Trees

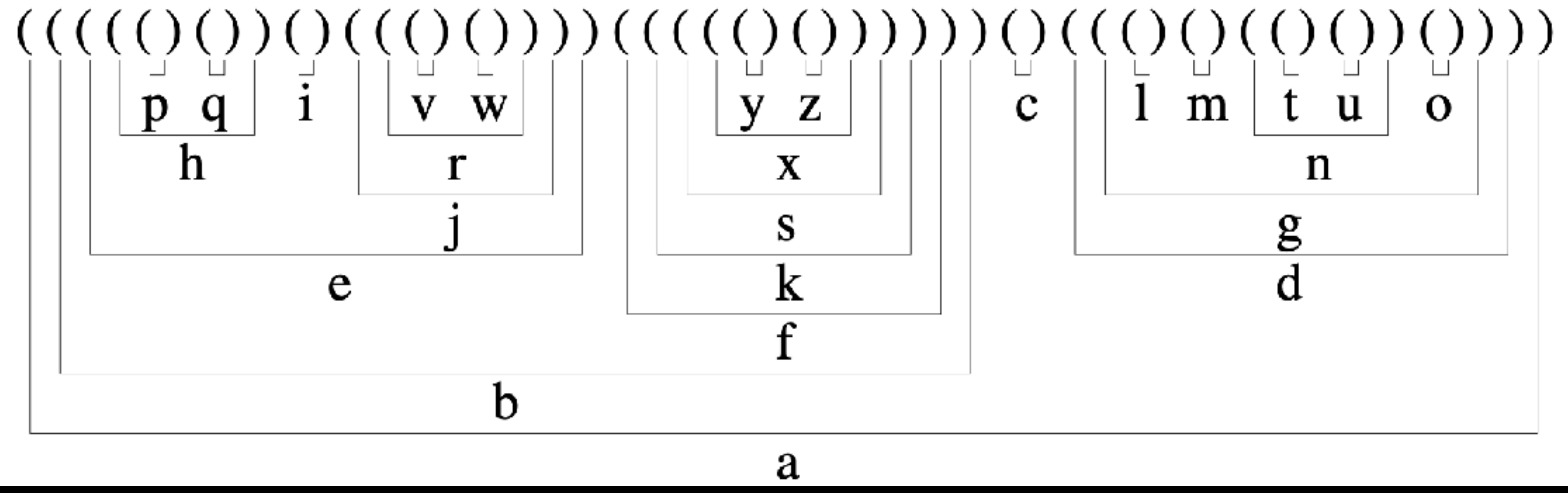
Advanced representations without links, or sparse arrays is possible
You can refer to http://erikdemaine.org/papers/MaryTrees_Algorithmica/paper.pdf for a review of those sophisticated representations.



LOUD(level-order unary degree) represention by Jacobson'89

1110110010111010111101100101000110000110100000011000

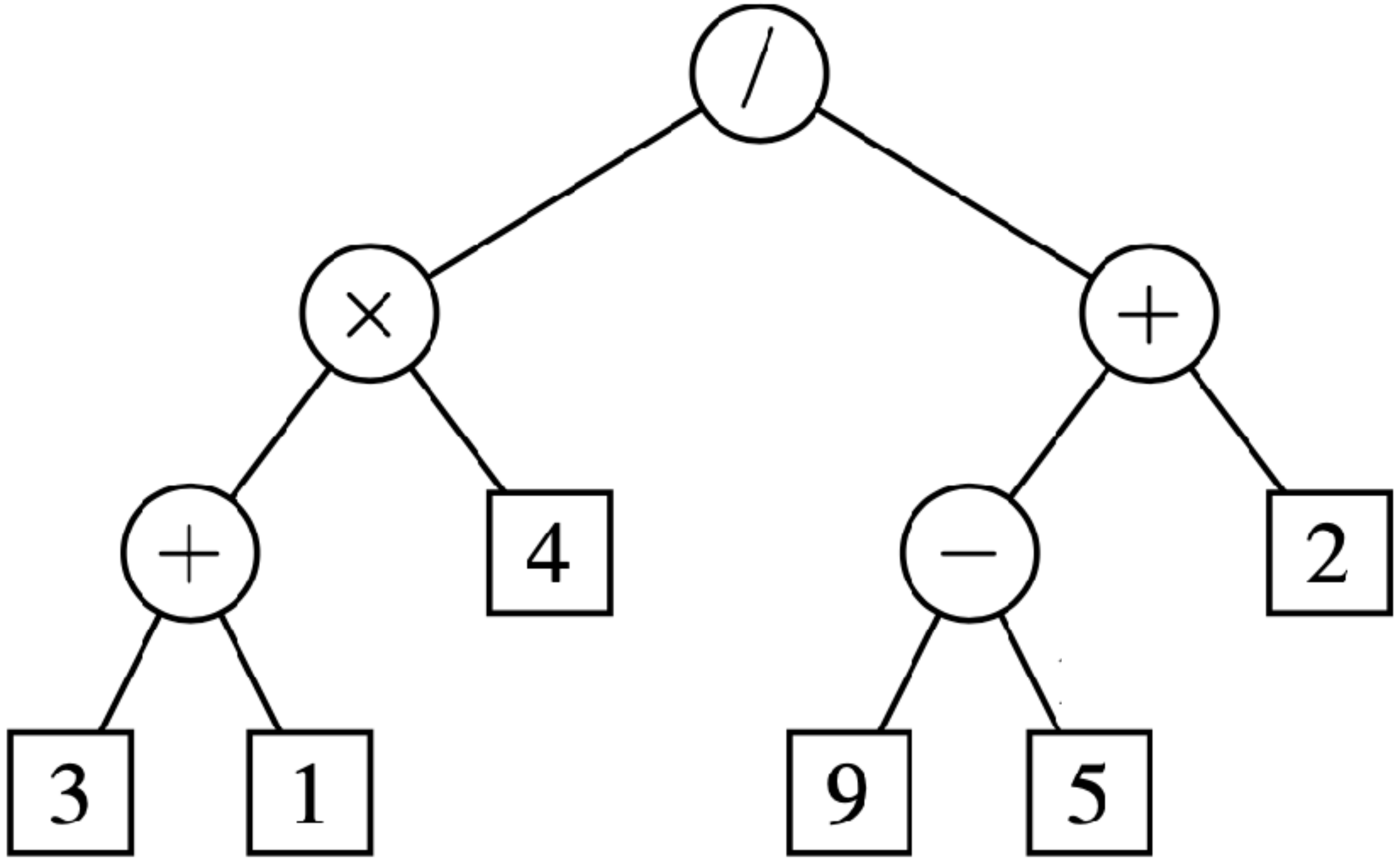
Munro and Raman's (2001) Balanced Parantheses Representation



Tree Traversals

Breadth-First Traversal

```
Algorithm breadthfirst(T):
  Initialize queue Q to contain T.root()
  while Q not empty do
    p = Q.dequeue()                       {p is the oldest entry in the queue}
    perform the "visit" action for position p
    for each child c in T.children(p) do
      Q.enqueue(c)       {add p's children to the end of the queue for later visits}
```

