Data Structures and Algorithm I

Lesson 5

Dr. Noel Fernando/Senior Lecturer

UCSC

Trees

- A *tree* data structure is a powerful tool for organizing data objects based on keys.
- It is equally useful for organizing multiple data objects in terms of hierarchical relationships (think of a ``family tree", where the children are grouped under their parents in the tree).

Trees

Further, Tree is one of the most important non-linear data structures in computing. It allows us to implement faster algorithms(compared with algorithms using linear data structures).

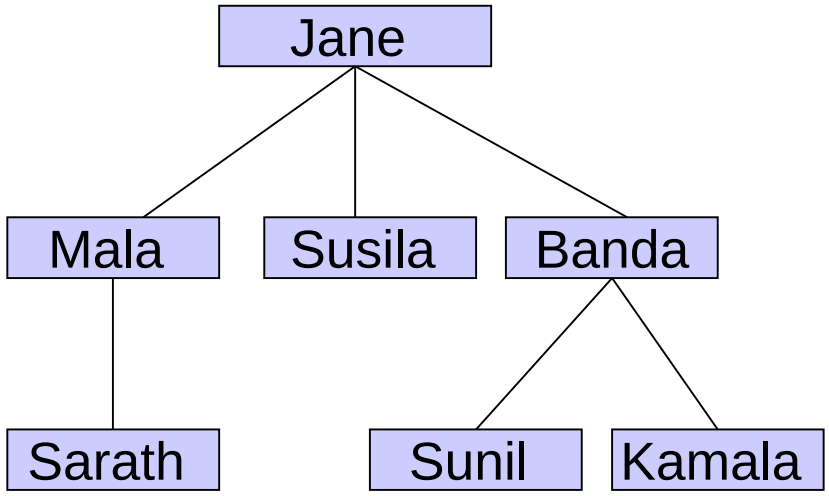
Application areas are :

- Almost all operating systems store files in trees or tree like structures.
- Compiler Design/Text processing
- Searching Algorithms
- Evaluating a mathematical expression.
- Analysis of electrical circuits.

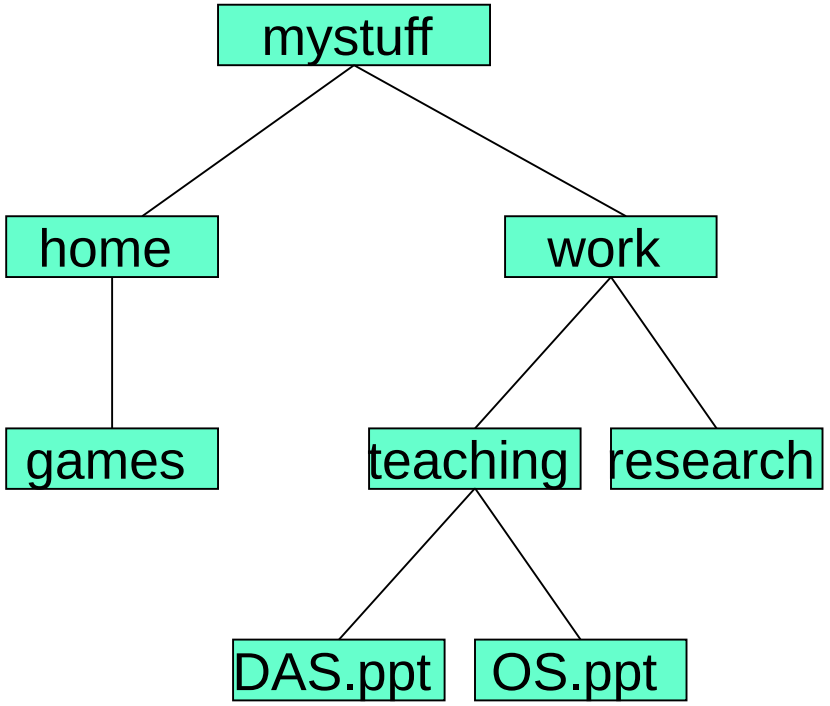
An application :File system

- There are many applications for trees. A popular one is the directory structure in many common operating systems, including VAX/VMX,Unix and DOS.

Trees



Jane's children and grandchildren

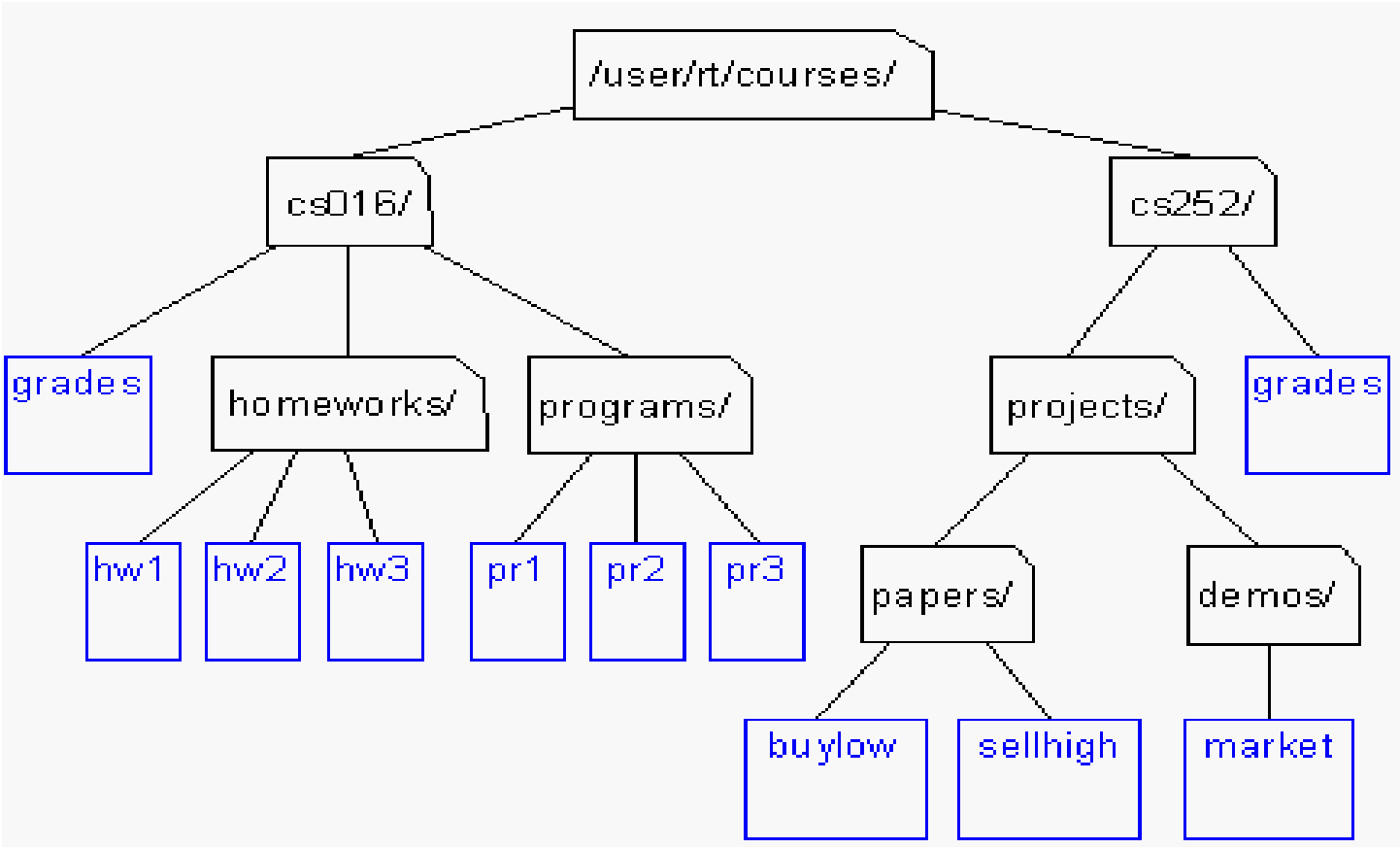


File organization in a computer

5

More Trees Examples

- Unix / Windows file structure



Tree Types

- Binary tree — each node has at most two children
- General tree — each node can have an arbitrary number of children.
- Binary search trees
- AVL trees

Tree Terminology and Basic Properties

Definition: A Tree is a set of nodes storing elements in a parent-child relationship with the following properties:

It has a special node called **root**.

- Each node different from the **root** has a **parent** node.
- **Parent** — the parent of a node is the node linked above it.

Definitions

- ✓**tree** - a non-empty collection of vertices & edges
- ✓**vertex (node)** - can have a name and carry other associated information
- ✓**path** - list of distinct vertices in which successive vertices are connected by edges

Definitions

- any two vertices must have one and only one path between them else its not a tree
- a tree with N nodes has N-1 edges

