Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Trees I: General Trees and Binary Trees

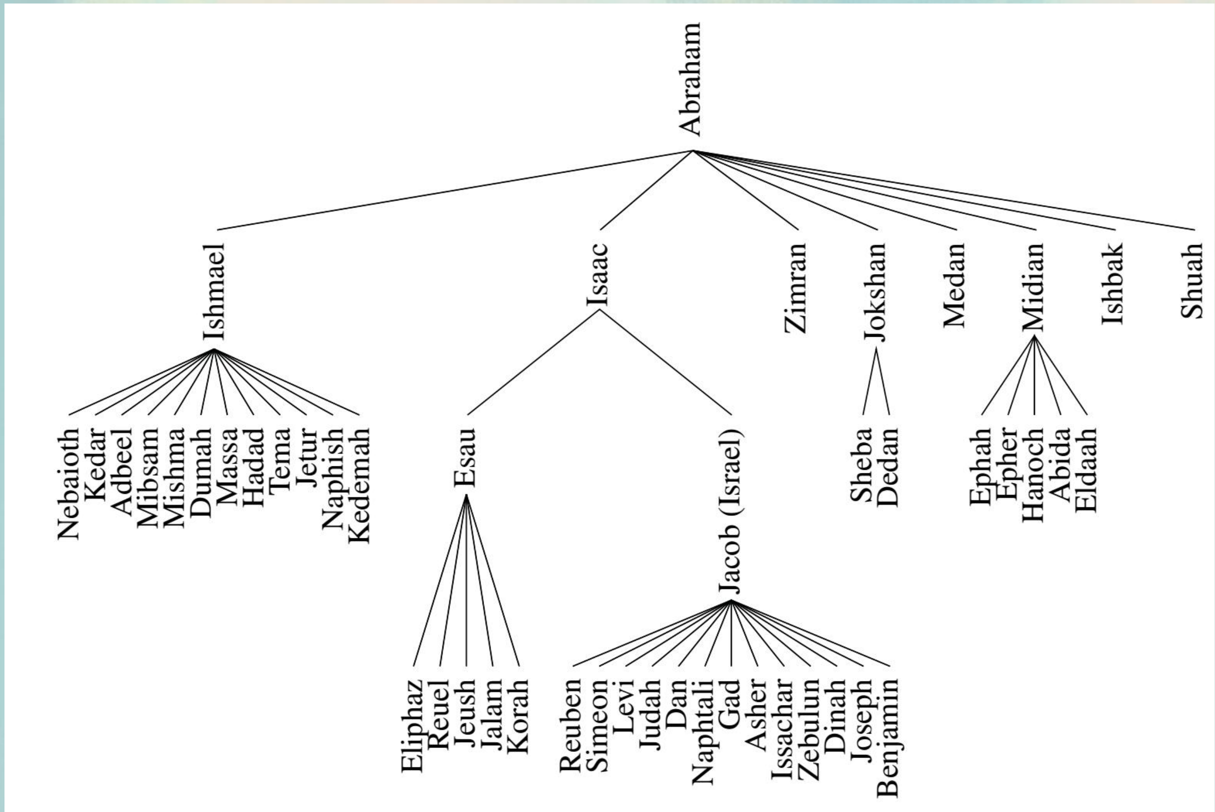
""

CS2013: Programming with Data Structures

General Trees

General Trees

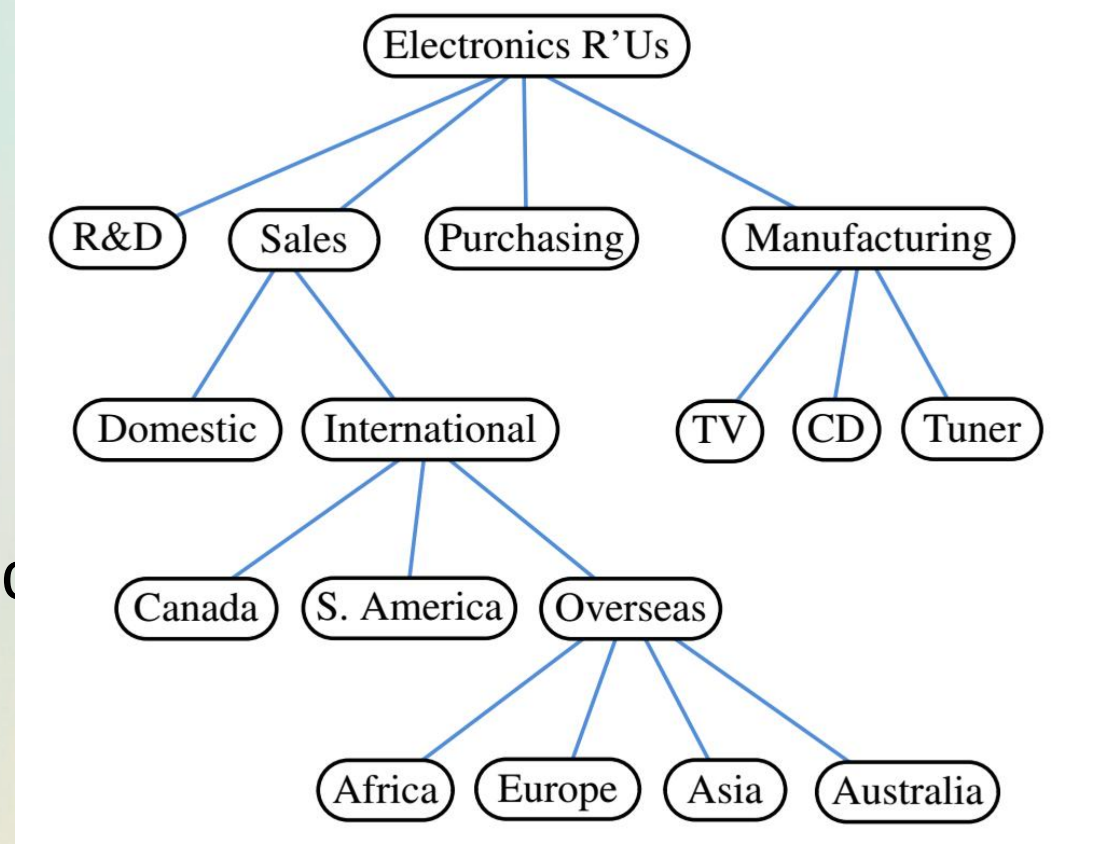
- ***tree***: an abstract data type which stores its elements non-linearly using a hierarchical structure.
  - instead of thinking about elements being before or after in a sequence, we think of them as being above or below each other.
  - we use the terms, parent/child or ancestor/descendant.



Tree Properties

- ***root***: the top-most node in the tree.

- ***parent:***
  - the node directly before another node.
  - all nodes have parents exc for the root.



- ***child***:
  - any node that comes directly after another node.
  - a parent node can have zero or more children.

Formal Tree Definition

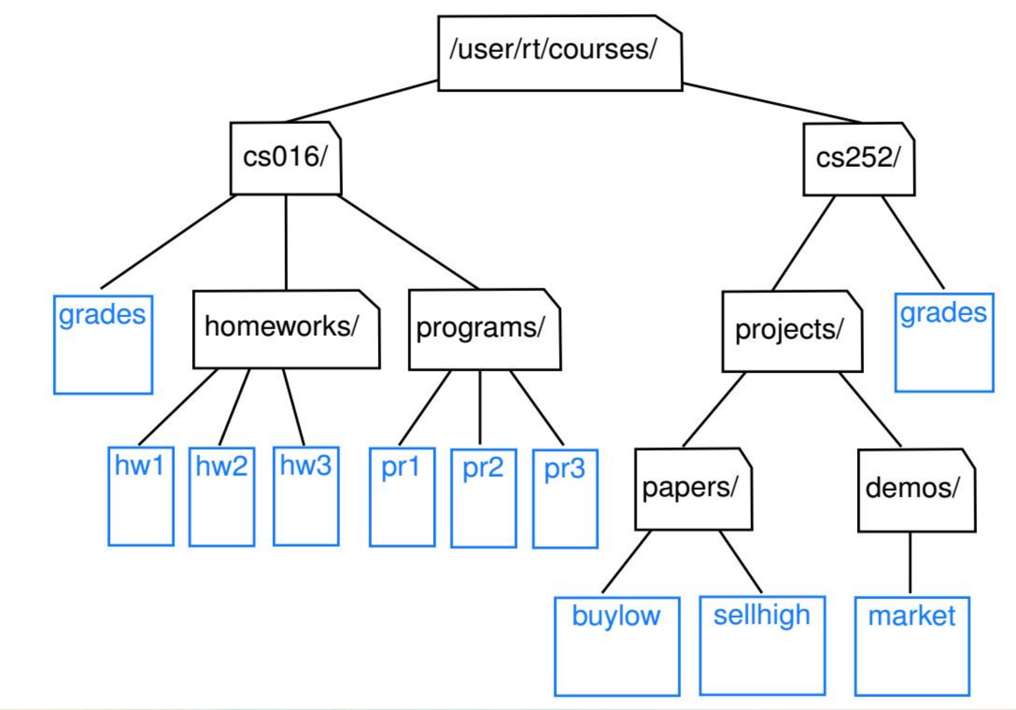
- a ***tree T*** is a set of nodes storing elements such that the nodes have a ***parent-child*** relationship that satisfies the following properties:
  - If ***T*** is nonempty, it has a special node called the ***root*** of T, that has no parent.
  - Each node *v* of ***T*** different from the root has a unique parent node *w*; every node with parent *w* is a child of *w*.

- The formal definition allows the tree to be empty (does not have any nodes).

- The definition also allows us to define the tree recursively such that tree *T* is either empty or consists of a node *r* called the root of *T*, and a possibly empty set of subtrees whose roots are the children of *r.*

Other Node Relationships

- ***siblings***: two or more nodes that share the same parent.
- ***leaf (external node)***: a node with no children.
- ***internal node***: a node with children.
- ***ancestor***: a node *u* is an ancestor of a node *v* if u is the parent of *v* or an ancestor of the parent of *v*.
- ***descendant***: a node *v* is a descendant of a node *u* if *u* is an ancestor of *v*.
- ***subtree*** of *T* rooted at node *v*: the tree consisting of all descendants of *v* in *T* (including *v* itself).