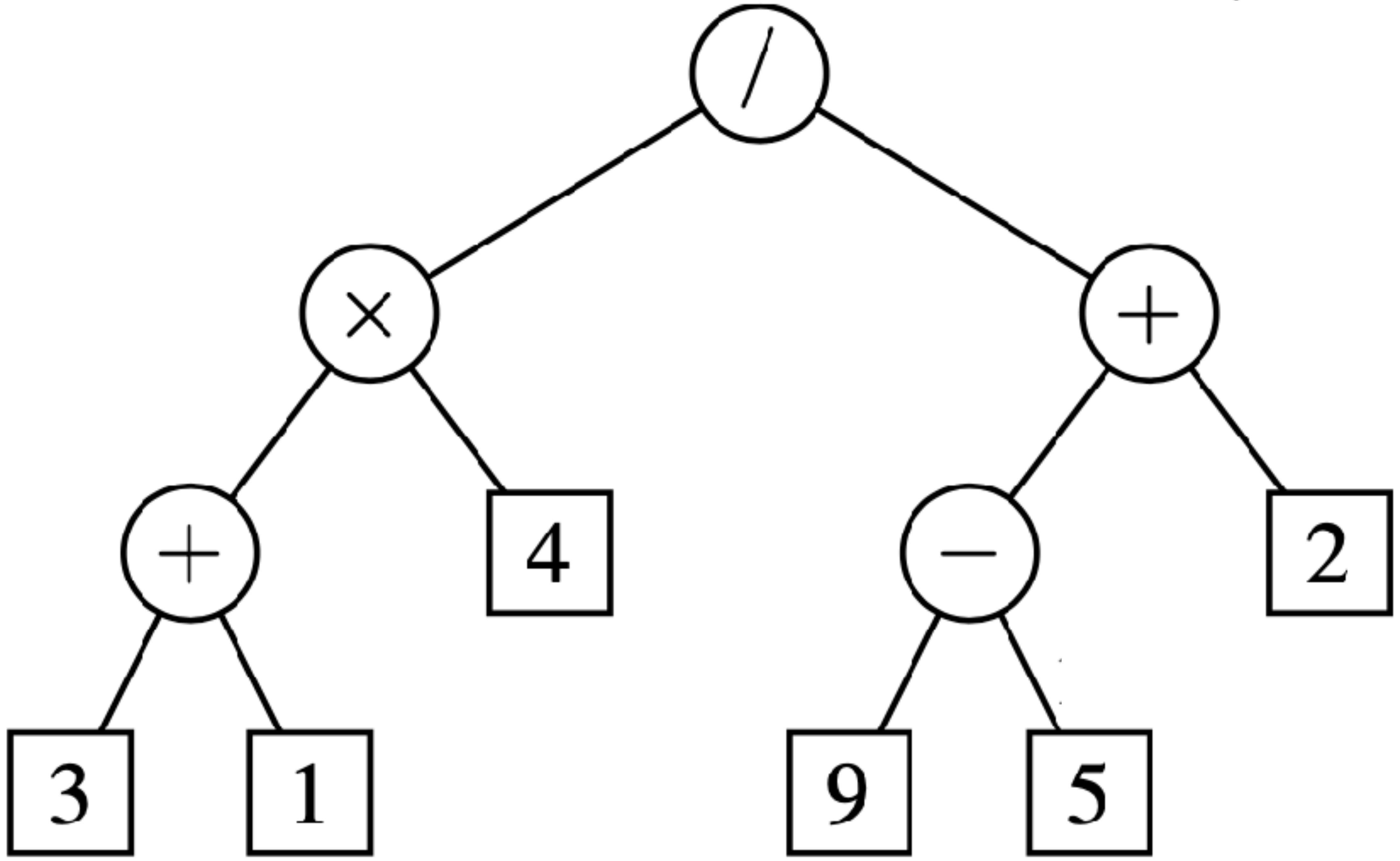


	/
/	x , +
x	+ , + , 4
+	+ , 4 , - , 2
+	4 , - , 2 , 3 , 1
4	- , 2 , 3 , 1 , 9 , 5
-	2 , 3 , 1 , 9 , 5
2	3 , 1 , 9 , 5
3	1 , 9 , 5
1	9 , 5
9	5
5	

Tree Traversals

Preorder Traversal

Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) **do**
    preorder(T, c)   {recursively traverse the subtree rooted at c}