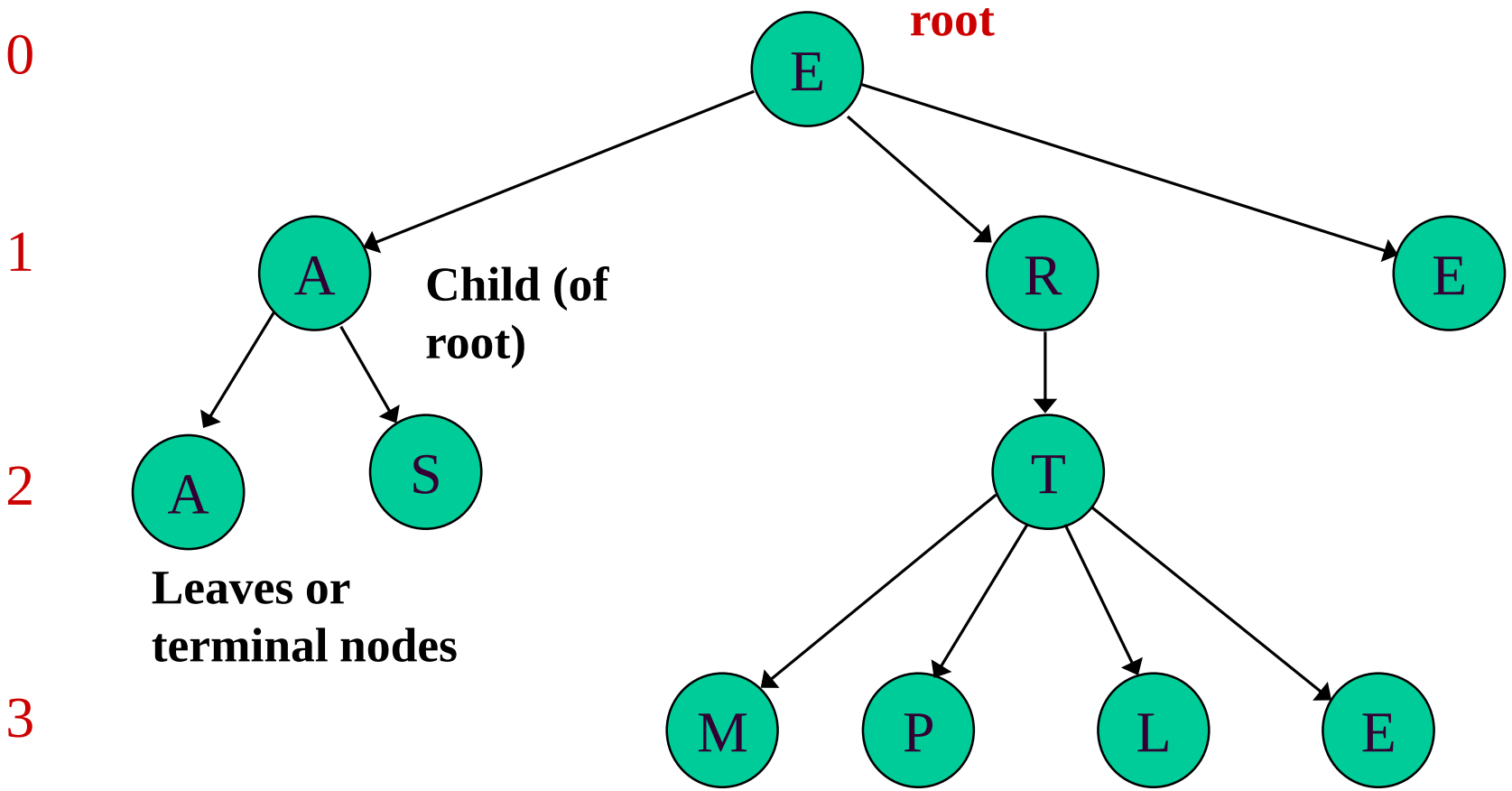
Definitions

- ✓**leaves** (terminal nodes) - have no children
- ✓**level** - the number of edges between this node and the root
- ✓**ordered tree** - where children's order is significant

Definitions

- ✓ **Depth of a node** - the length of the path from the root to that node
  - **root: depth 0**
- ✓ **Height of a node** - the length of the longest path from that node to a leaf
  - **any leaf: height 0**
- ✓ **Height of a tree:** The length of the longest path from the root to a leaf

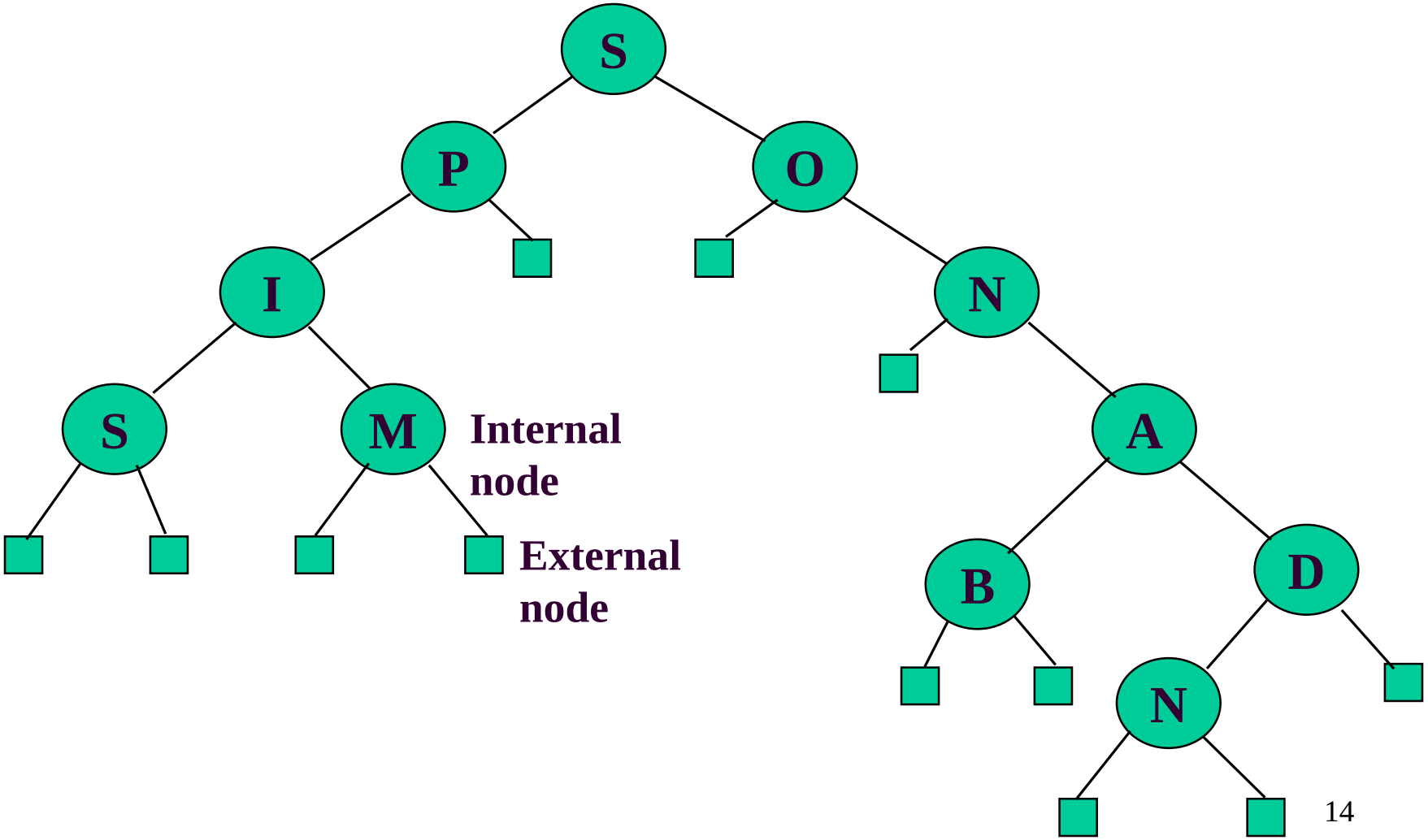
Trees - Example



Depth of T: 2

Height of T: 1

EXAMPLE



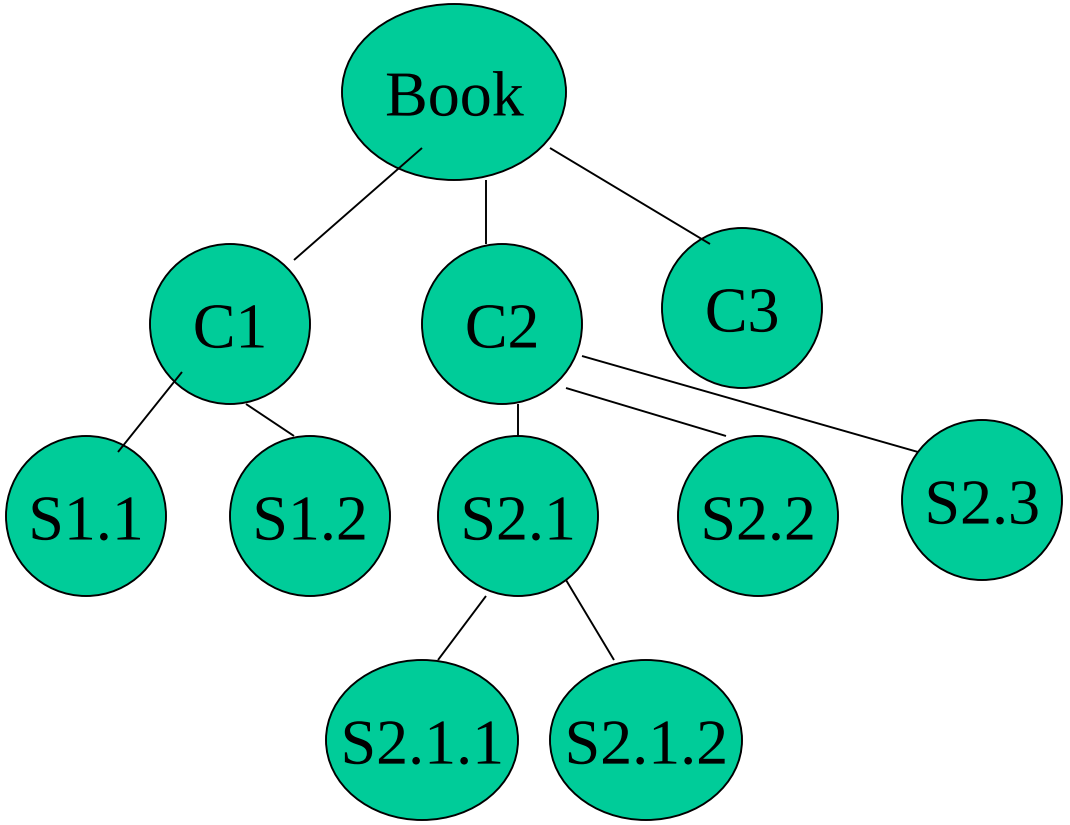
Tree Terminology and Basic Properties

- **Sibling**
- **Ancestor**
- **Descendant**
- **Leaf**.
- **Subtree**
- **Path of two nodes**
- **Length of a path**

Eg. A table of contents and its tree representation

Book

- C1
  - S1.1
  - S1.2
- C2
  - S2.1
    - S2.1.1
    - S2.1.2
  - S2.2
  - S2.3
- C3



- Degree : The number of sub tree of a node is called its degree.
- Eg. Degree of book → 3, C1→2,C3→0
- Nodes that have degree 0 is called Leaf or Terminal node.Other nodes called non-terminal nodes. Eg.Leaf nodes :C3,S1.1,S1.2 etc.
- Book is said to be the father (parent) of C1,C2,C3 and C1,C2,C3are said to be sons (children) of book.
- Children of the same parent are said to be siblings.Eg.C1,C2,C3 are siblings (Brothers)
- Length : The length of a path is one less than the number of nodes in the path.(Eg path from book to s1.1=3-1=2

- If there is a path from node a to node b , then **a** is an ancestor of b and b is descendent of a.
- In above example, the ancestor of S2.1are itself,C2 and book, while it descendent are itself, S2.1.1 and S2.1.2.
- An ancestor or descendent of a node, other than the node itself is called a proper ancestor or proper descendent.
- Height of a tree — the maximum depth of a leaf node. ..[In above example height=3]
- **Depth of a node** — the length of the path between the root and the node.[In above example node C1 has **Depth** 1, node S2.1.2 has **Depth** 3.etc.]

General Trees.