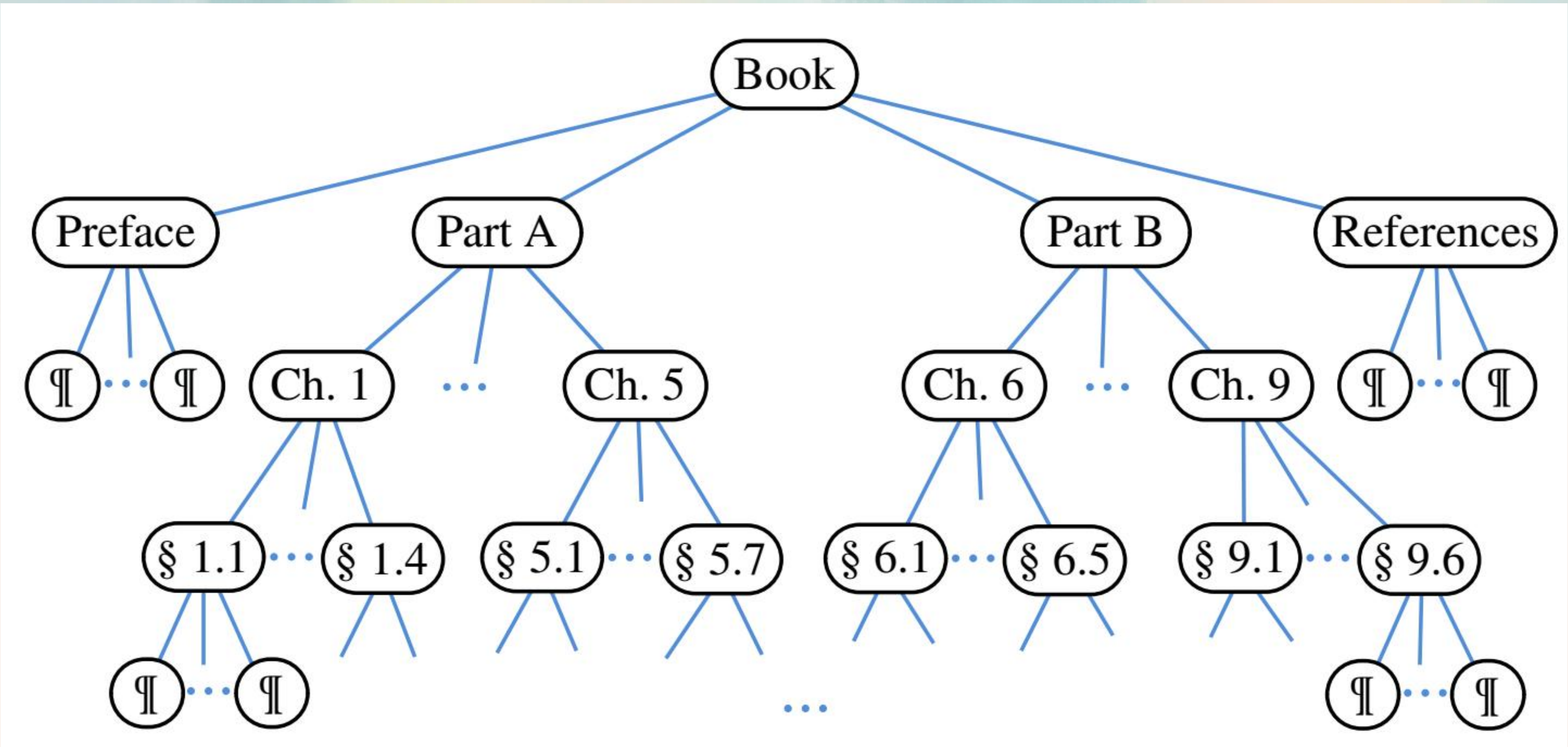
Edges and Paths in Trees

- ***edge***: an edge of tree *T* is a pair of nodes (*u*, *v*,) such that *u* is the parent of *v*, or vice versa.

- **path**: a path of *T* is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.

Ordered Trees

- ***ordered tree***: a tree is ordered if there is a meaningful linear order among the children of each node.
  - the children of each node are sorted in some way.



The Tree ADT

- The following defines the common operations of most trees. More functions of course can be added as necessary. NOTE: That for the continuing discussion, node refers to a node in the tree.

`getElement():`	Returns the element (value) stored in the given node.
`root():`	Returns the root node of the tree (or null if empty).
`parent(node):`	Returns the parent of the given node (or null if *node* is the root).
`children(node):`	Returns an iterable collection containing the children of *node* (if any). If the tree is an ordered tree, then the children will be given in the correct sorted order.
`numChildren(node):`	Returns the number of children of *node*.
`isInternal(node):`	Returns true if *node* has at least one child.
`isExternal(node): or isLeaf(node):`	Returns true if *node* has no children.
`isRoot(node):`	Returns true if *node* is the root of the tree.

The Tree ADT

- Trees may also define the following methods:
  - size(): return the number of nodes.
  - isEmpty()
  - iterator()
  - positions(): returns a iterable collection of all positions of the tree.

Tree Depth

- ***depth***: let *p* be a position in tree *T*. depth of *p* is the number of ancestors of *p* (not including *p*).

- Depth can be computed recursively:
  - if p is the root, then depth of p is 0.
  - otherwise, the depth of *p* is one plus the depth of the parent of *p*.

- Running time of depth:
  - running time of depth(p) for position p is $O(d_p + 1)$, where d_p denotes the depth of p in the tree.
  - the algorithm performs a constant time recursive step for each ancestor of p.
  - runs in O(n) worst-case time, where n is the total number of positions in the tree since a position of T may have depth n – 1 if all nodes form a single branch (***skew tree***).
  - such a running time is a function of the input size, it is more informative to characterize the running time in terms of the parameter d_p as this may be smaller than *n*.

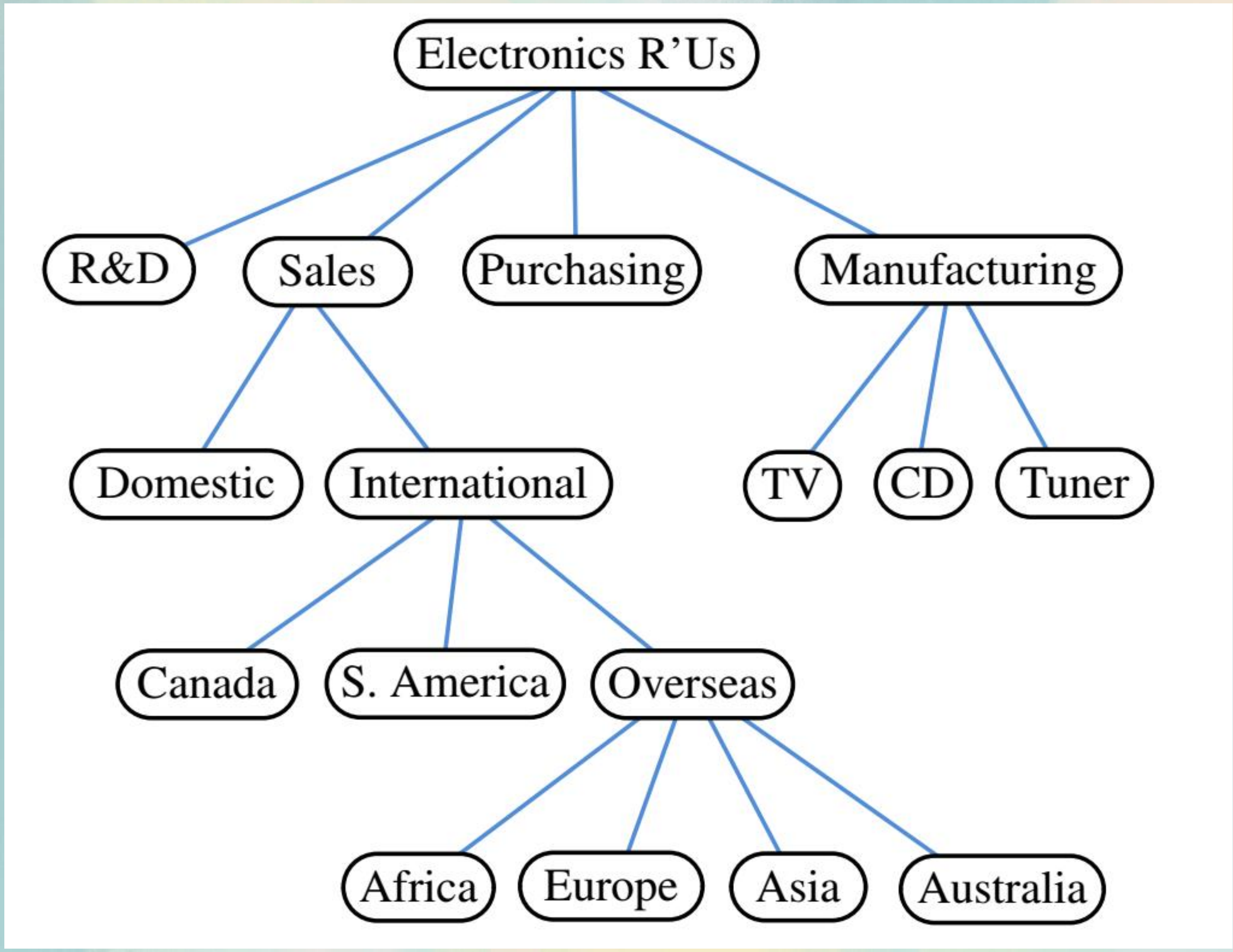
Depth of a Node Algorithm

- Note: The input to this function could be an actual node of the tree, or a value stored in the tree. If the input is a value, you would have to search for the node containing the value, first.

```
depth(node):
  if node is the root:
    return 0;
  else:
    return 1 + depth(parent(node))
```

Height of a Tree

- ***height of a tree***: the maximum of the depths of all positions (or zero if the tree is empty).