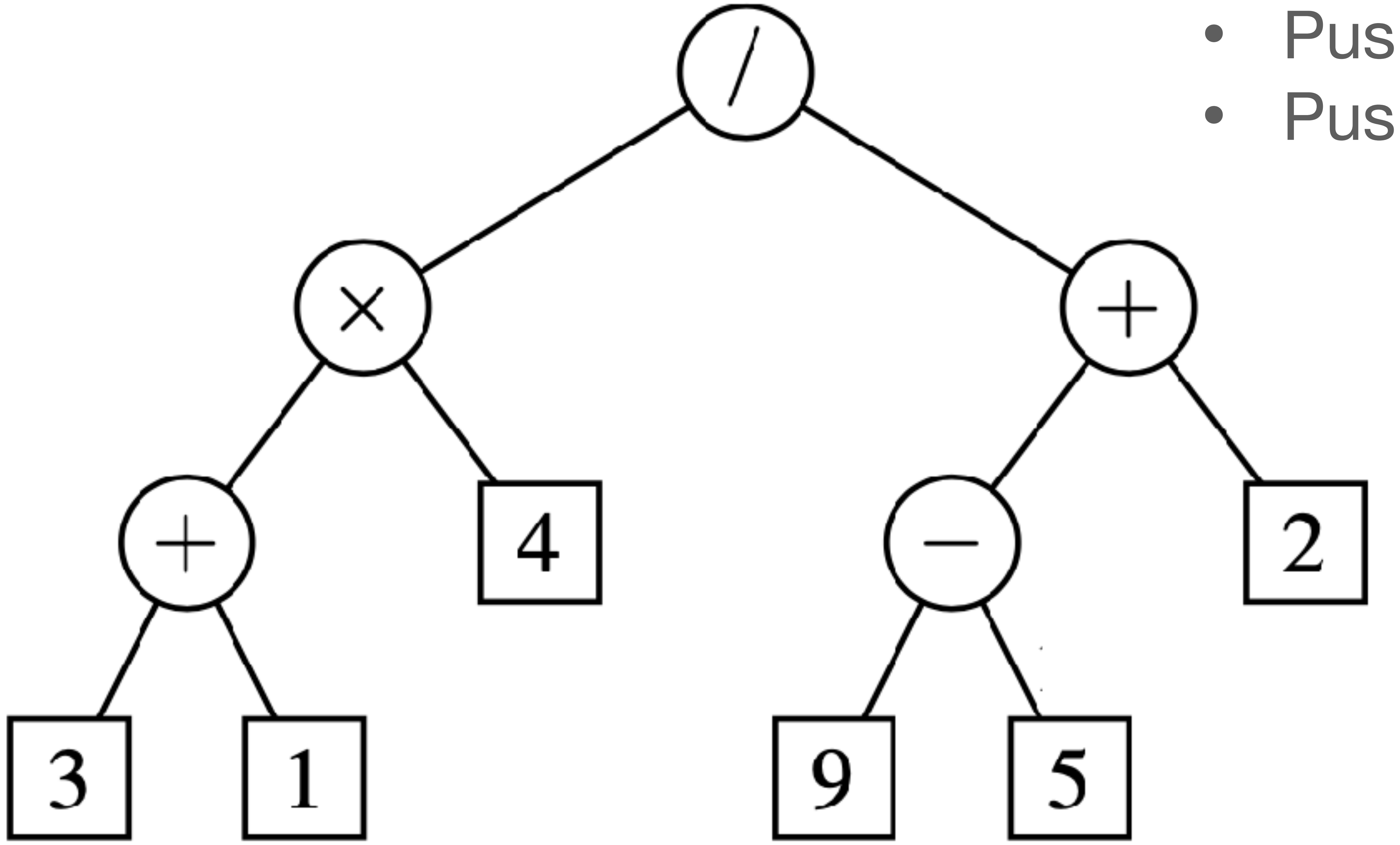


/	x	+	3	1	4	+	-	9	5	2

How can you evaluate such a given prefix expression ?

Tree Traversals

Iterative Preorder Traversal with a stack

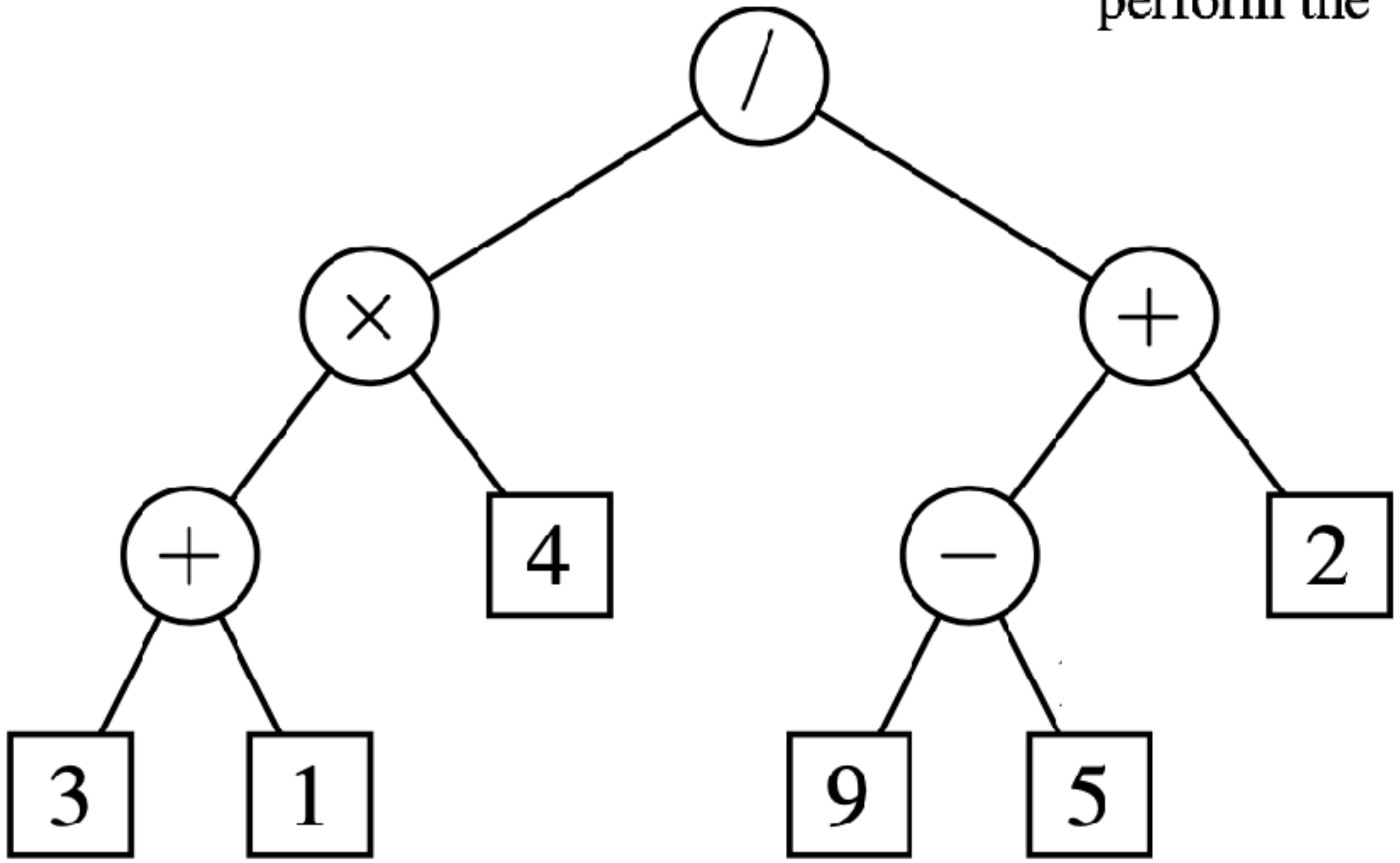


- Push root to the stack
- While stack is not empty
  - Pop from the stack and print it
  - Push **right** child of the popped item into the stack
  - Push **left** child of the popped item into the stack

Tree Traversals

Postorder Traversal

Algorithm postorder(T, p):
  for each child c in T.children(p) **do**
    postorder(T, c)   {recursively traverse the subtree rooted at c}
  perform the "visit" action for position p