Trees can be defined in two ways :

- Recursive
- Non- recursive

One natural way to define a tree is recursively

Definition of Tree -general tree

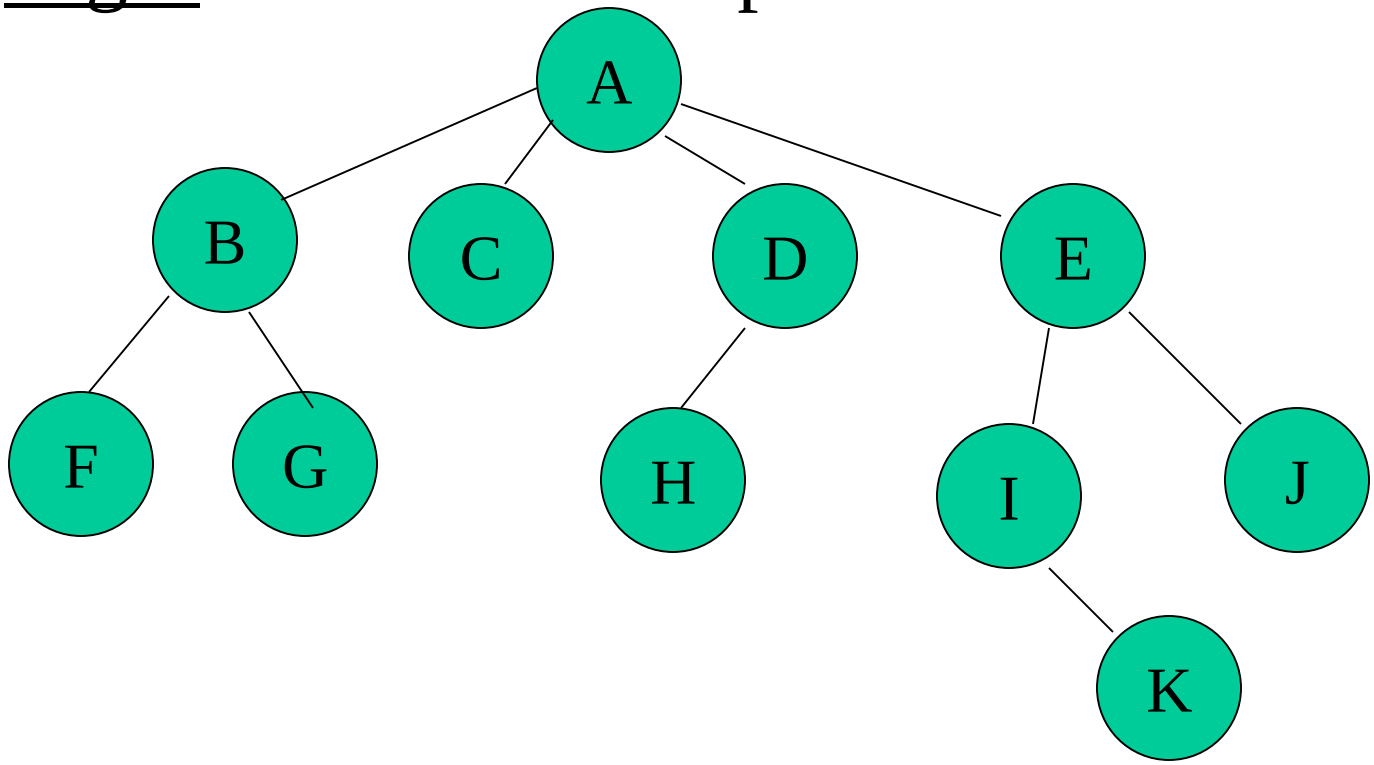
- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets T_1, ..., T_n, where each of these sets is a tree.
- We call T_1, ..., T_n the subtrees of the root.

General trees-Definition

- A tree is a collection of nodes. The collection can be empty; otherwise a tree consists of a distinguish node r, called root, and zero or more non-empty (sub)trees $T_1, T_2, T_3, \ldots T_K$. Each of whose roots are connected by a directed edge from r.

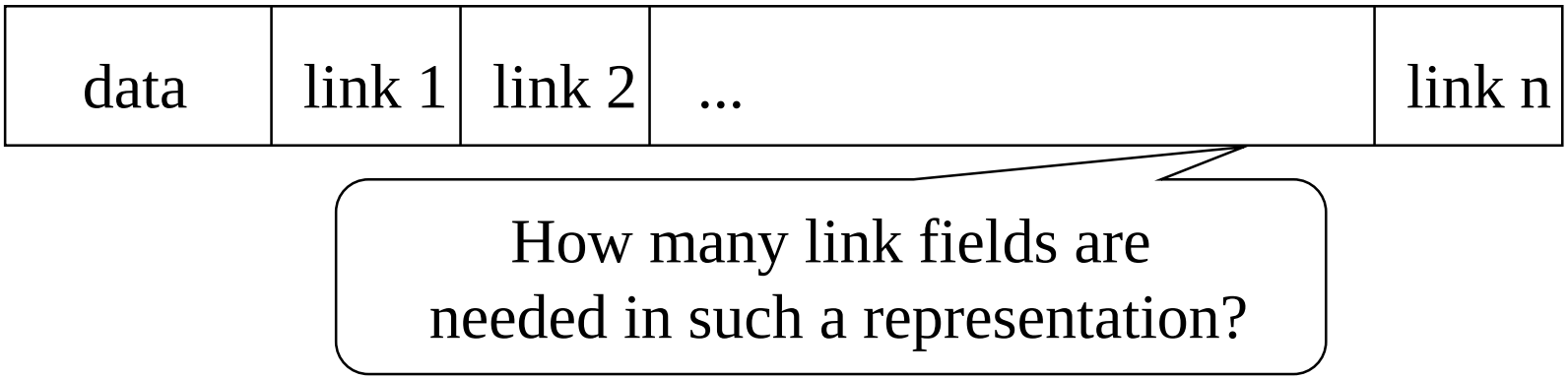
General trees-Definition

- A tree consists of set of nodes and set of edges that connected pair of nodes.



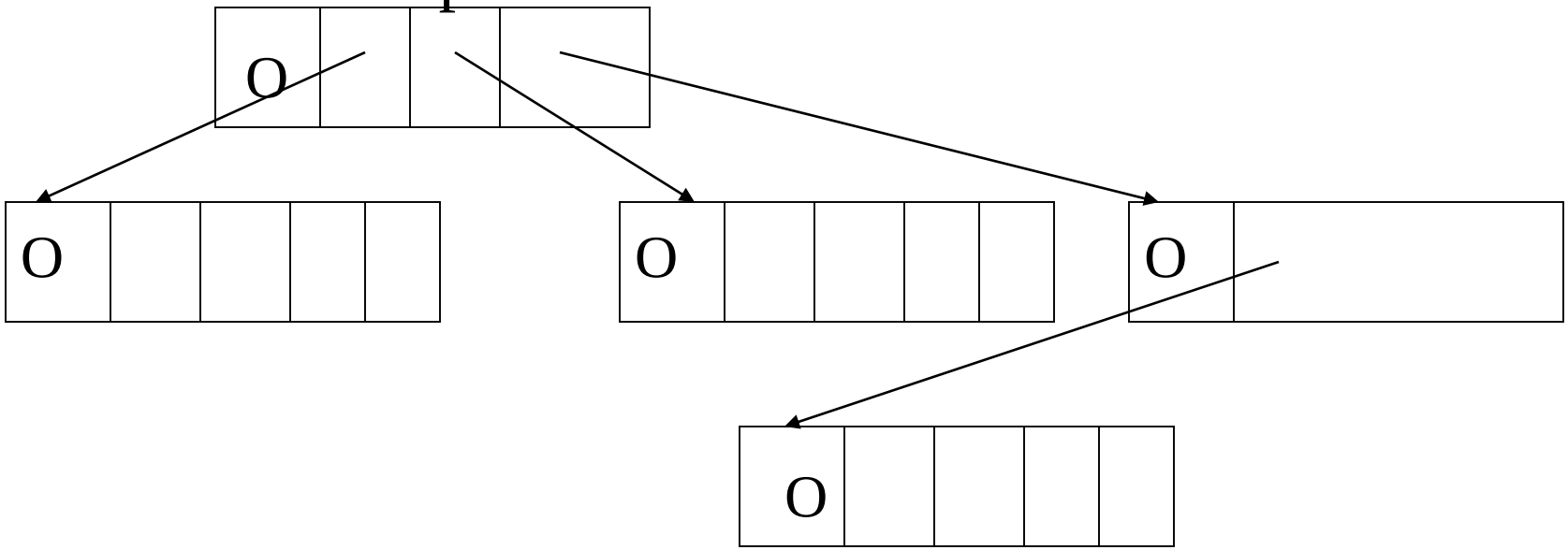
Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees



A Tree Node

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



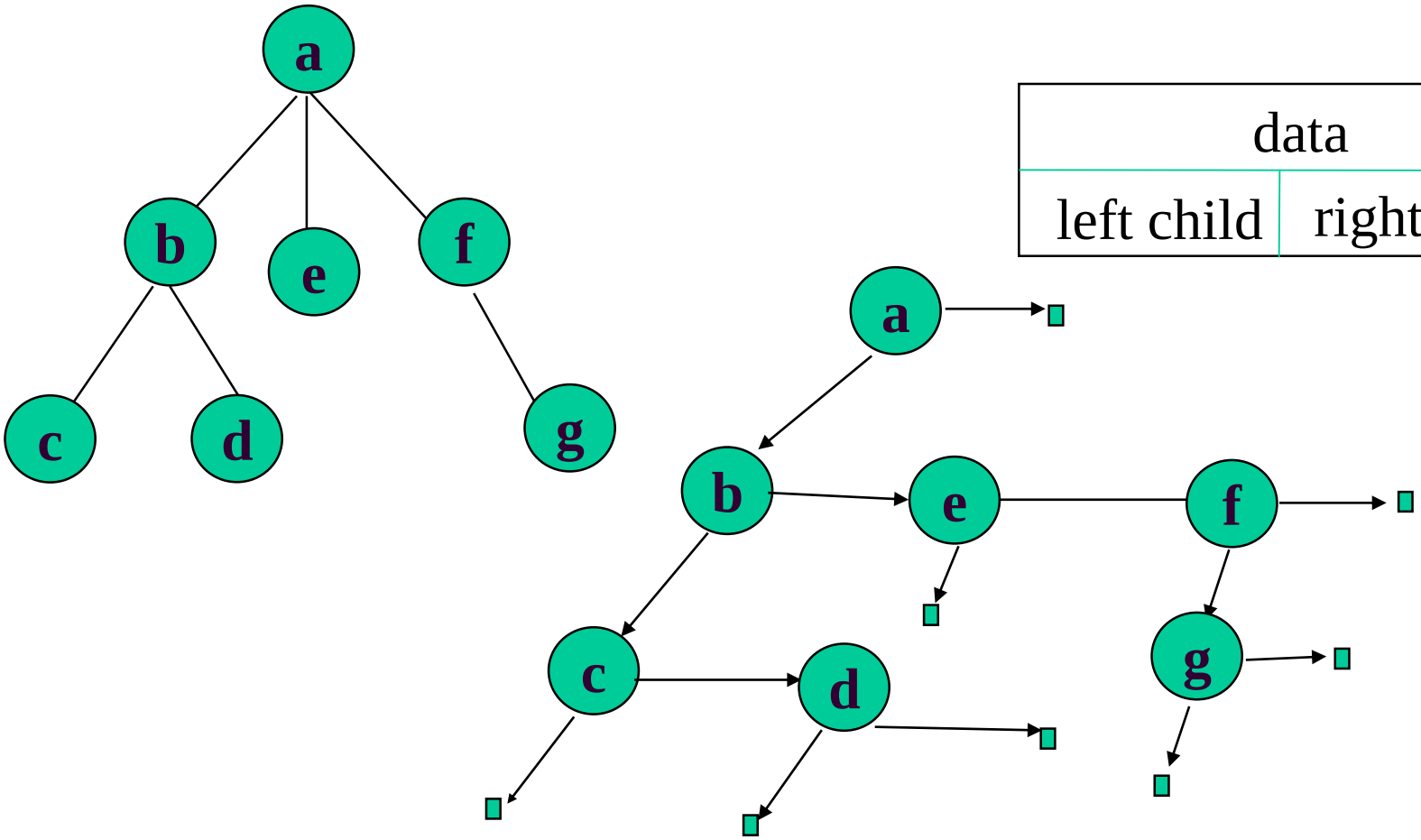
Tree - Implementation

- One way to implement a tree would be to have in each node a reference to each child of the node in addition to its data.
- However, the number of children per node can vary so greatly and is not known in advanced.
- It might be infeasible to make the children direct links in the data structure.
- These would be too much wasted space.