Height Algorithm (Bad Example)

```
heightBad():
  h = 0
  for all nodes n:
    if (n is a leaf):
      h = maximum(h, depth(p))
  return h
```

- Fact 1: getting a list of all nodes for the loop to traverse can be done in O(n) time.
- Fact 2: we used the depth function whose running time is $O(d_p + 1)$.
- Total running time = $O(n + \Sigma_{p \in L}(d_p + 1))$, where L is the set of leaf nodes of T
  - NOTE: that in the worst case the summation of this runtime is $O(n^2)$.
- This algorithm can result in an $O(n^2)$ runtime, but we can do better.

Height Algorithm (Good)

- The height can be computed in O(n) as follows using recursion:
  - if node is a leaf, the height of p is 0.
  - Otherwise, the height of the node is one more than the maximum of the heights of the node's children.

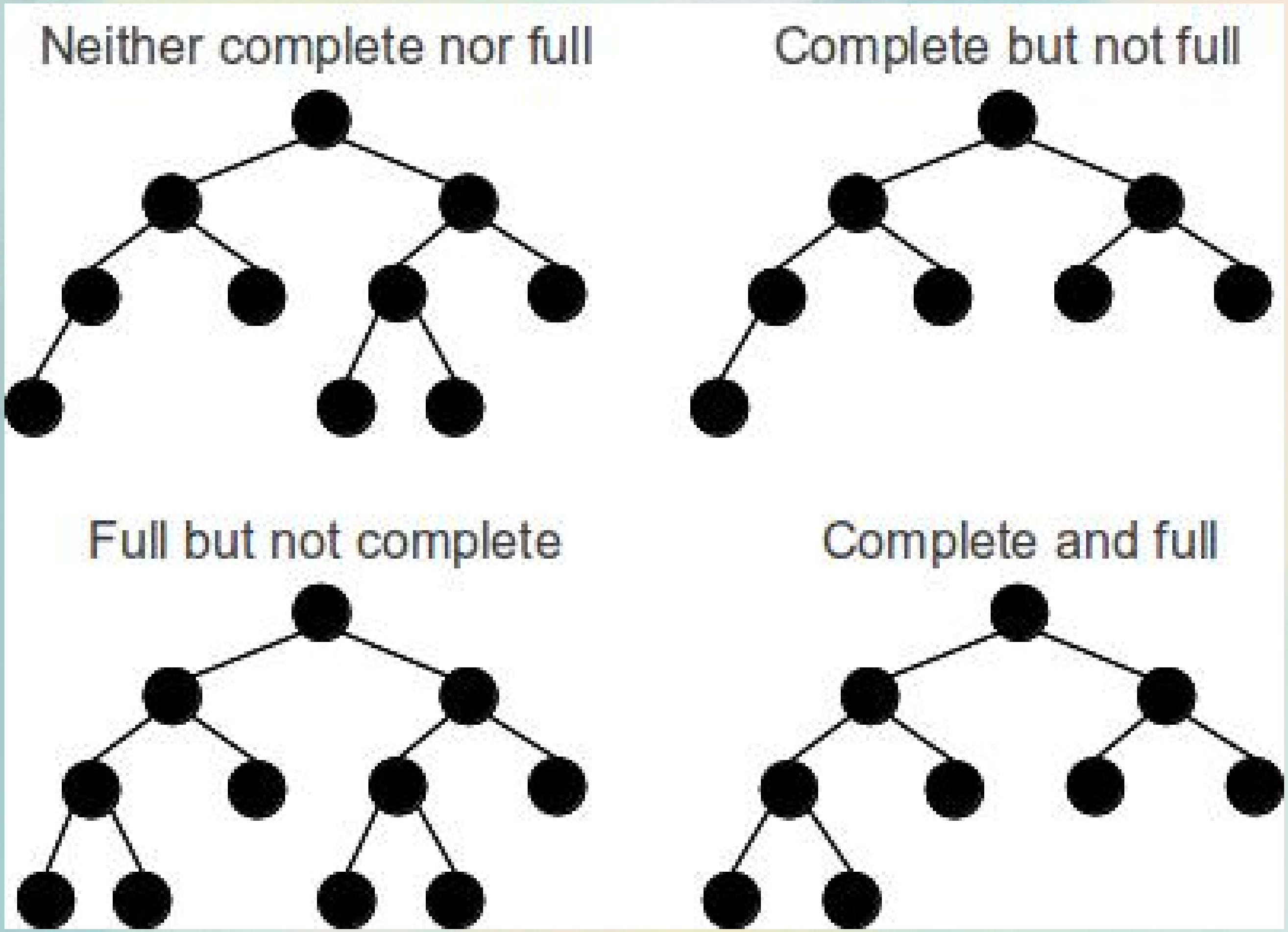
```
height(node):
  h = 0
  for all children c of node:
    h = maximum(h, 1 + height(c))
  return h
```

- If the algorithm is called on the root, then the recursion will be called once for each node of the tree.

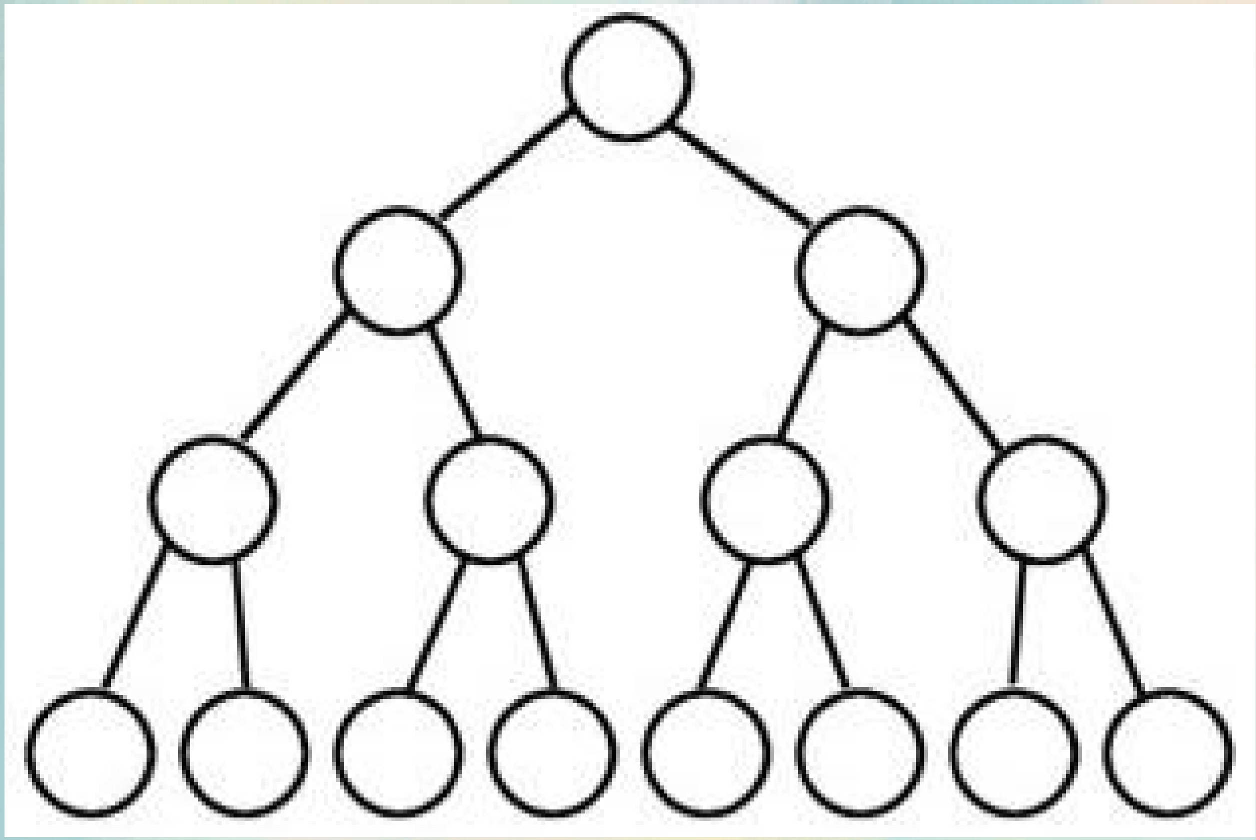
Binary Trees

- a ***binary tree*** is an ordered tree with the following properties:
  - 1. Every node has at most two children.
  - 2. Each child node is labeled as being either a ***left child*** or a ***right child***.
  - 3. A left child precedes a right child in the order of the children of a node.
- ***left subtree*:** the subtree rooted at a left internal node.
- ***right subtree***: the subtree rooted at a right internal node.
- ***full (proper) binary tree:*** a binary tree where each node is either a leaf or has exactly two children.
- ***complete binary tree***: a binary tree where all of its levels are completely full, except possibly the last level, and the last level has all of its nodes as far left as possible.
- ***perfect binary tree***: a binary tree where each level is filled with its maximum number of nodes.