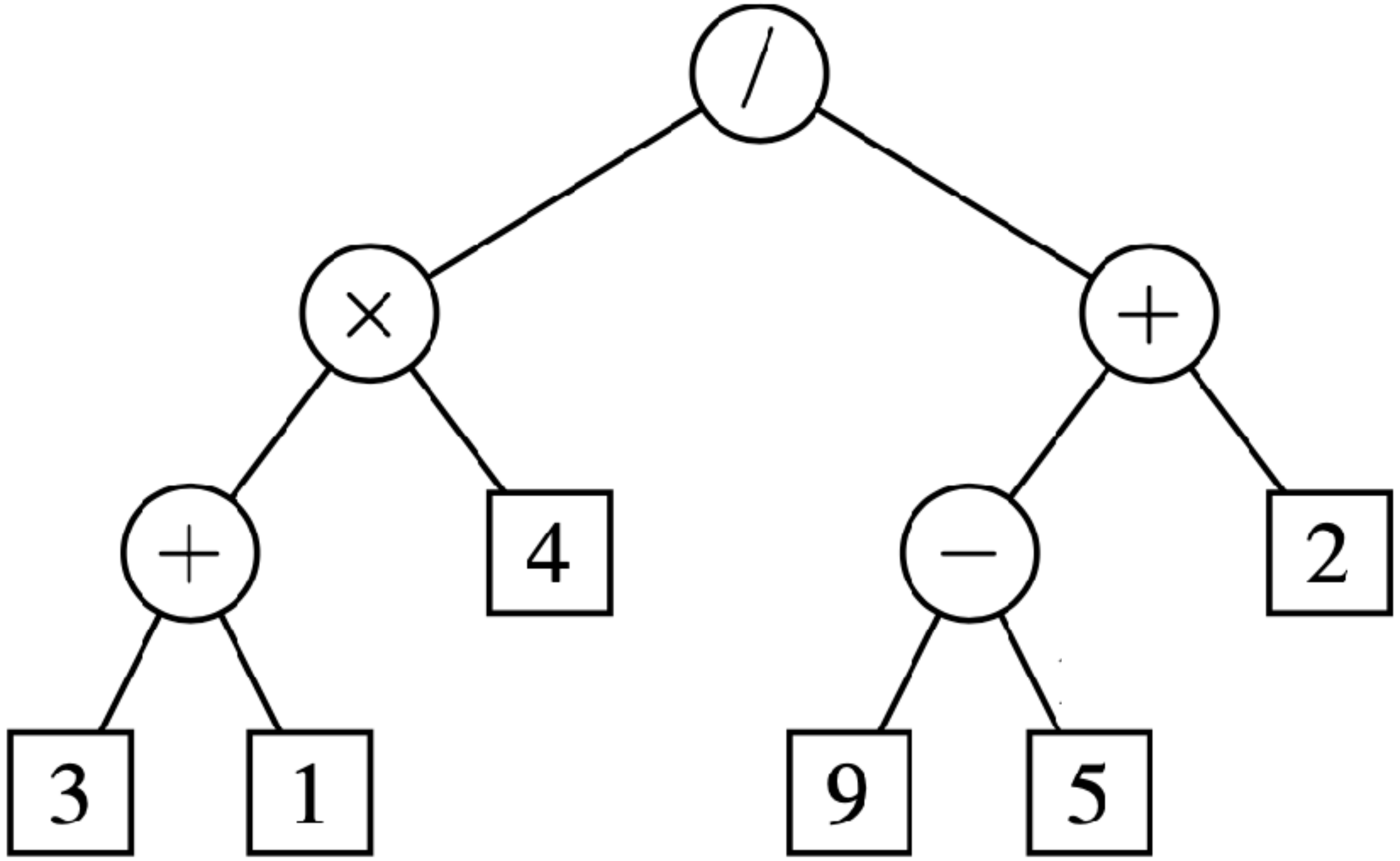
3 1 + 4 x 9 5 - 2 + /

How can you evaluate such a given postfix expression ?

What is the difference in between using a prefix or postfix expression ?

Tree Traversals

Iterative Postorder Traversal

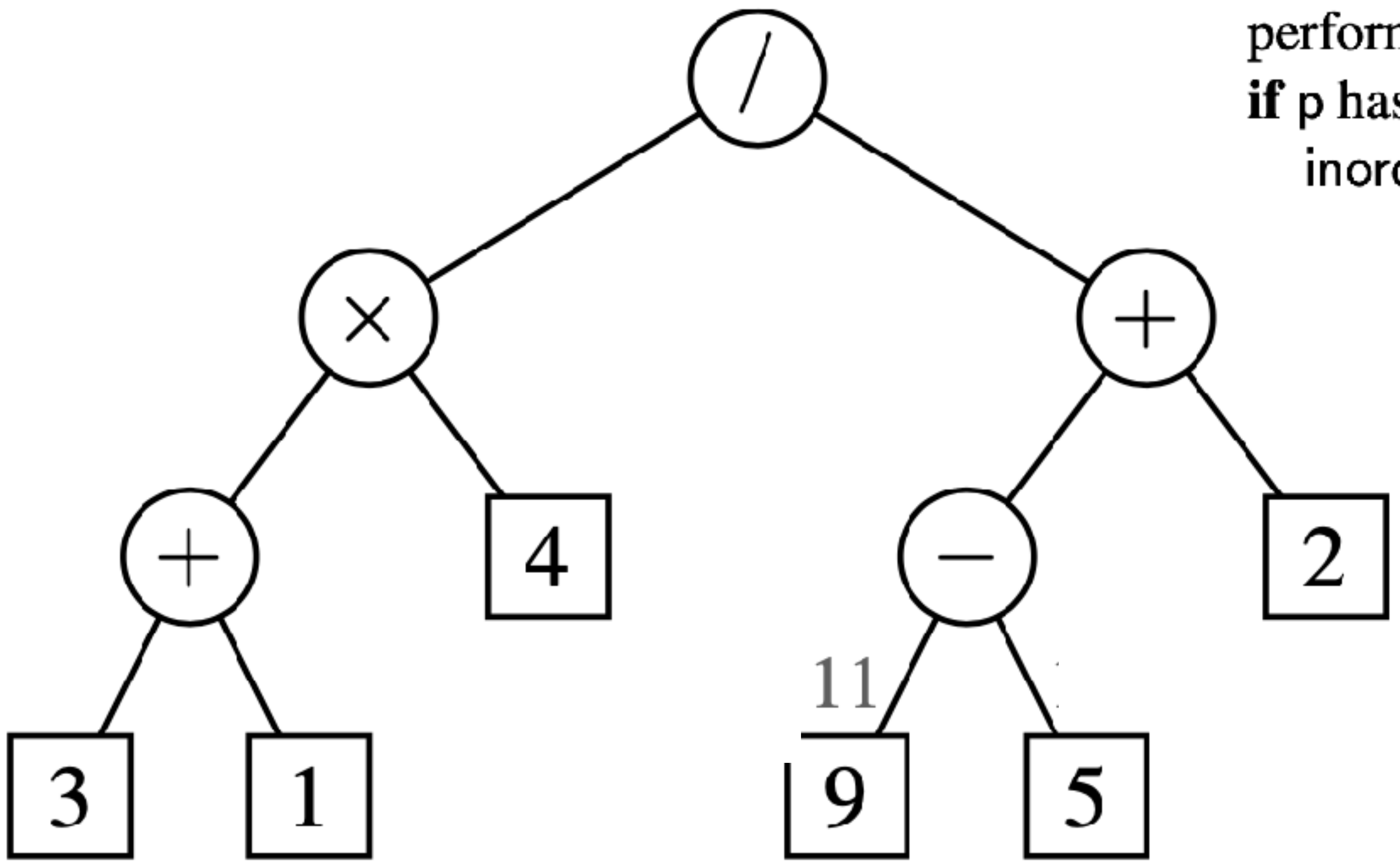


- Push root to the FIRST stack
- While FIRST stack is not empty
  - Pop from the FIRST stack and push it into SECOND stack
  - Push left child of the popped item into the FIRST stack
  - Push right child of the popped item into the FIRST stack
- Pop everything from the SECOND stack

We used two stacks. It is also possible to make it with one stack !