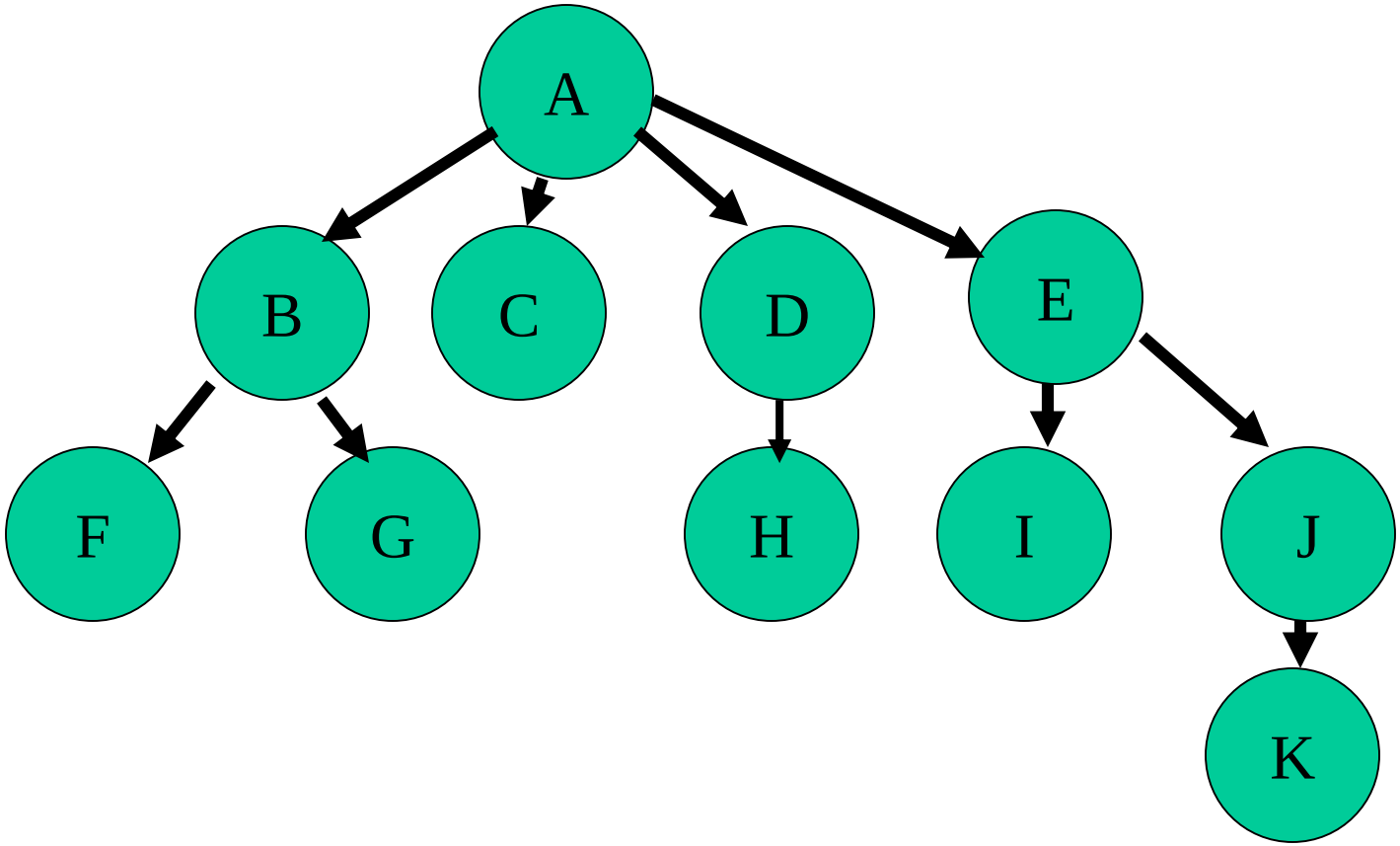
The solution is simple :

- Keep the children of each node in a linked list of tree nodes. Thus each node keeps two references : one to its leftmost child and other one for its right sibling.