Complete vs Full Binary Trees

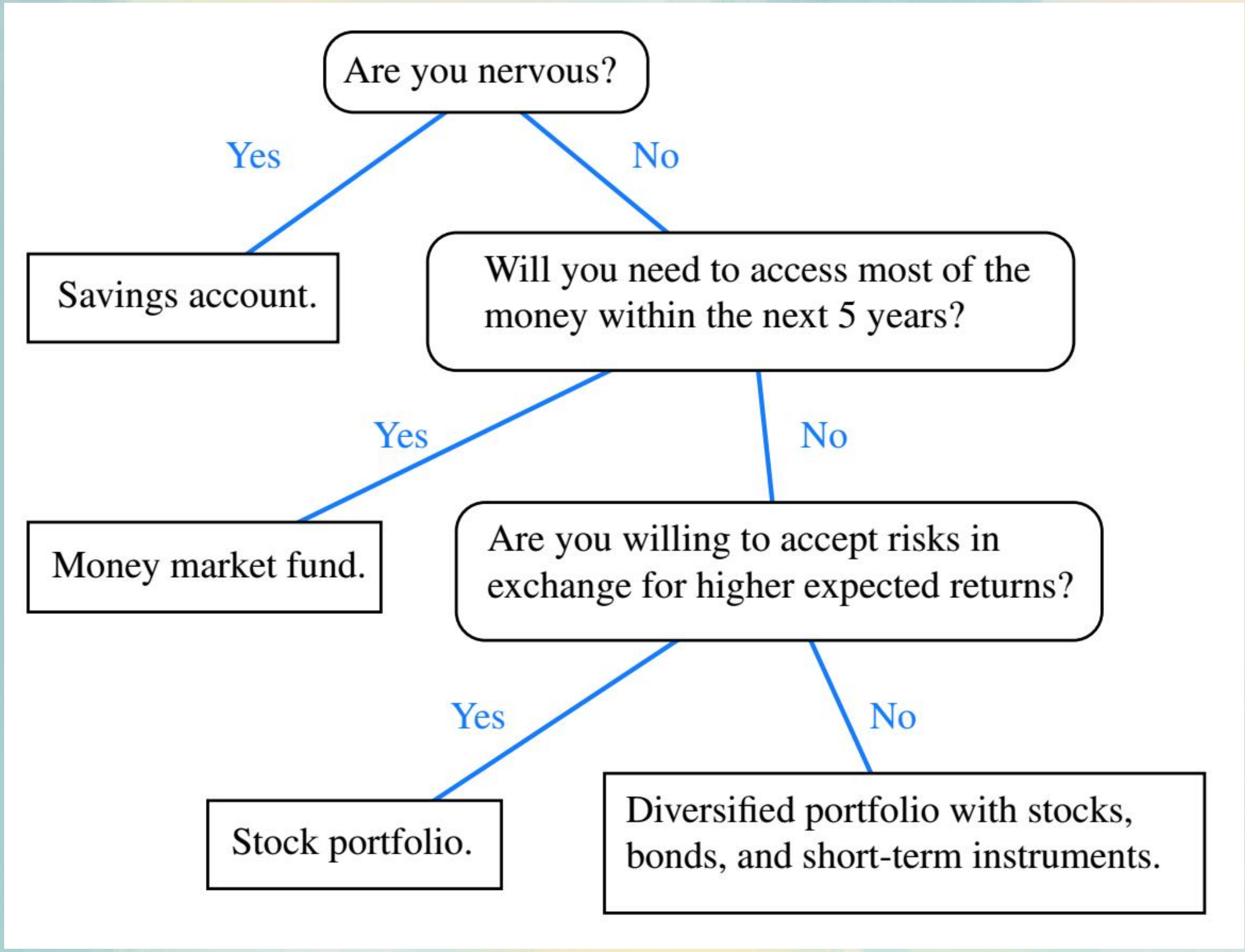


- Perfect Binary Tree:



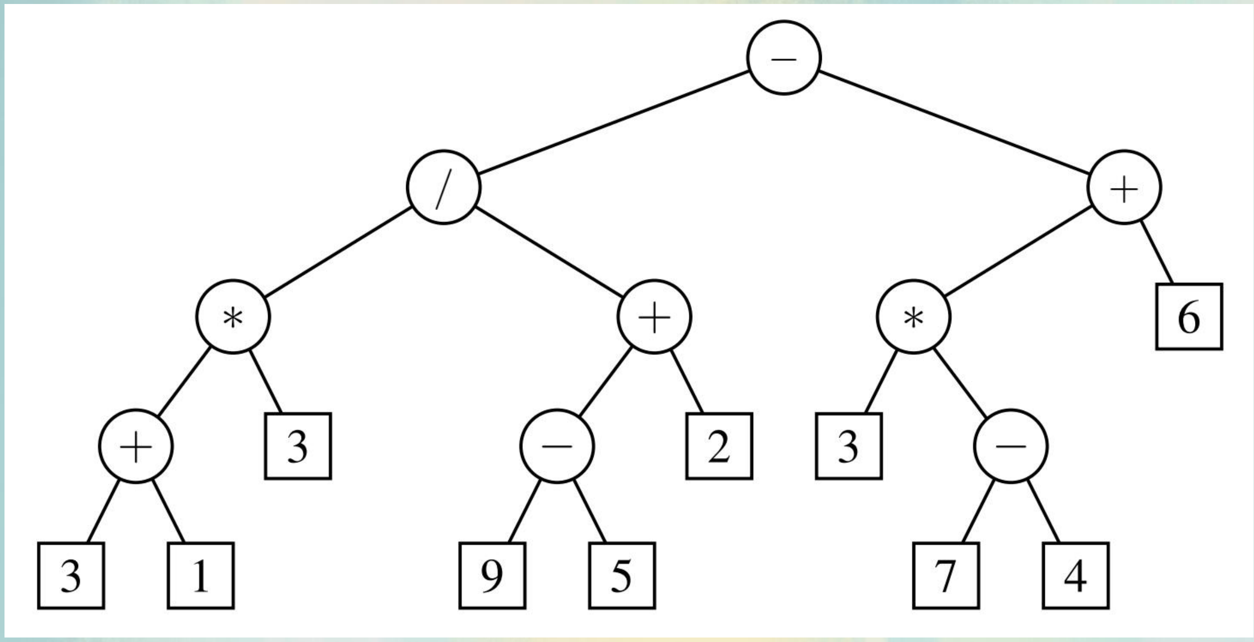
Binary Tree Examples: Decision Tree

- A decision making tree can be represented internally as a binary tree where choosing yes means taking a left path, and choosing no means taking a right path.



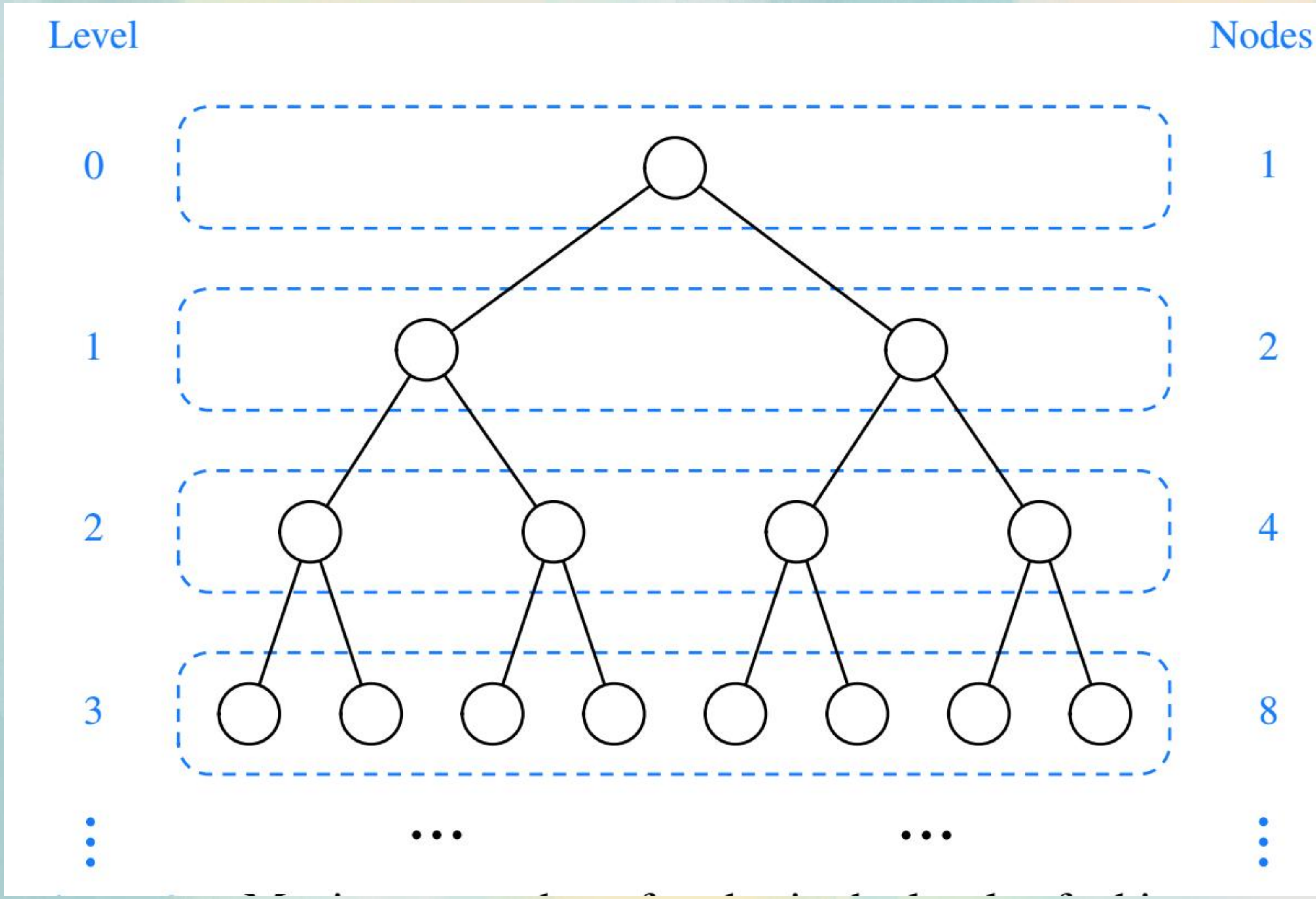
Binary Tree Examples: Arithmetic Expression

- In this example an arithmetic expression can be represented using a tree. The leaves are the numbers and the nodes are the arithmetic operators.
  - If the node is a leaf, its value is that of a variable or constant.
  - If the node is internal, its value is defined by applying its operation to the values of its children.



Properties of a Binary Tree

- ***level #***: the level *n* of tree *T* is the set of all nodes at the same depth. In general, a level has at most 2^n nodes.