Tree Traversals

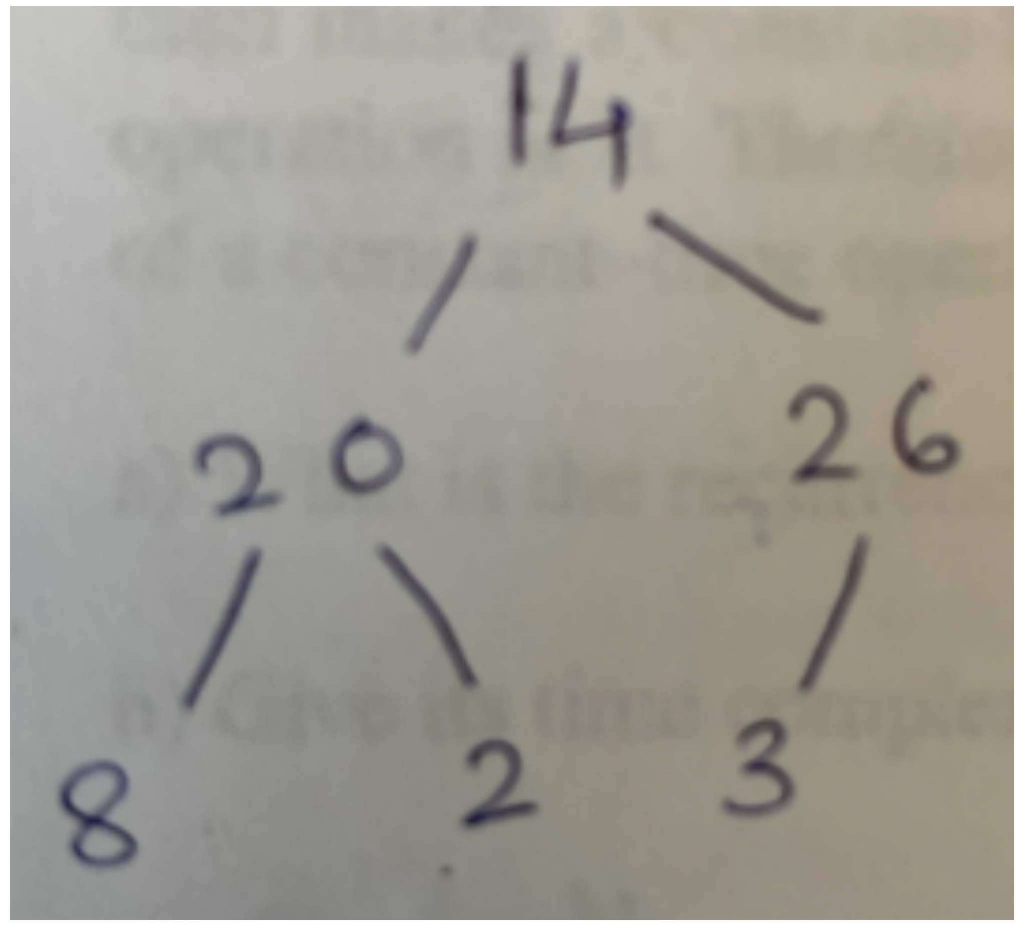
Inorder Traversal (only on binary trees)



Algorithm inorder(p):
  if p has a left child lc **then**
    inorder(lc) {recursively traverse the left subtree of p}
  perform the "visit" action for position p
  if p has a right child rc **then**
    inorder(rc) {recursively traverse the right subtree of p}