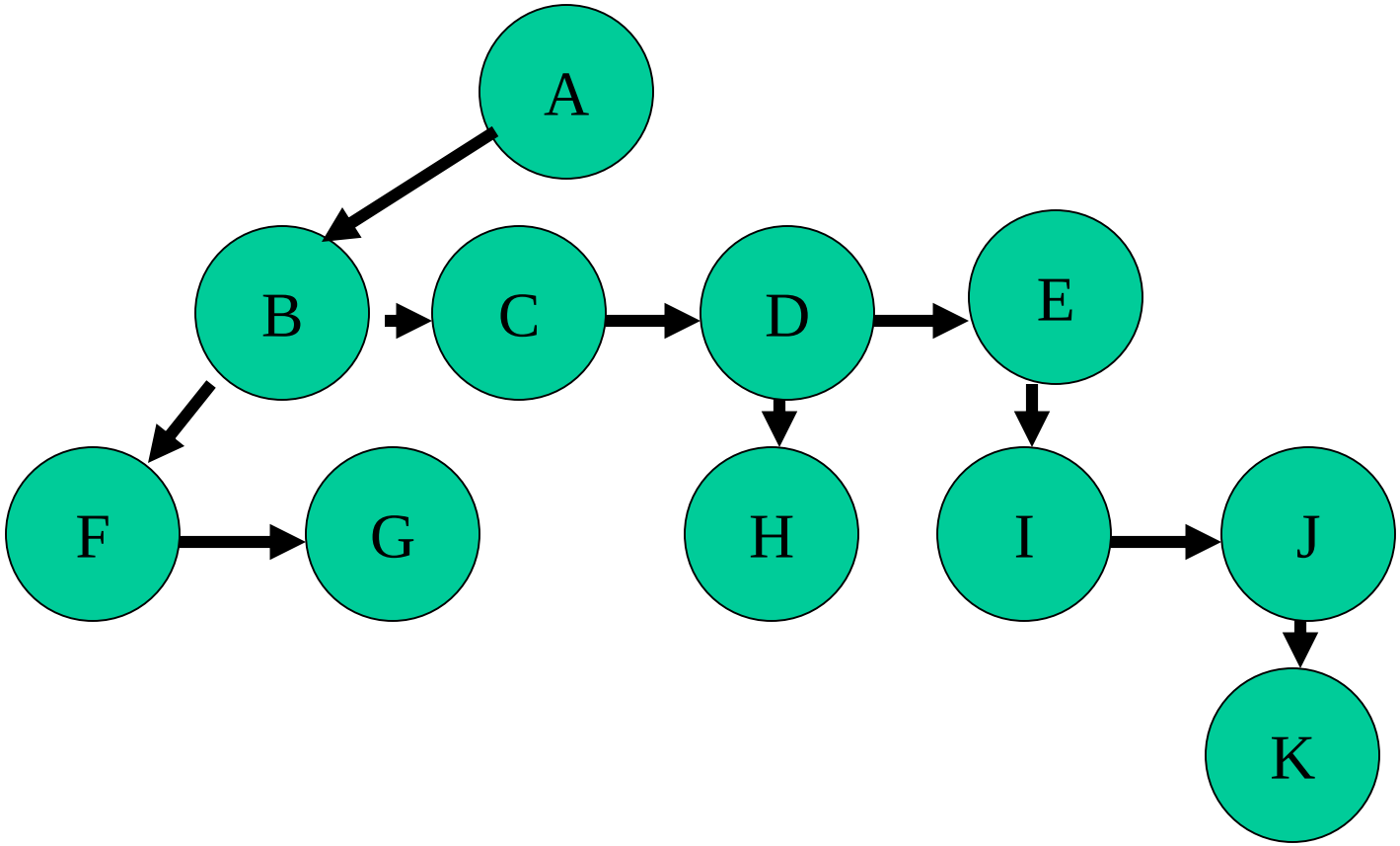
Example



Left Child - Right Sibling representation of a tree

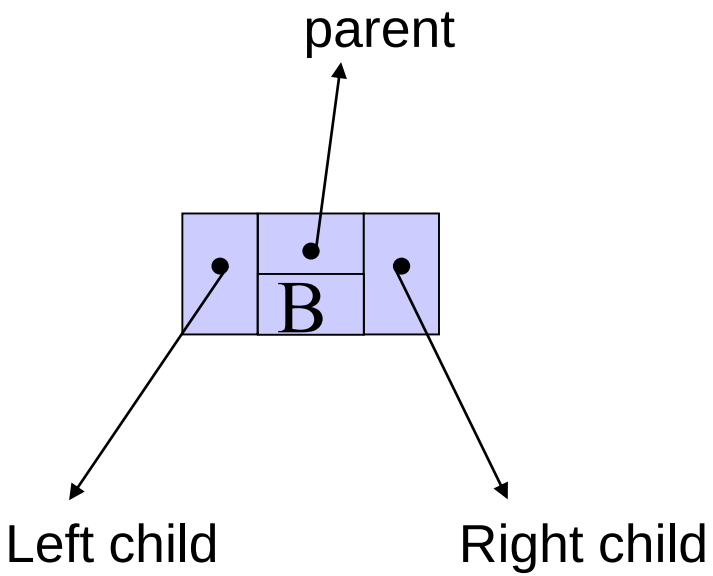


Left most Child - Right Sibling representation of a tree

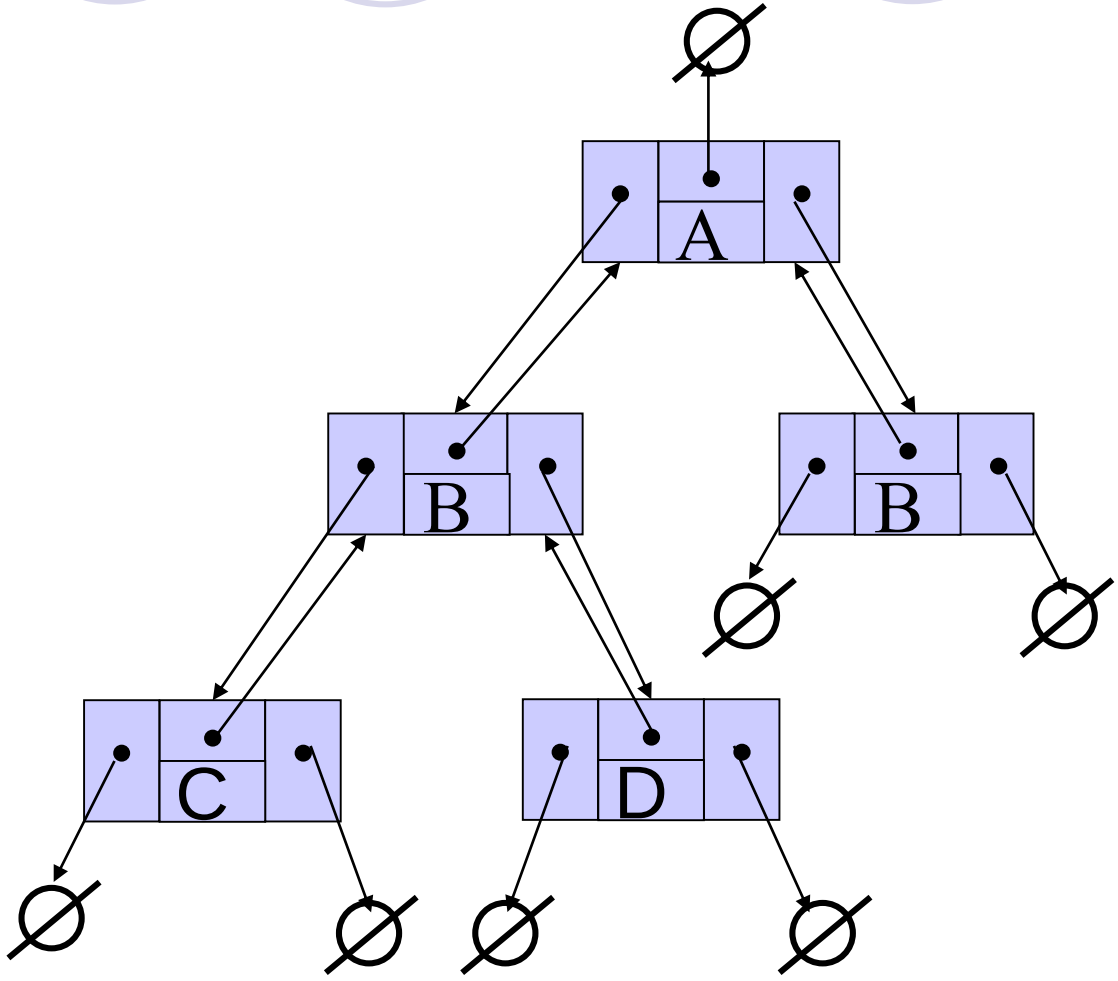


7.3.2 A Linked Structure for Binary Trees

- We represent each node of a binary tree by an object which stores
  - Element
  - References to its parent and children nodes



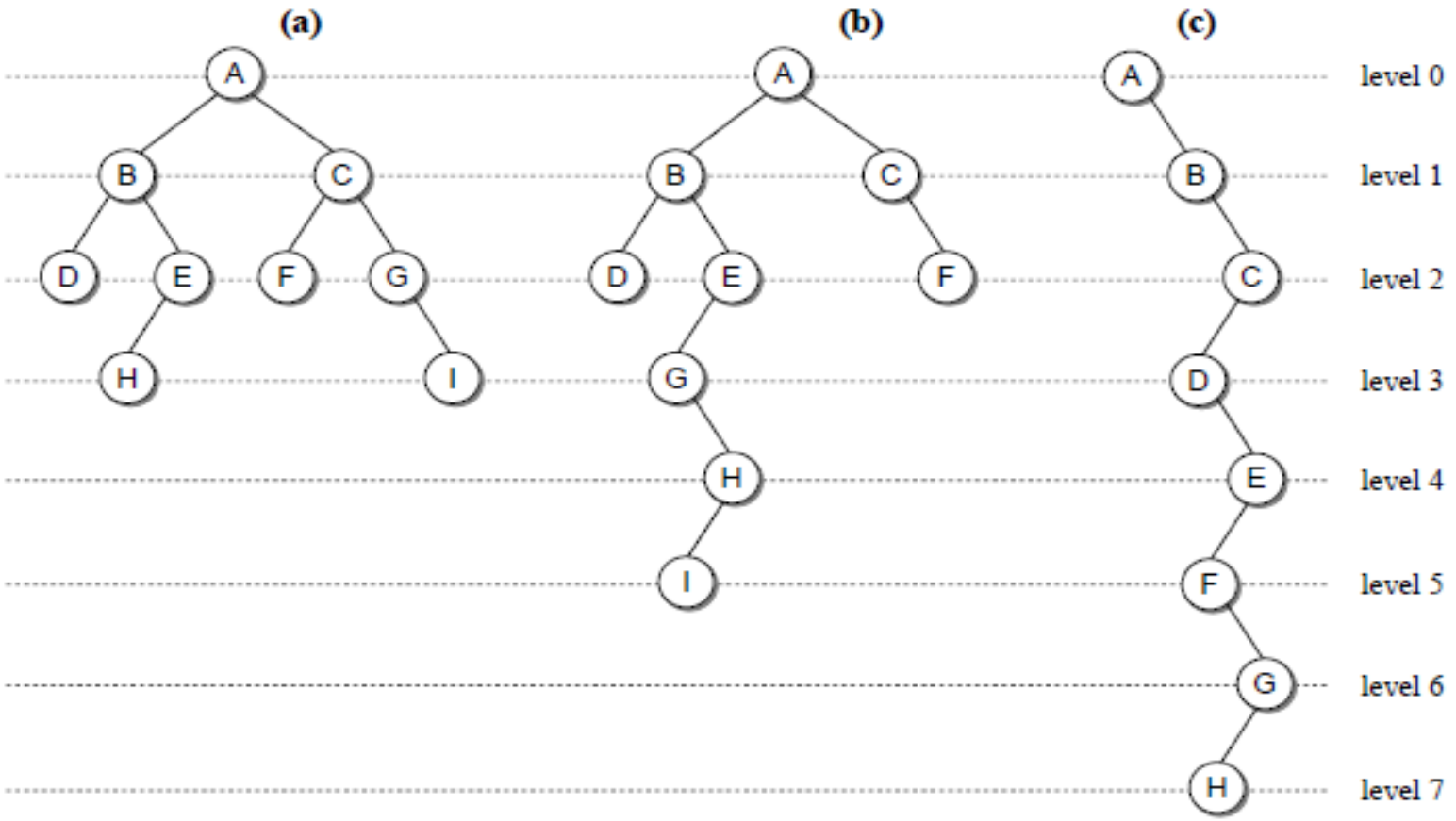
7.3.2 A Linked Structure for Binary Trees



Properties of Binary Tress

- Binary trees come in many deferent shapes and sizes. The shapes vary depending on the number of nodes and how the nodes are linked.

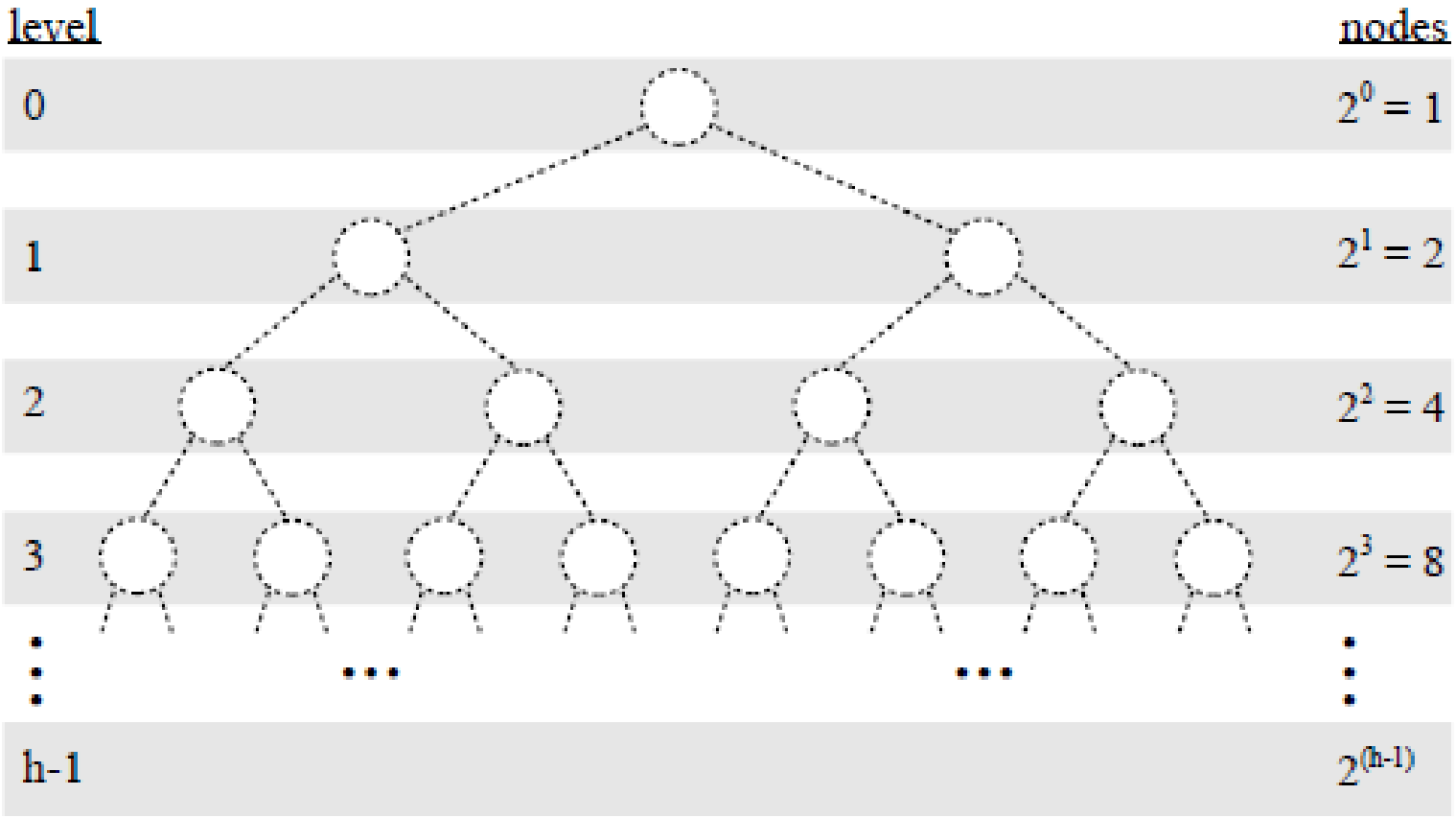
Three different arrangements of nine nodes in a binary tree.