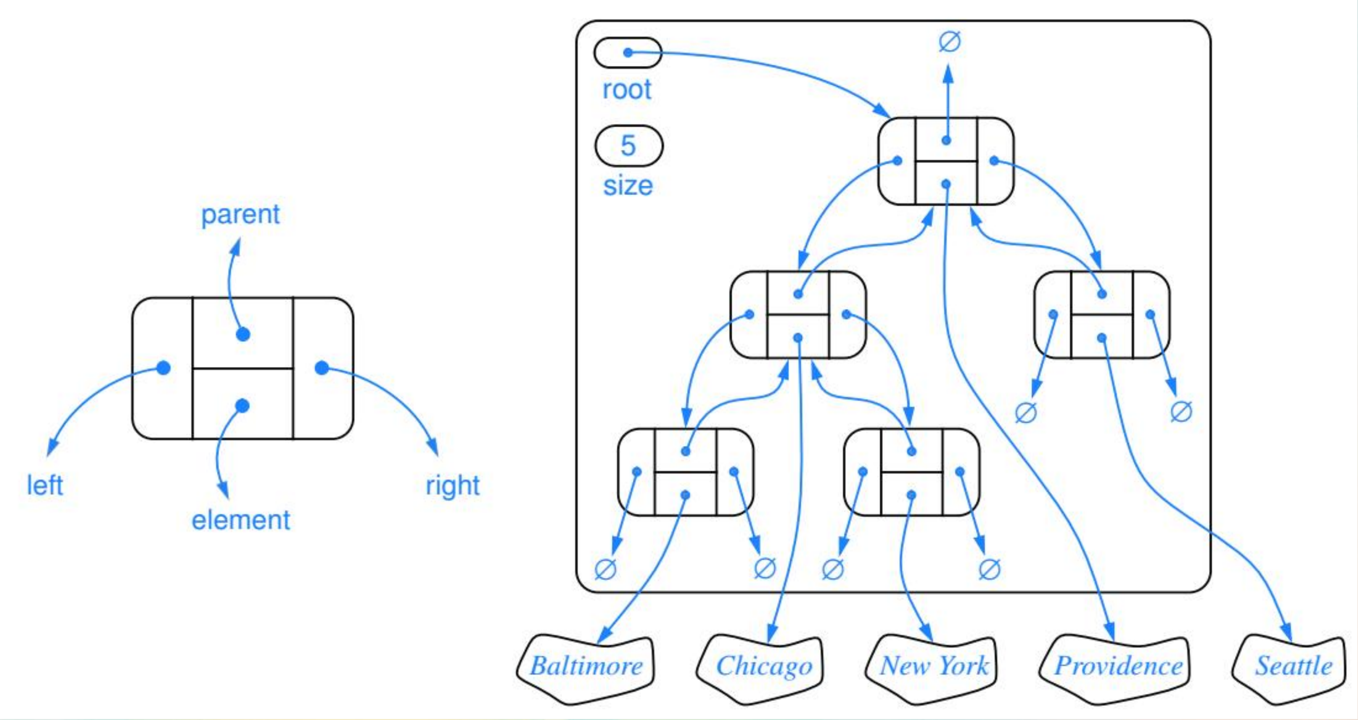


Implementing Binary Trees

- Trees can be implemented with a linked structure.
- A tree node might look something like this:

```
class TreeNode<E> {
   private TreeNode<E> parent;
   private TreeNode<E> left;
   private TreeNode<E> right;

   private E element;
}
```



Binary Tree Implementation: Array-Based

- Another representation of a Binary Tree would be to use an array with the following properties:
  - If a node is the root, then its index is 0.
  - If a node is a left child of a parent then its index is:

    2(parent index) + 1
  - If a node is a right child of a parent, then its index is:

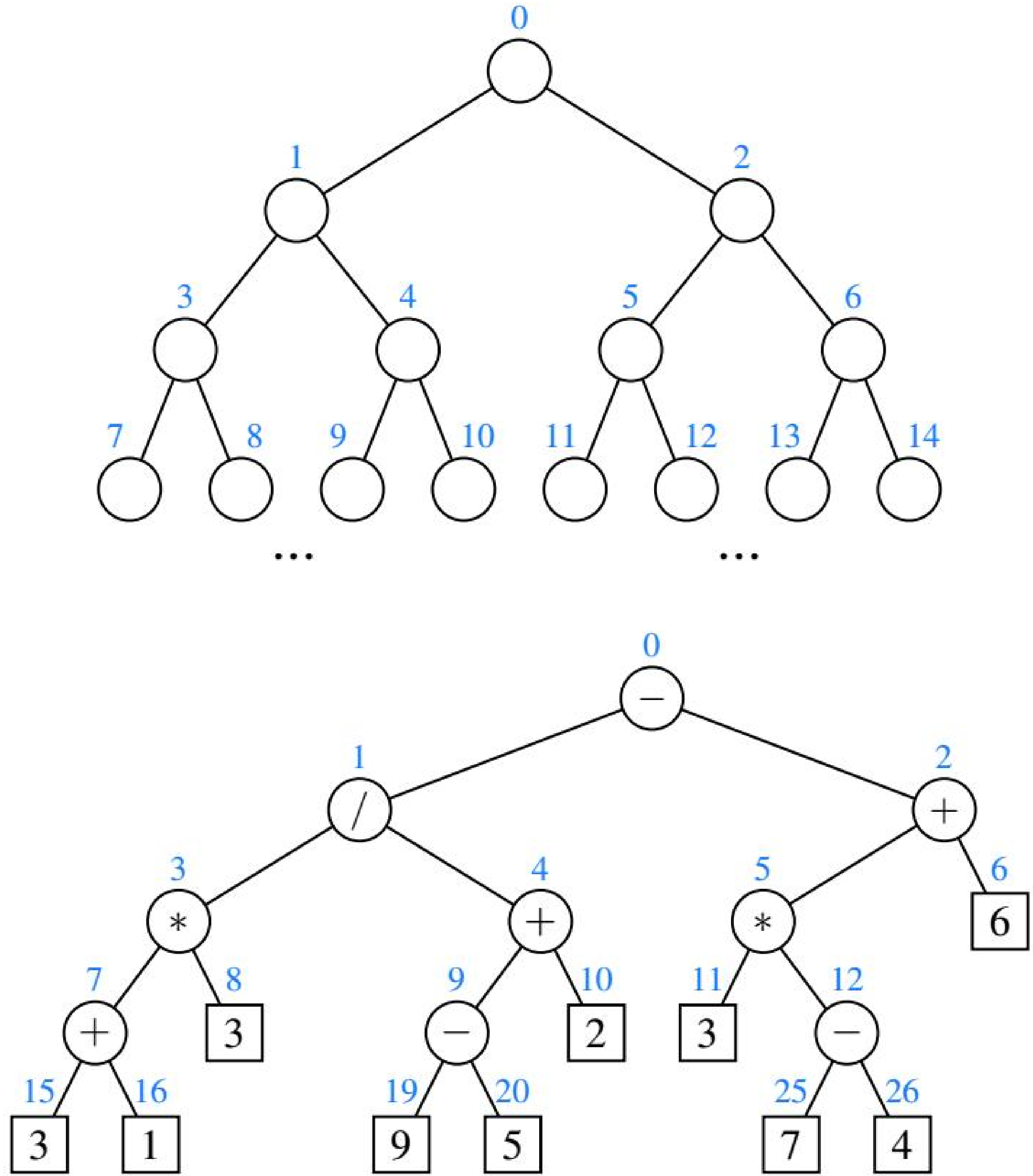
    2(parent index) + 2

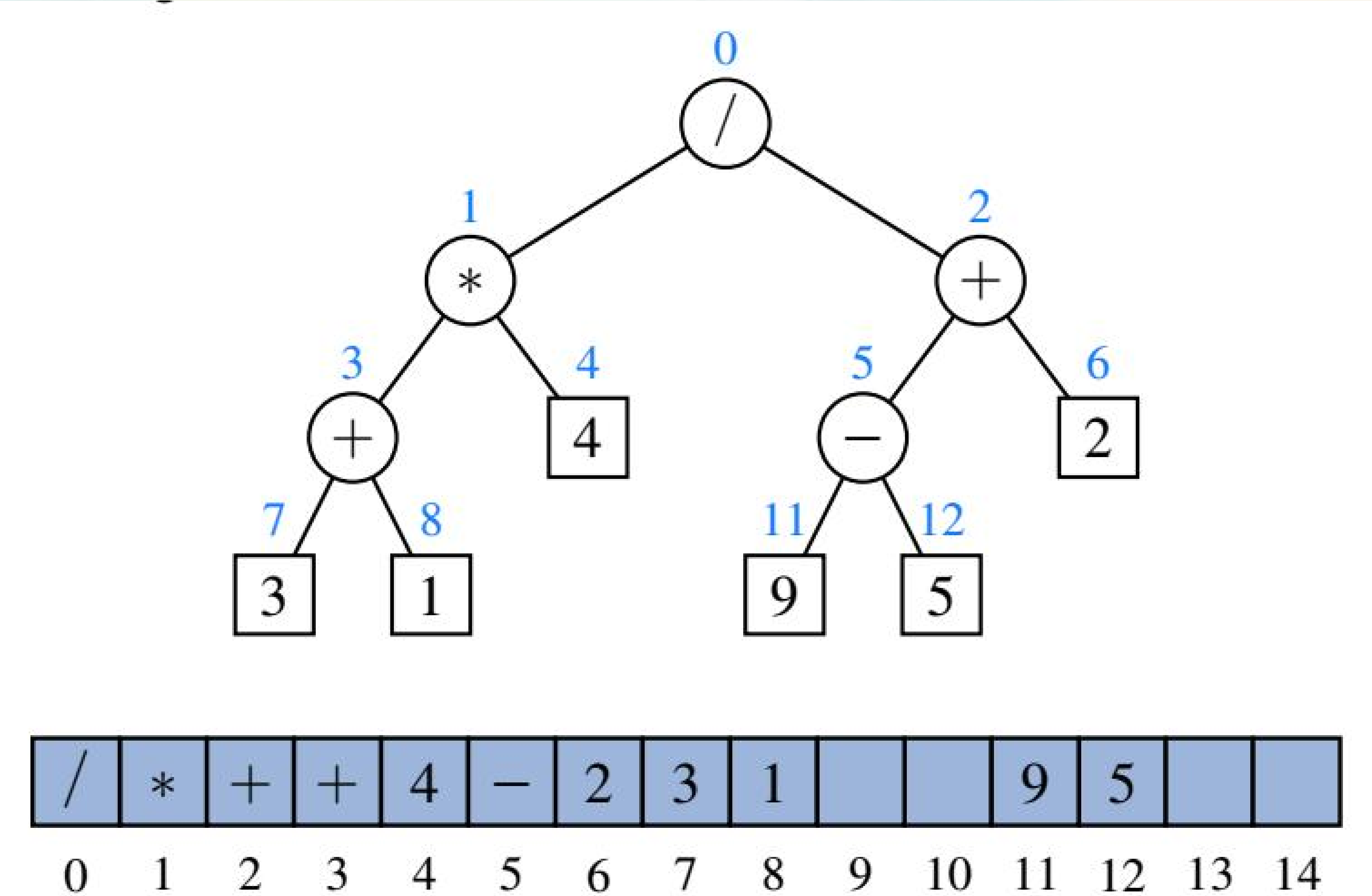
- This is known as *level numbering* of the positions in a binary tree. Each node is assigned a number from left to right in increasing order.