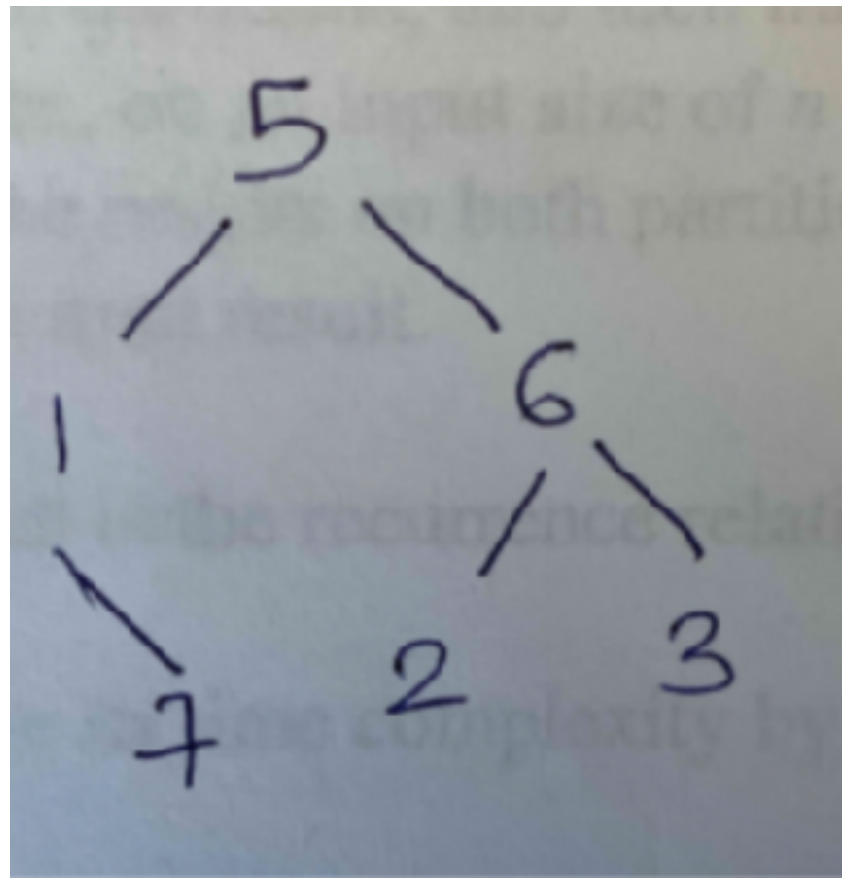
3 + 1 x 4 / 9 - 5 + 2

Tree Traversal Exercises

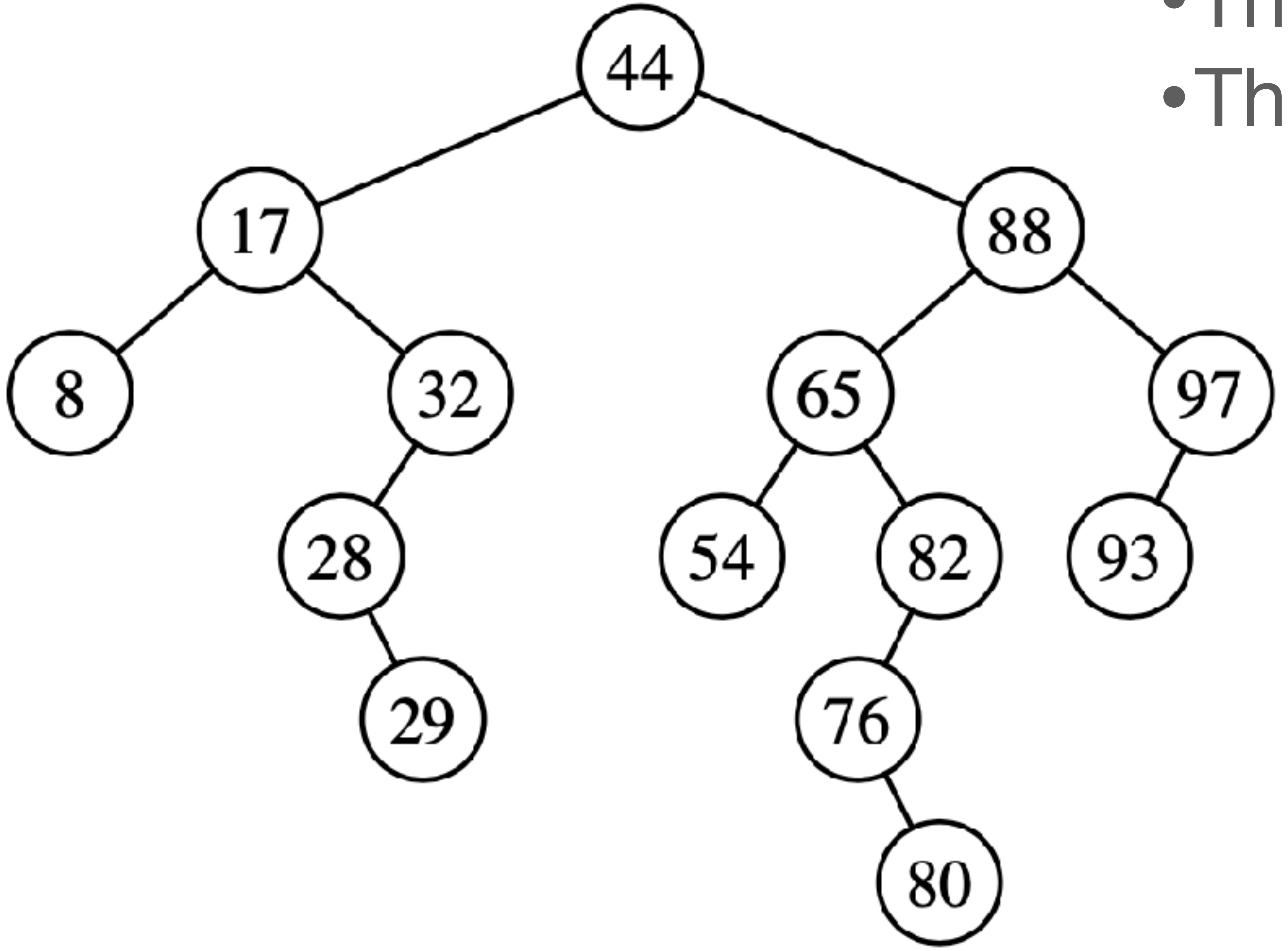


Draw that binary tree, if its inorder traversal is [8, 20, 2, 14, 3, 26] and the preorder traversal is [14, 20, 8, 2, 26, 3].

Draw that binary tree, if its inorder traversal is [1, 7, 5, 2, 6, 3] and the postorder traversal is [7, 1, 2, 3, 6, 5].



Binary Search Trees



For each node v,

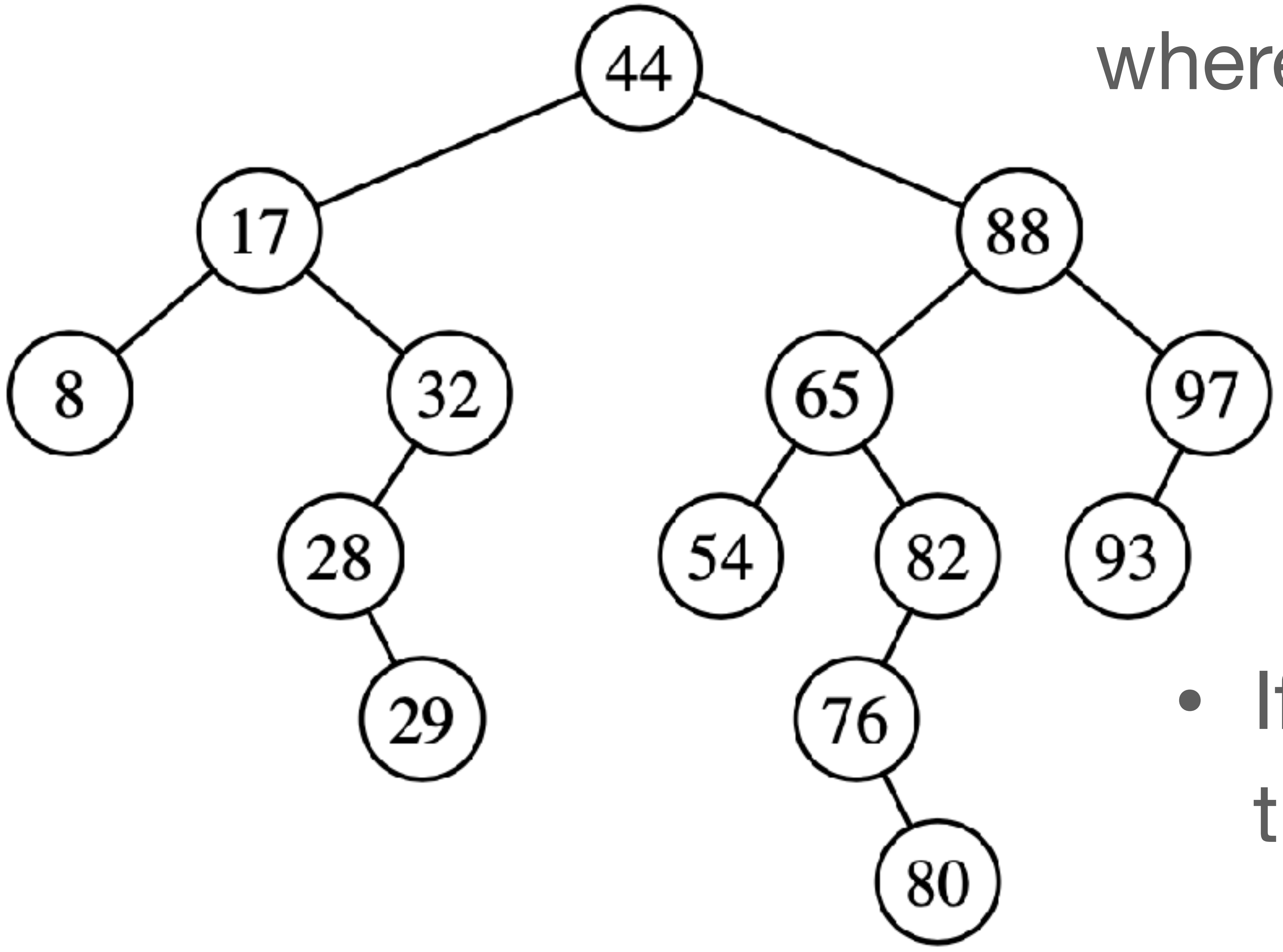
•The nodes on the **left** subtree are **less** than v

•The nodes on the **right** subtree are **greater** than v

What do you observe if you perform an inorer traversal of a binary search tree?