Three different arrangements of nine nodes in a binary tree (cont..)

- Consider node G in the three trees of above Figure. In tree (a), G has a depth of 2, in tree (b) it has a depth of 3, and in (c) its depth is 6.
- The height of a binary tree is the number of levels in the tree. For example, the three binary trees in the above Figure have different heights: (a) has a height of 3, (b) has a height of 5, and (c) has a height of 7.

Possible slots for the placement of nodes in a binary tree



Data structure definition

```
1 # The storage class for creating binary tree nodes.
2 class _BinTreeNode :
3   def __init__( self, data ):
4     self.data = data
5     self.left = None
6     self.right = None
```

The binary tree node class

Binary trees

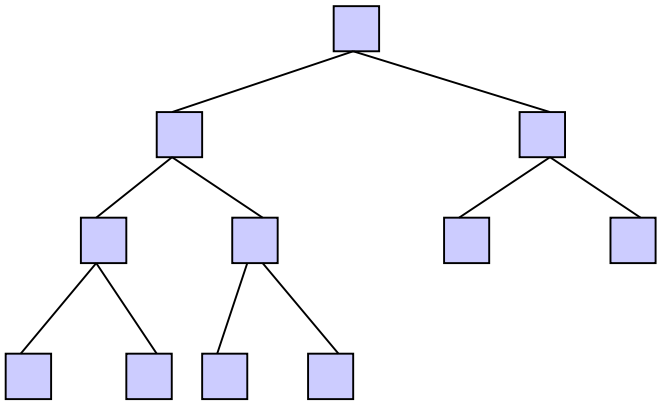
- A binary tree is a tree in which no nodes can have more than two children.
- The recursive definition is that a binary tree is either empty or consists of a root, a left tree, and a right tree.
- The left and right trees may themselves be empty; thus a node with one child could have a left or right child. We use the recursive definition several times in the design of binary tree algorithms.

Tree Terminology and Basic Properties

Complete binary tree

A binary tree is complete if every level except the deepest contains as many nodes as possible, and all the nodes at the deepest level are as far left as possible

Tree Terminology and Basic Properties

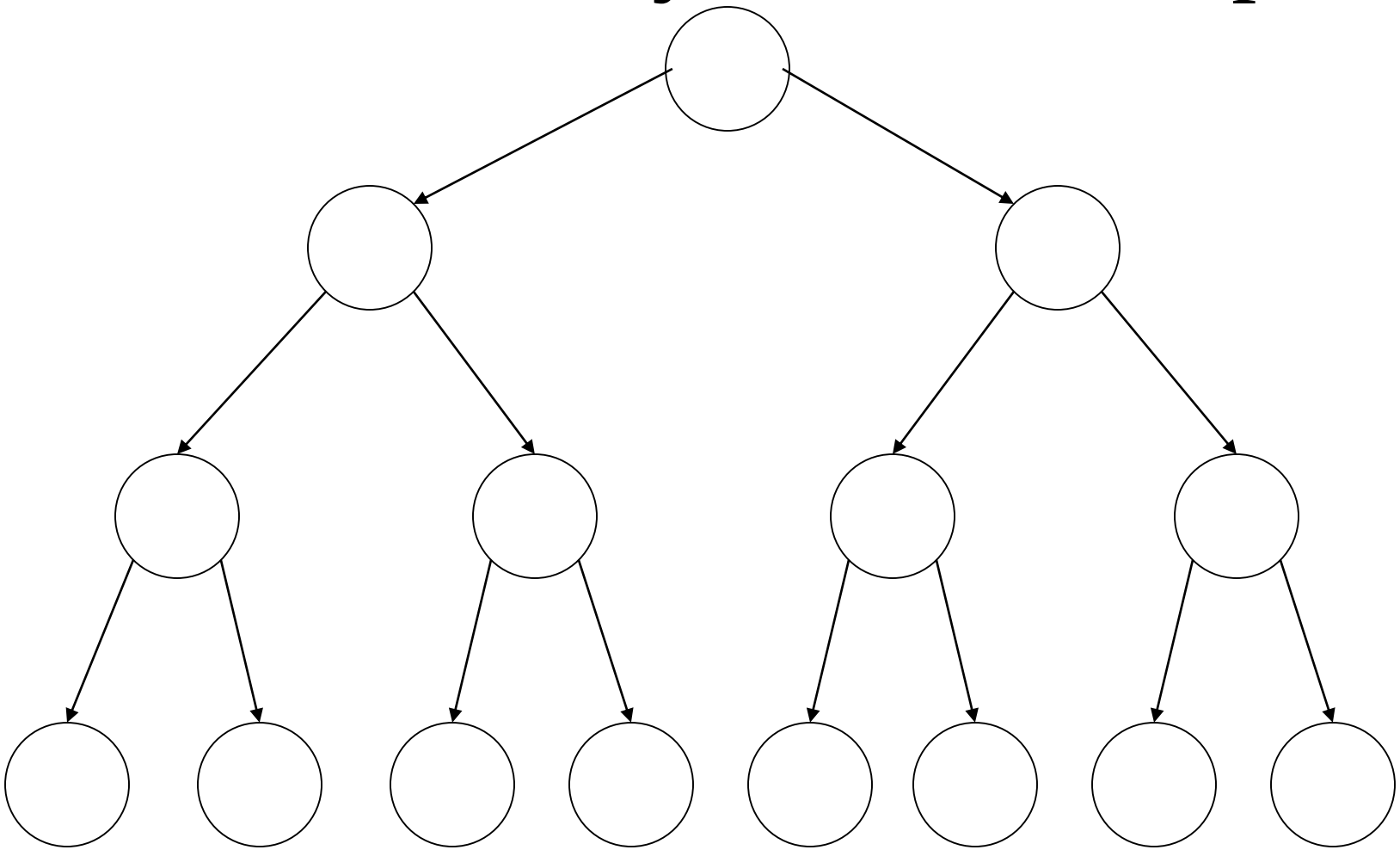


A complete binary tree

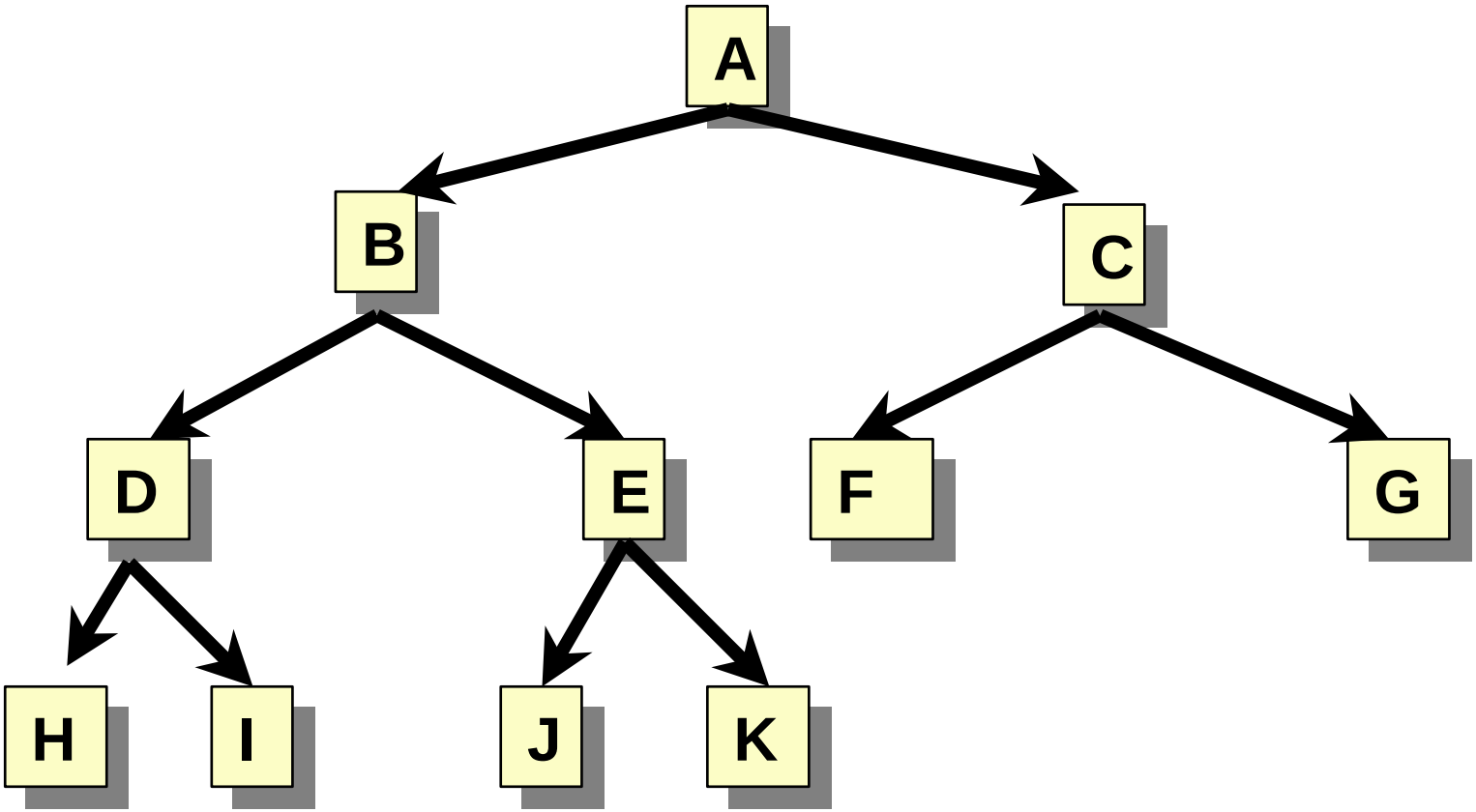
Properties of Binary Trees

- A binary tree is a **full** binary tree if and only if:
  - Each non leaf node has exactly two child nodes
  - All leaf nodes have identical path length
- It is called **full** since all possible node slots are occupied

A Full Binary Tree - Example



Complete Binary Trees - Example



Tree Terminology and Basic Properties

- Properties of Binary Trees
  - Let T be a complete binary tree with *n* nodes, and let *h* denote the height of T, then T has the following properties:
  -

	At least	At most
The number of leaf nodes	$h+1$	2^h
The number of non-leaf nodes	h	2^h-1
The total number of nodes	$2h+1$	$2^{h+1}-1$
The height of the tree *h*	$\log(n+1)-1$	$(n-1)/2$

Properties of binary trees.