- NOTE that the numbers are only consecutive if it is a perfect binary tree.

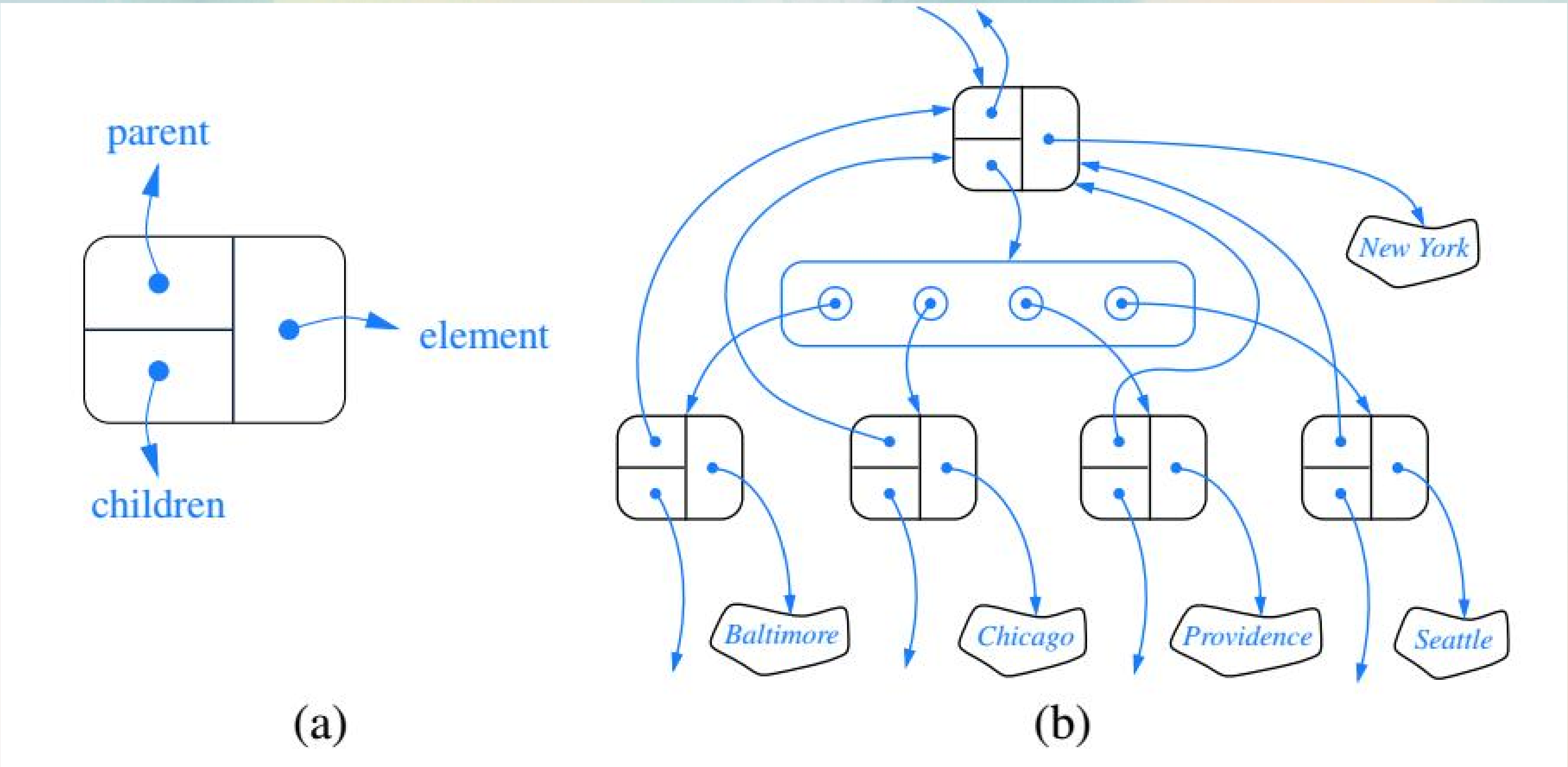
(a)

(b)


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
...
...
0
−
1
/
2
+
3
*
4
+
5
*
6
6
7
+
8
3
9
−
10
2
11
3
12
−
15
3
16
1
19
9
20
5
25
7
26
4


0
/
1
*
2
+
3
+
4
4
5
−
6
2
7
3
8
1
11
9
12
5
/ * + + 4 − 2 3 1 9 5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

General Tree Linked Structure



Tree Traversal Algorithms

Tree Traversal

- a ***traversal*** of a tree ***T*** is a systematic way of visiting all the nodes of the tree.
  - NOTE: ***visiting*** depends on the algorithm, but could mean incrementing the value of each node, printing the value in each node, performing an action on each node etc.

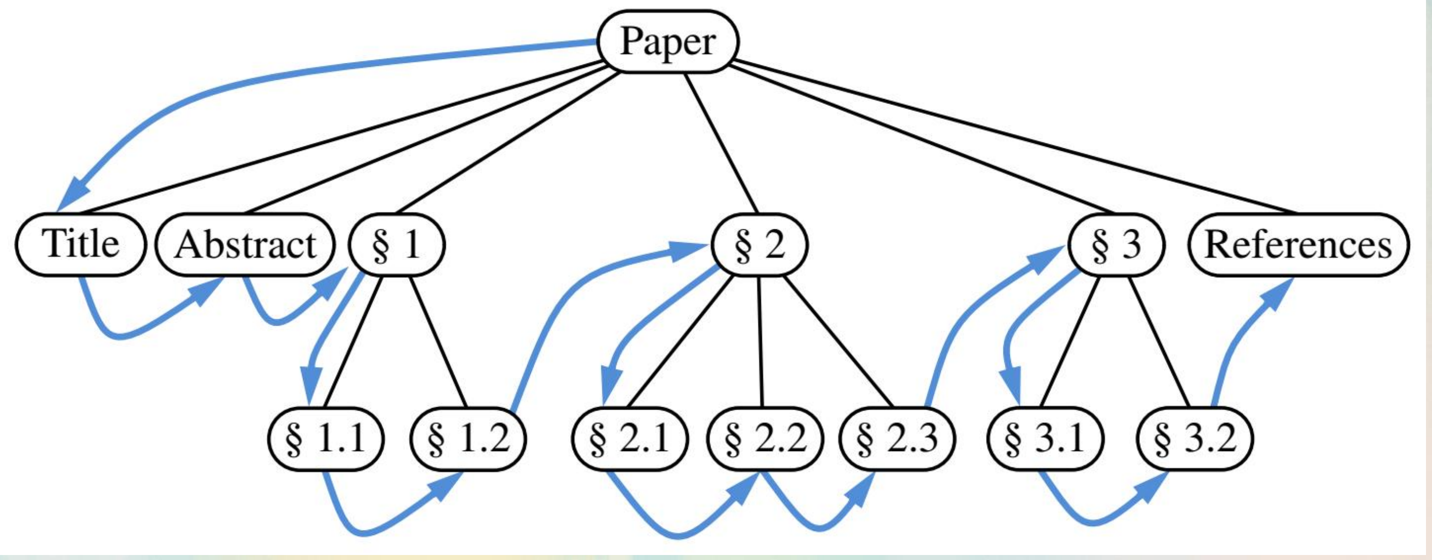
- traversals can be implemented using recursion or a stack if not using recursion.

- Each traversal is named for the order in which the root node of a subtree is visited.
  - For General Trees the root node can be visited before or after the child nodes. (Preorder and Postorder)
  - For a Binary Tree we have three traversals:
    - Root, Left Child, Right Child – Preorder
    - Left Child, Root, Right Child – Inorder
    - Left Child, Right Child, Root – Post Order
  - Both types of trees have a Breadth First Traversal.

Preorder Traversal – General Trees

- ***preorder*** traversal of a general tree means that the root is visited first and then the subtrees rooted at its children are visited recursively.
  - Also known as Depth-First Search
- If the tree is ordered, the children are visited according to their ordering.