- Search queries
- Predecessor queries
- Successor queries

Binary Search Trees



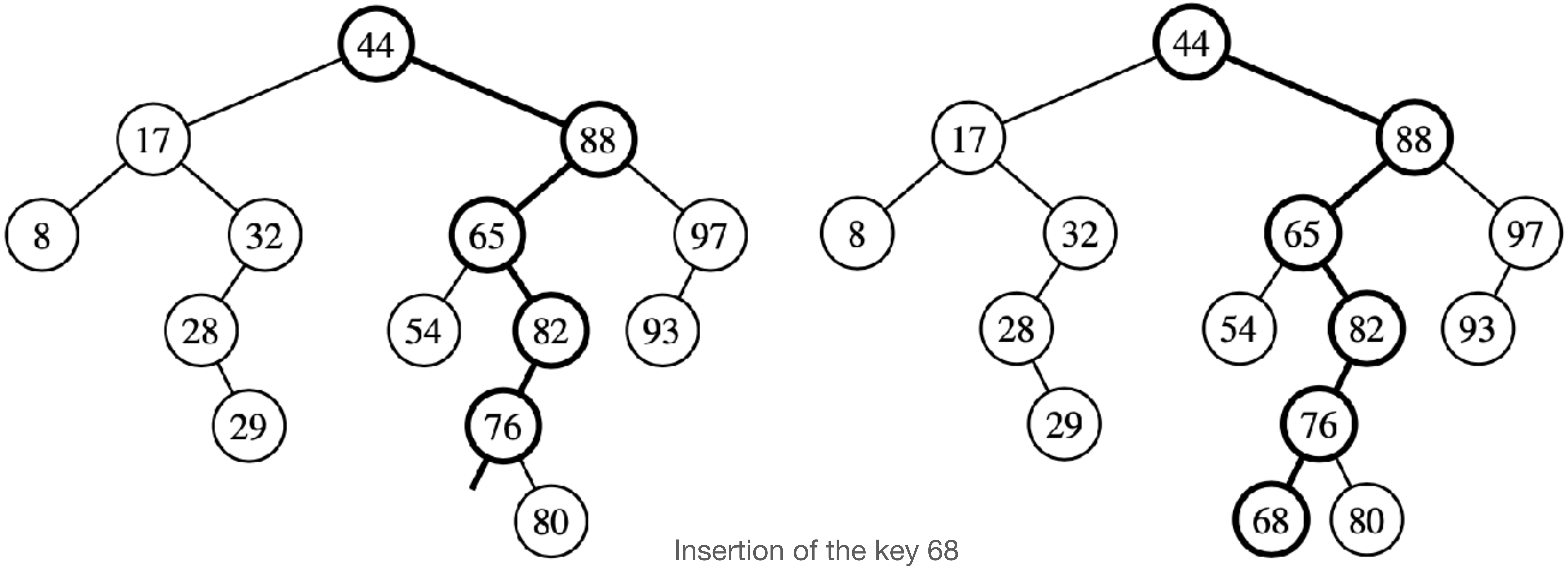
Search queries in $O(h)$-time worst-case complexity, where h is the height of the tree.

- If we maintain the BST as a complete-binary tree, then $O(\log n)$-time as $h = \lceil \log n \rceil$.

- We can initially construct BST complete-binary, but what happens with **insert/delete** ?

Binary Search Trees - Inserting a Node

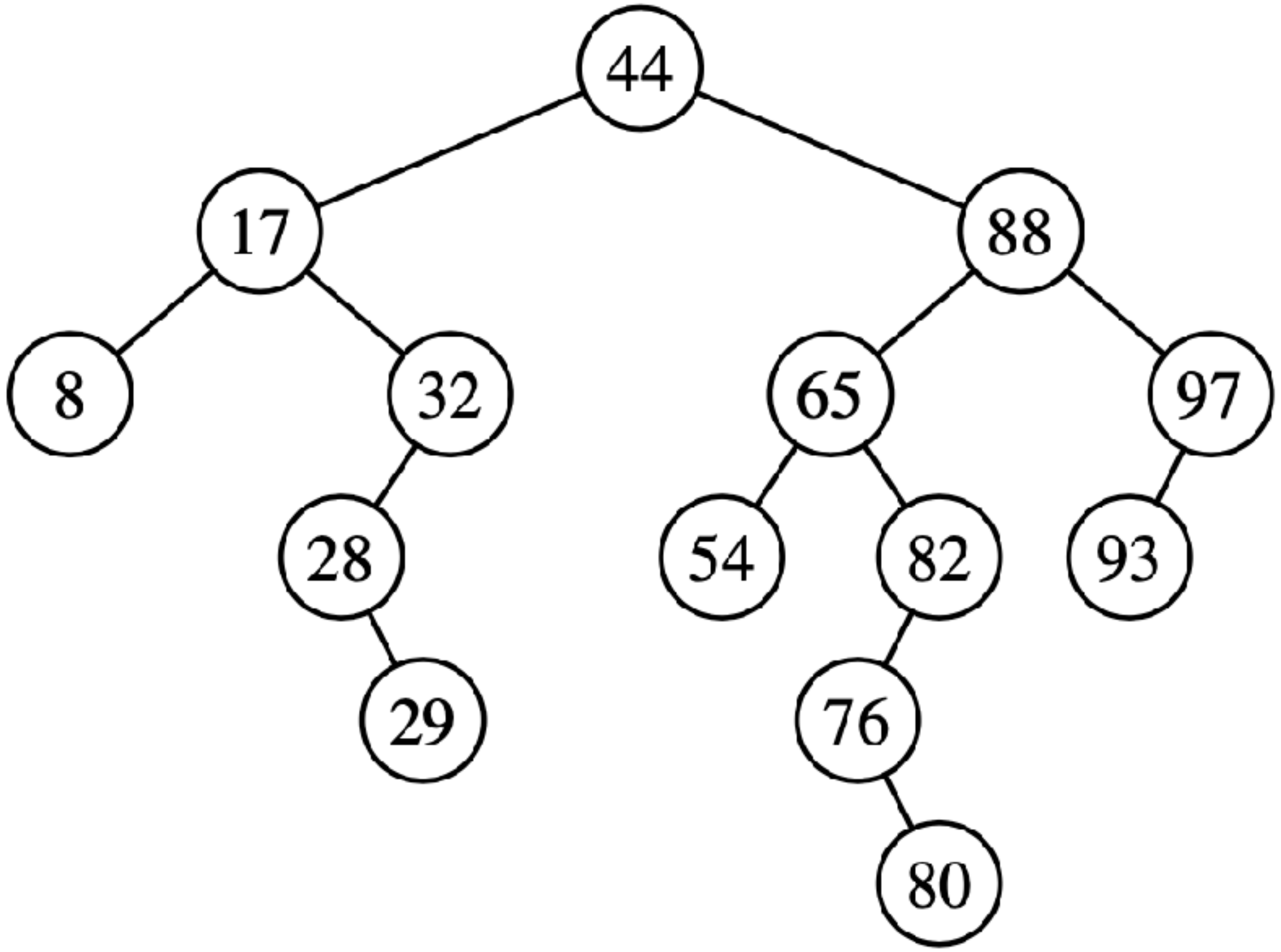
- First search the to-be-inserted key on the tree
- When we arrive the position, insert the key as a left child if it is less, or as a right child if it is greater.