- If a binary tree contains m nodes at level L, then it contains at most 2m nodes at level L+1.
- A binary tree can contain at most 2^L nodes at L

At level 0 B-tree can contain at most $1= 2^0$ nodes

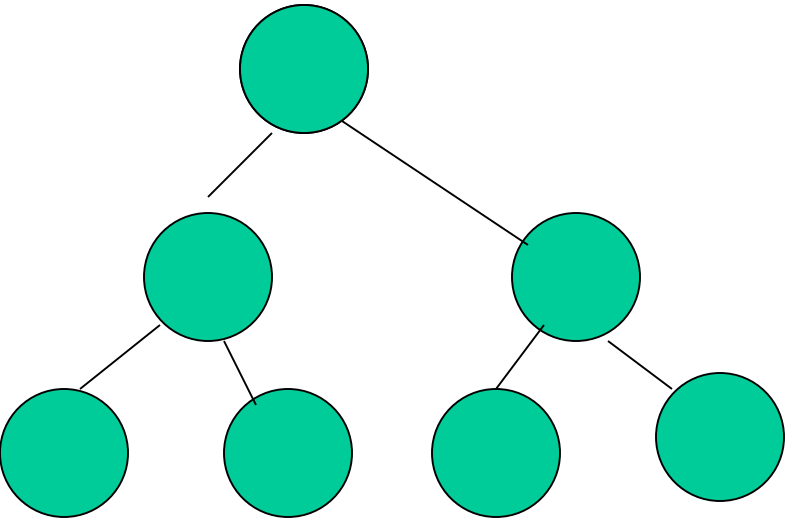
At level 1 B-tree can contain at most $2= 2^1$ nodes

At level 2 B-tree can contain at most $4= 2^2$ nodes

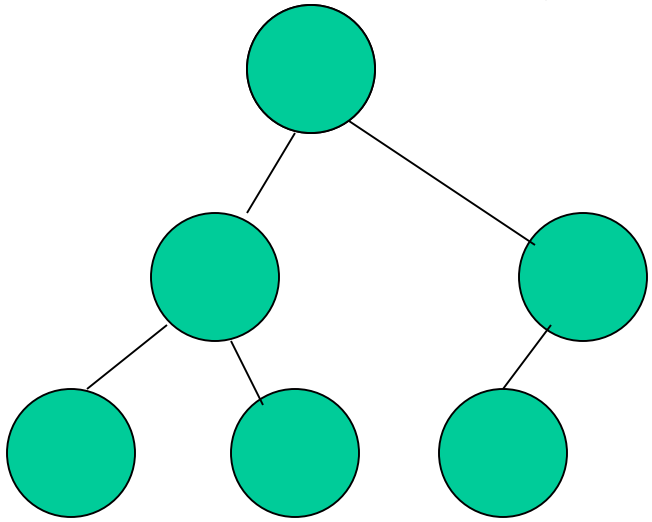
At level L B-tree can contain at most → 2^L nodes

Full B-tree

- A full B-tree of depth d is the B-tree that contains exactly 2^L nodes at each level between 0 and d (or 2^d nodes at d)



full B-tree



Not a full B-tree

The total number of nodes (Tn) in a full binary tree of depth d is $2^{d+1}-1$

- $Tn=2^0+2^1+2^2+\ldots\ldots 2^d\ldots..(1)$
- $2Tn=2^1+2^2+\ldots\ldots\ 2^{d+1}\ldots.(2)$
- (2)-(1)→
- $Tn=2^{d+1}-1$

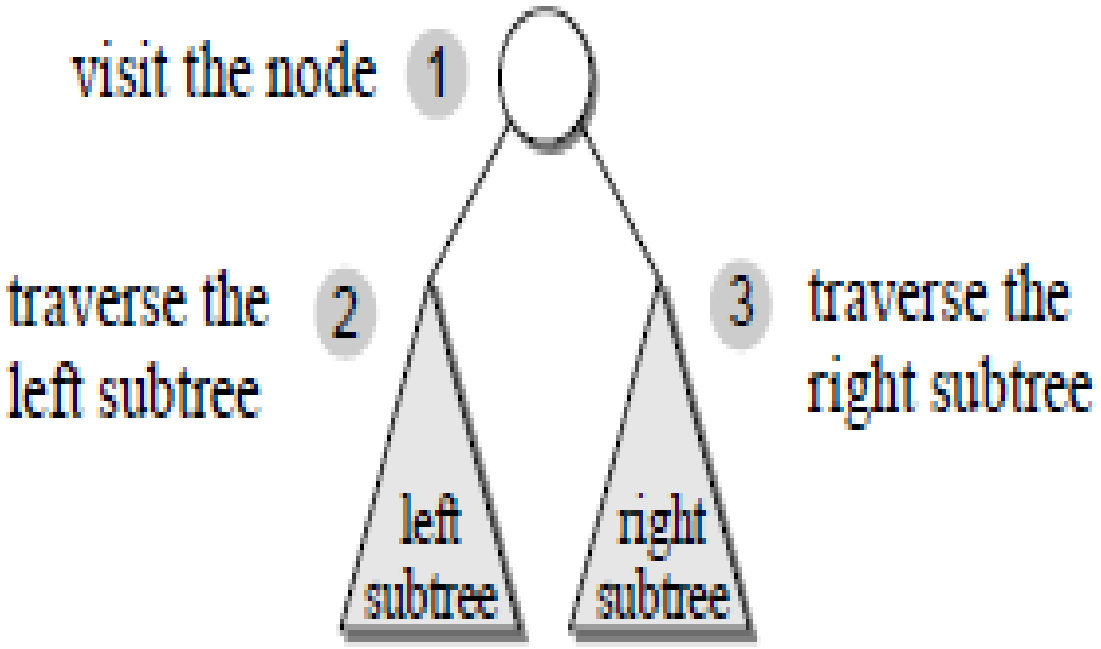
7.4 Traversals of Binary Trees

- A traversal of a tree is a systematic way of accessing or “visiting” all the nodes in the tree.
- There are three basic traversal schemes:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

7.4.1 Pre-Order Traversal

- A pre-order traversal has three steps for a nonempty tree:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  -