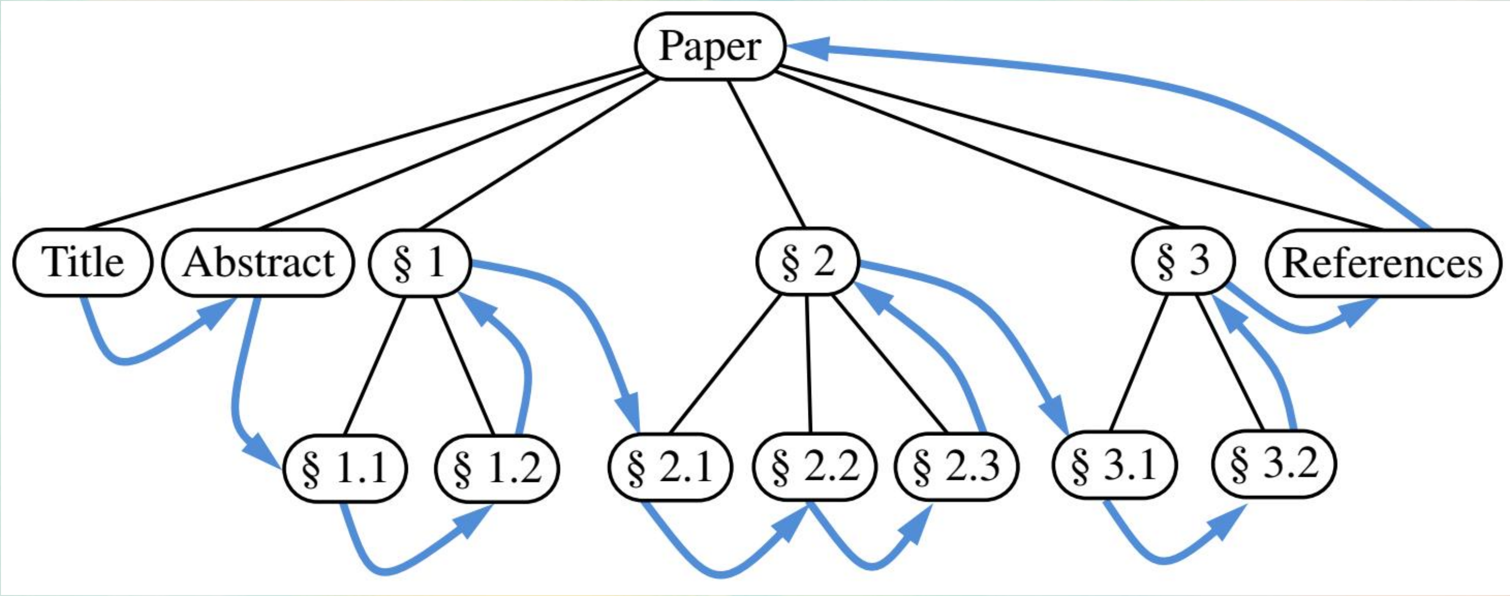
```
preorder(node):
  if node is null: return
  visit node
  for each child c of node:
    preorder(c)
```



Post Order Traversal – General Trees

- A ***postorder traversal*** recursively traverses the subtrees rooted at the children of the root first, and then visits the root.

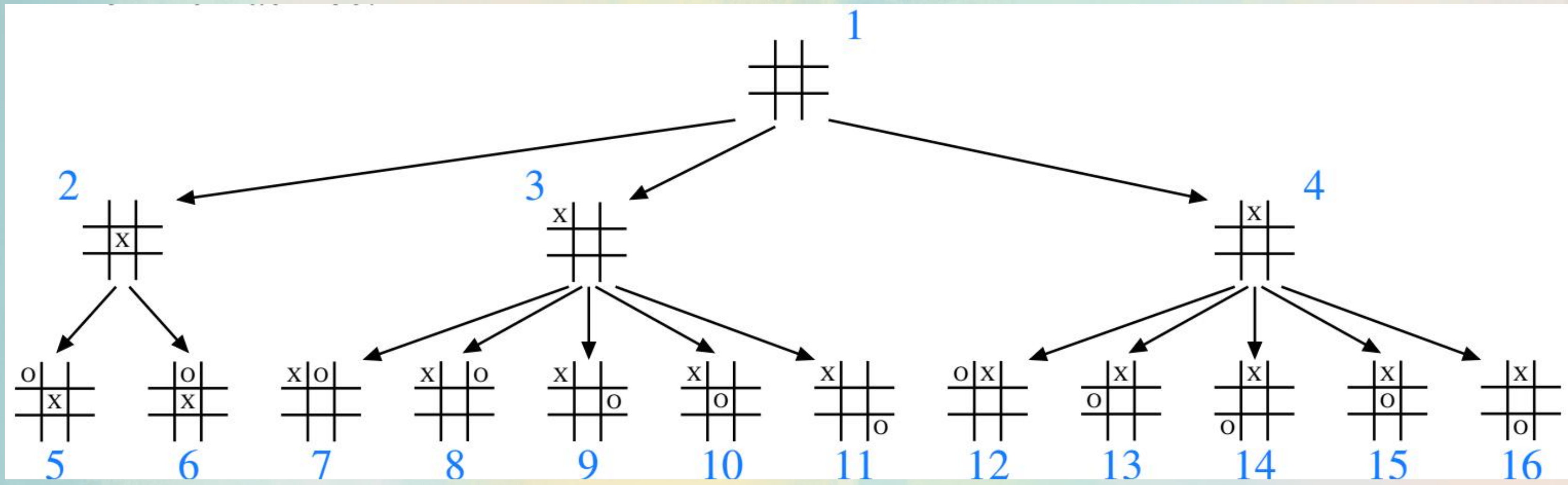
```
postorder(node):
  if node is null: return
  for each child c of node:
    postorder(c)
  visit node
```



Breadth-First Traversal – General Trees

- ***breadth-first traversal***: visiting all nodes a each level, usually from left to right, before moving down to the next level.

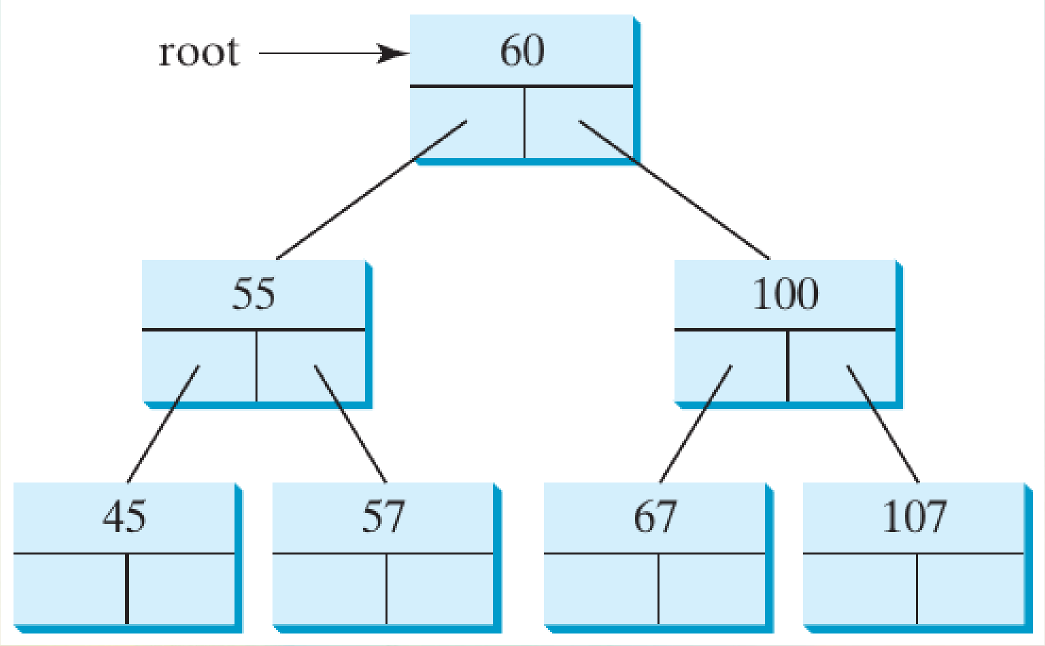
```
breadthfirst():
  create a Queue Q and add the root
  while Q not empty:
    node = Q.dequeue()
    visit node
    for each child c of node:
      Q.enqueue(c)
```



Preorder Traversal - Binary Tree

- ***preorder traversal***: of a binary tree, means that you visit the root of a subtree first before visiting the left and right children
  - also known as Depth-First Search

```
preorder(node):
   if node is null: return
   visit node
   preorder(node.left)
   preorder(node.right)
```

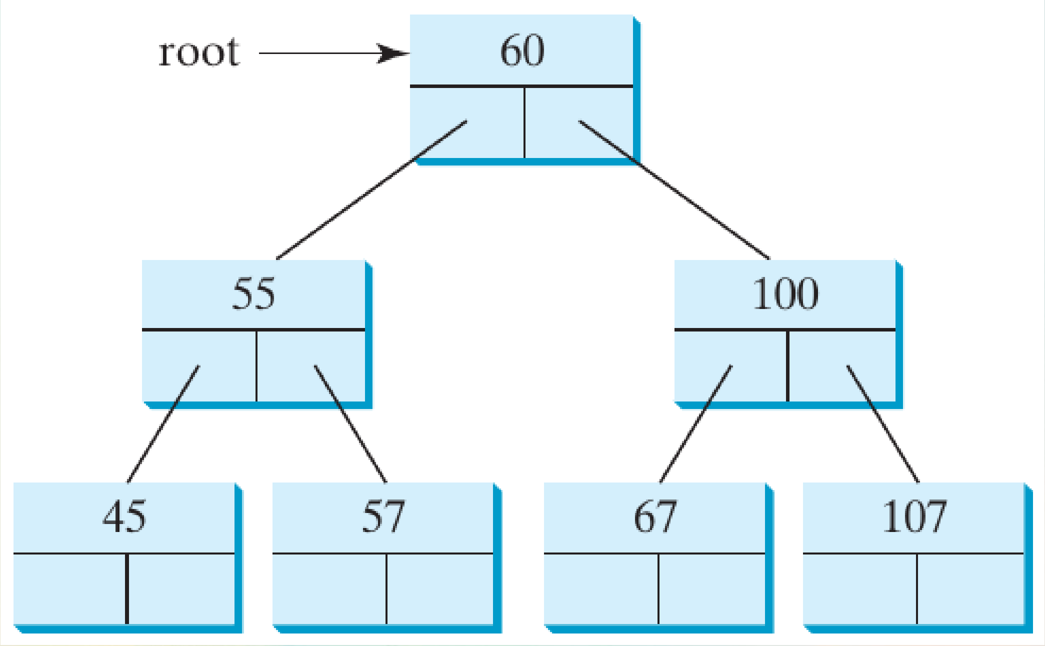


- In the example to the right the preorder traversal would be:

  60, 55, 45, 57, 100, 67, 107

Inorder Traversal - Binary Tree

- ***inorder traversal***: of a binary tree, means that you visit the left child first, then the root of the subtree, then the right child.
  - In a binary search tree, the nodes will be given in sorted order.




```
inorder(node):
  if node is null: return
  inorder(node.left)
  visit node
  inorder(node.right)
```