Insertion of the key 68

Binary Search Trees -Predecessor/Successor Queries

Predecessor (before), Successor (after) queries



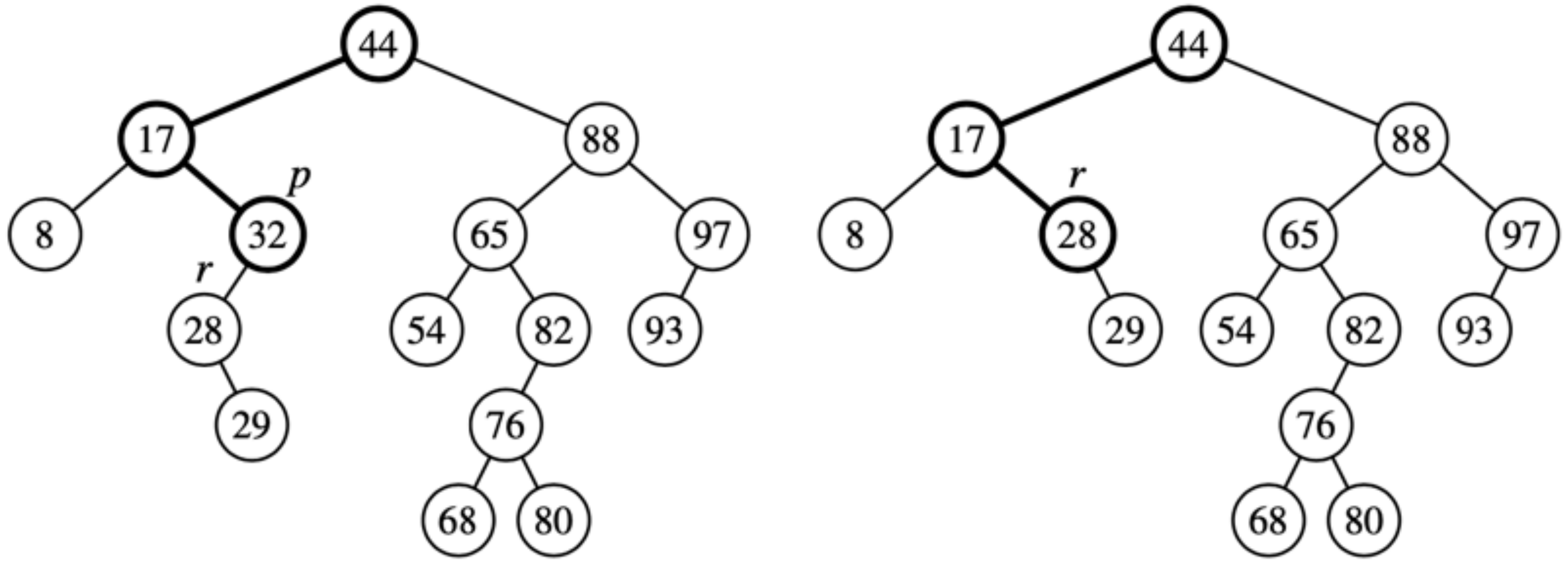
Predecessor (before): **Rightmost** of the **left** subtree
What if the left subtree is empty ?

Then navigate through ancestors until the ancestor node is a **right** child.

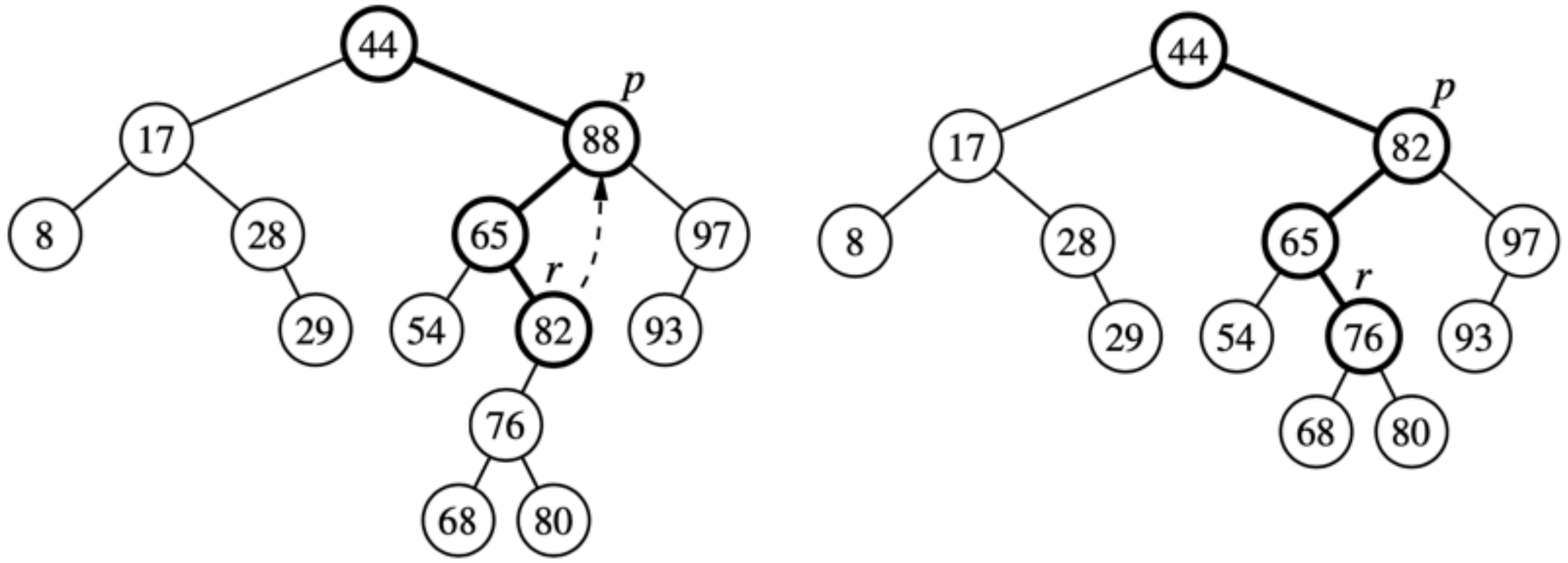
Successor (after): **Leftmost** of the **right** subtree
What if the **right** subtree is empty ?

Then navigate through ancestors until the ancestor node is a **left** child.

Binary Search Trees - Deleting a Node



If the node has only one child, trivial.



Else,

- Find the largest key (**predecessor** query) before the node, which is the rightmost position of the left subtree
- Swap this new node with the to-be-deleted node
- Now to-be-deleted node has no right child for sure and can be deleted with the trivial method

Questions, comments ?

- We studied the basic tree data structure and reviewed the binary search trees.
- We will continue with the amortized analysis in the next lecture.