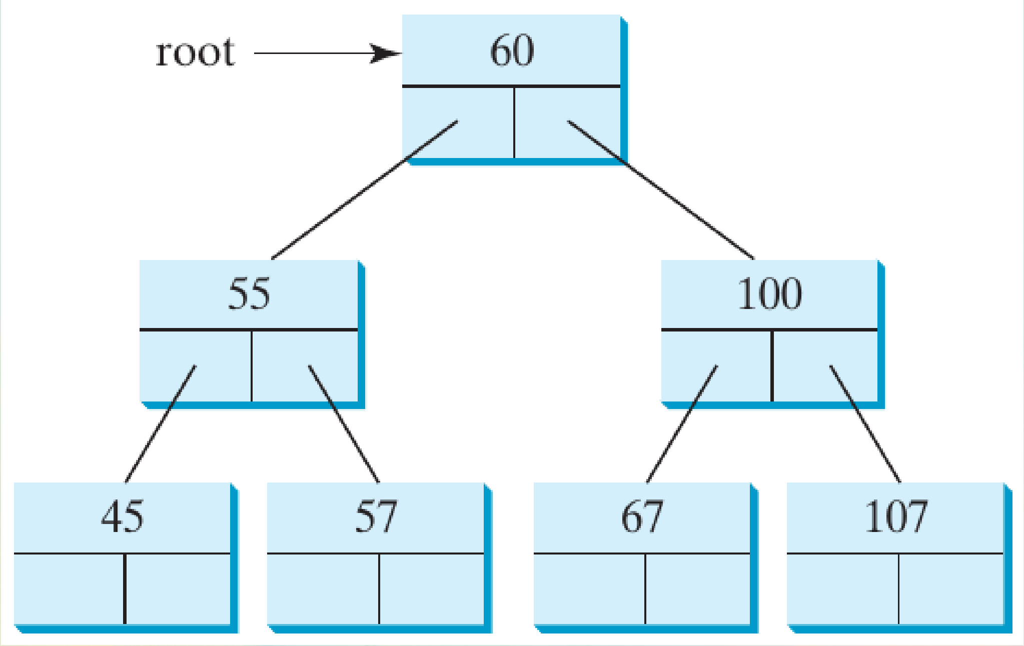
- In the example to the right the inorder traversal would be:

  45, 55, 57, 60, 67, 100, 107

Postorder Traversal - Binary Tree

- ***postorder traversal***: of a binary tree, means that you visit the left child first, then the right child, then the root of the subtree.
  - Finding the size of a directory uses post order.

```
postorder(node):
   if node is null: return
   postorder(node.left)
   postorder(node.right)
   visit node
```

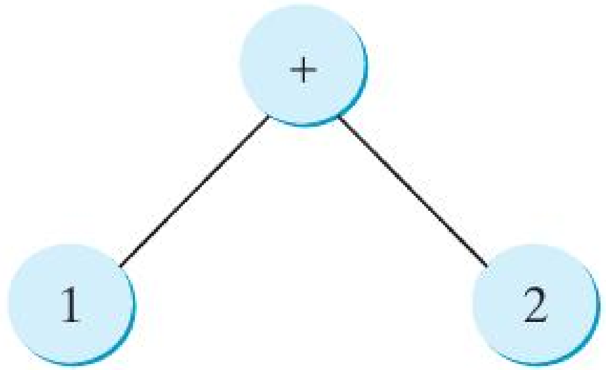


- In the example to the right the postorder traversal would be:

  45, 57, 55, 67, 107, 100, 60

Traversal Mnemonic

You can use the following tree to help remember inorder, postorder, and preorder.

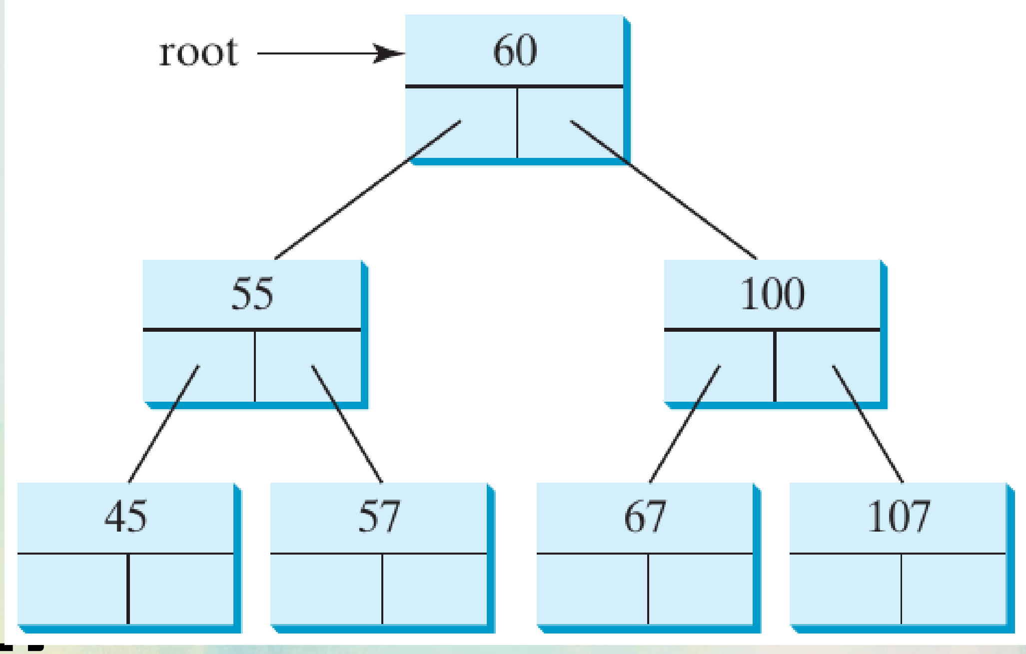


The inorder is 1 + 2, the postorder is 1 2 +, and the preorder is + 1 2.

Breadth-First Traversal - Binary Tree

- ***breadth-first traversal***: means that nodes are visited level by level.

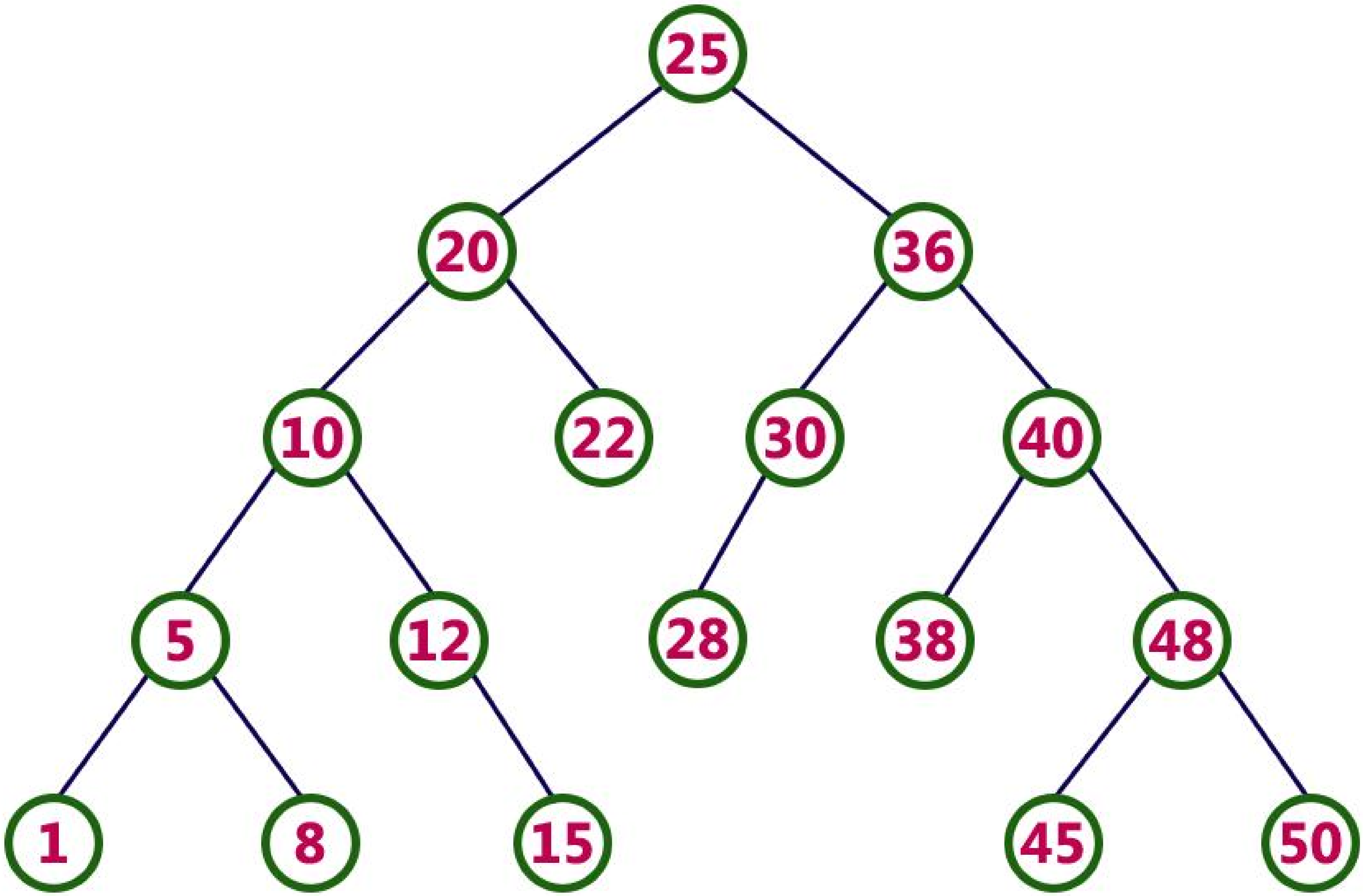
```
breadthfirst(root):
  create a Queue Q a
  add root to Q
  while Q not empty:
    node = Q.dequeue()
    visit node
    Q.enqueue(node.left)
    Q.enqueue(node.right)
```



- In the example to the right the breadth-first traversal would be:

  60, 55, 100, 45, 57, 67, 107

Traversal Practice



Self-Study

- Study the examples in section 8.4.5.

- I may ask about them on the quiz for this section.