Pre-order traversal

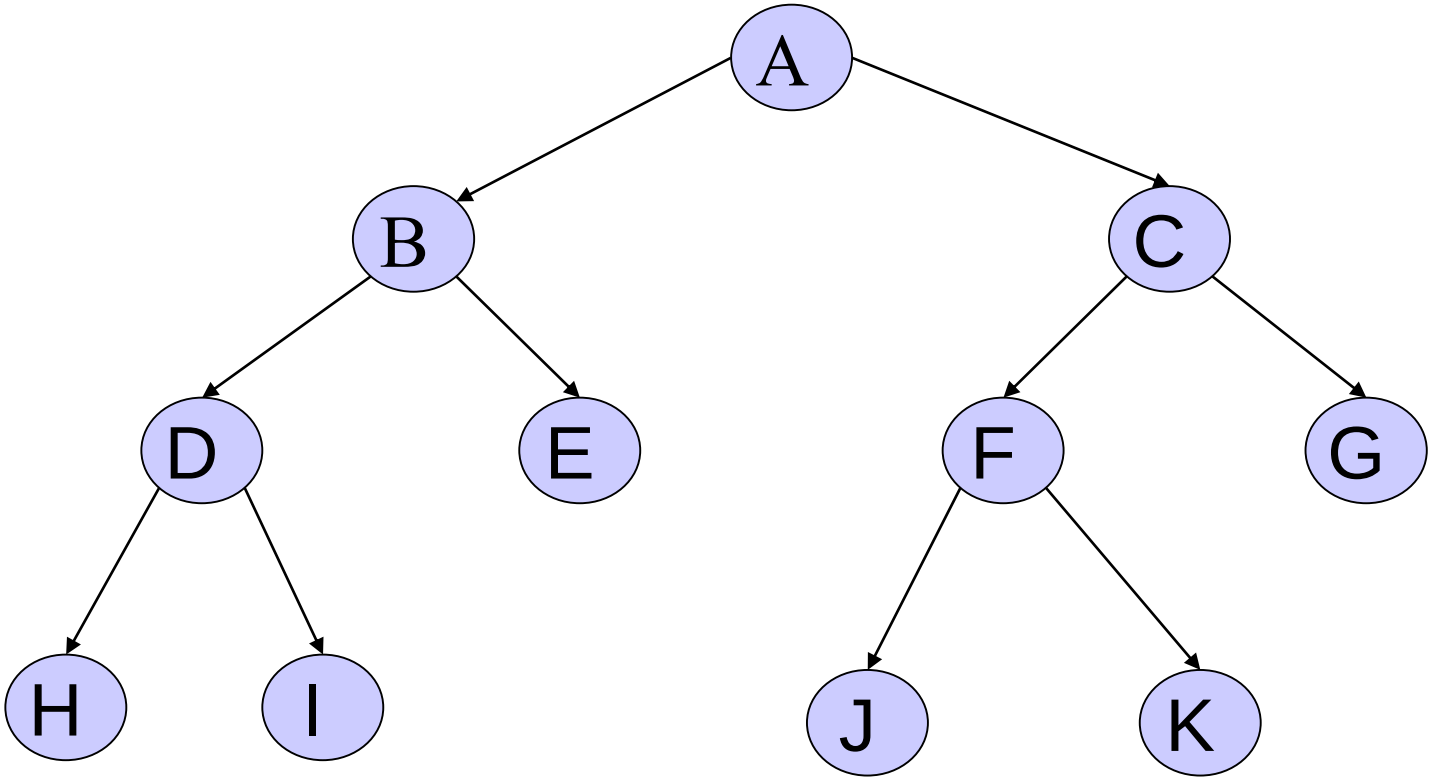


Pre-order traversal

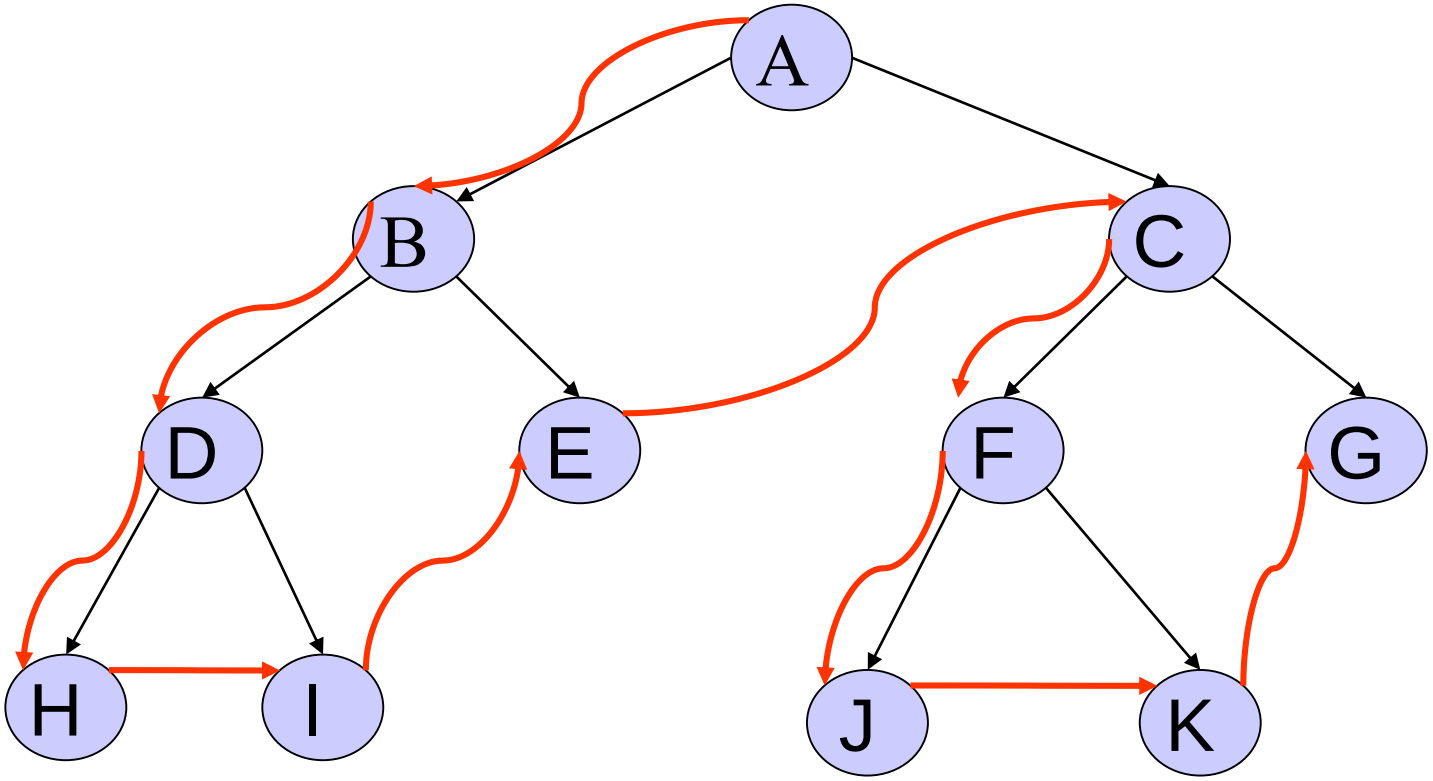
To traverse a non-empty binary tree in pre-order (also known as depth first order), we perform the following operations.

- Visit the root (or print the root)
- Traverse the left in pre-order (Recursive)
- Traverse the right tree in pre-order (Recursive)

7.4.1 Pre-Order Traversal

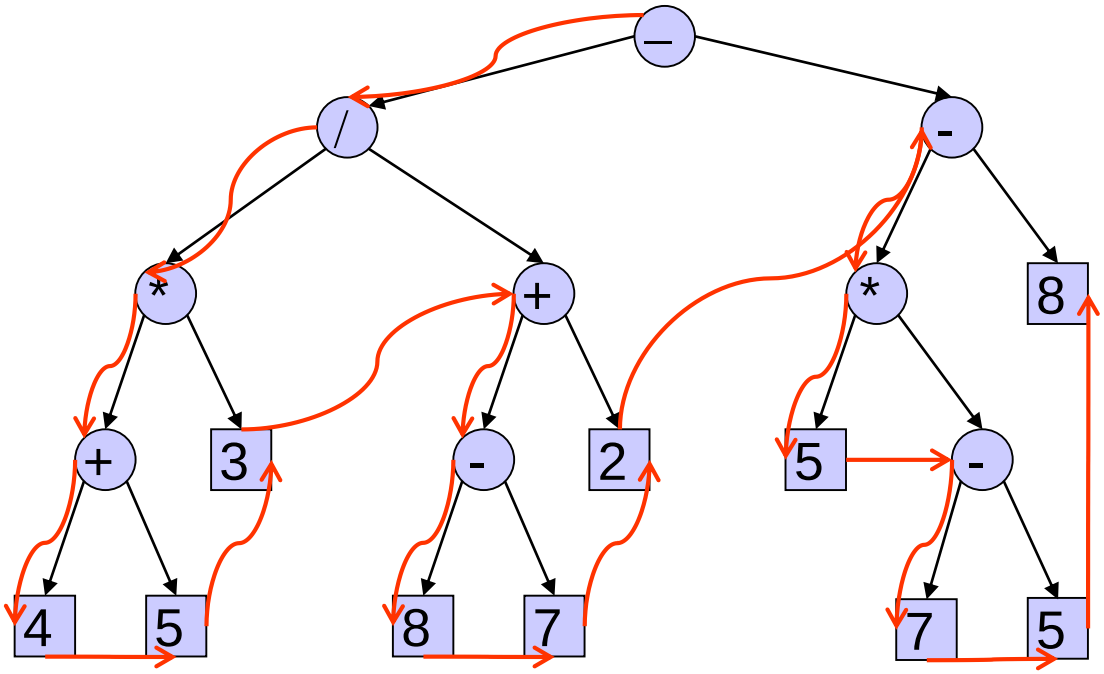


7.4.1 Pre-Order Traversal



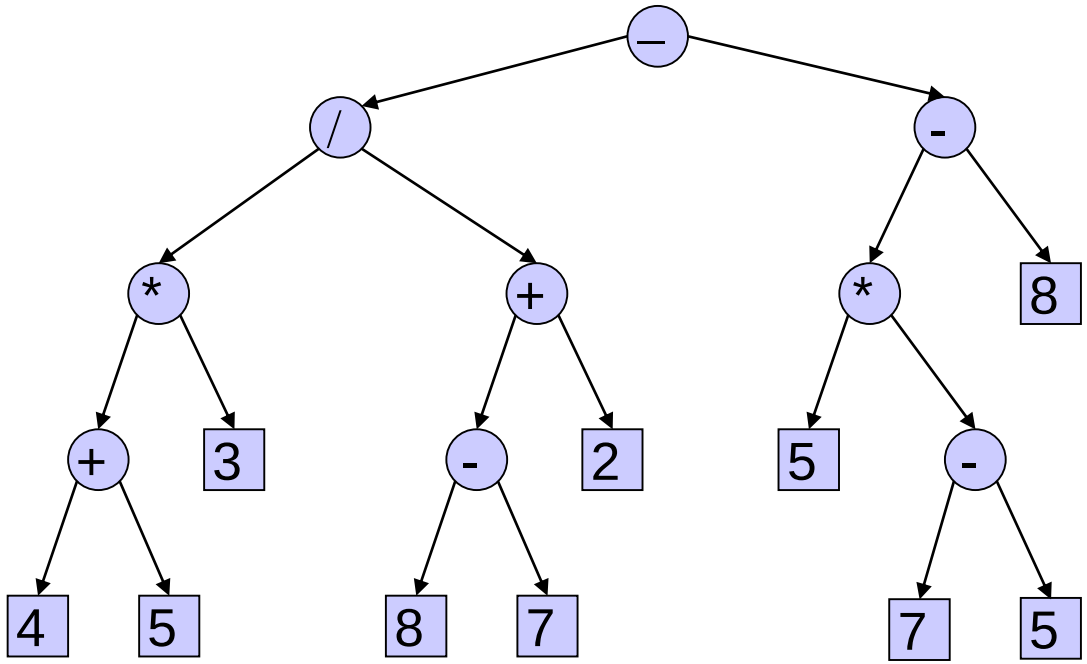
7.4.1 Pre-Order Traversal

- Using pre-order traversal of a binary tree to solve the expression evaluation problem.

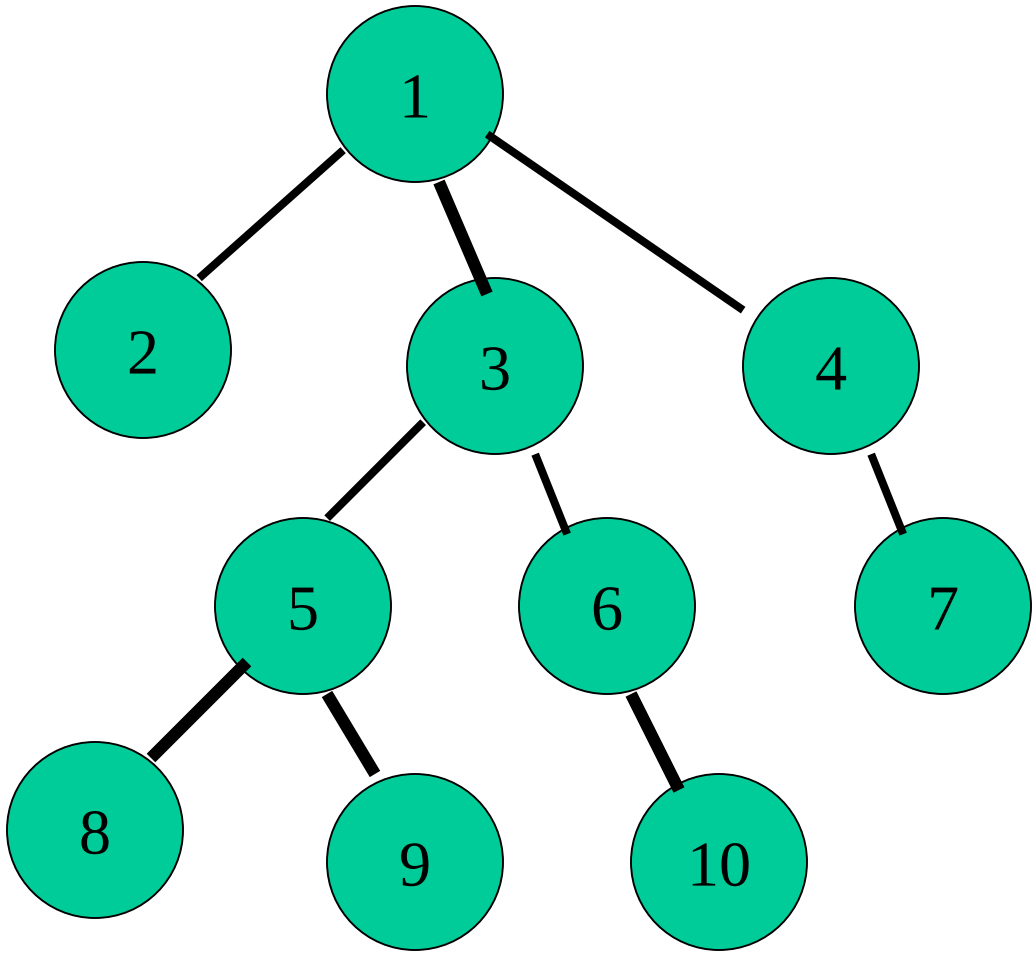


7.4.1 Pre-Order Traversal

- Using pre-order traversal of a binary tree to solve the expression evaluation problem.



Pre-order traversal



Pre-order list –1,2,3,5,8,9,6,10,4,7

7.4.2 In-Order Traversal

- An in-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.
  -
-

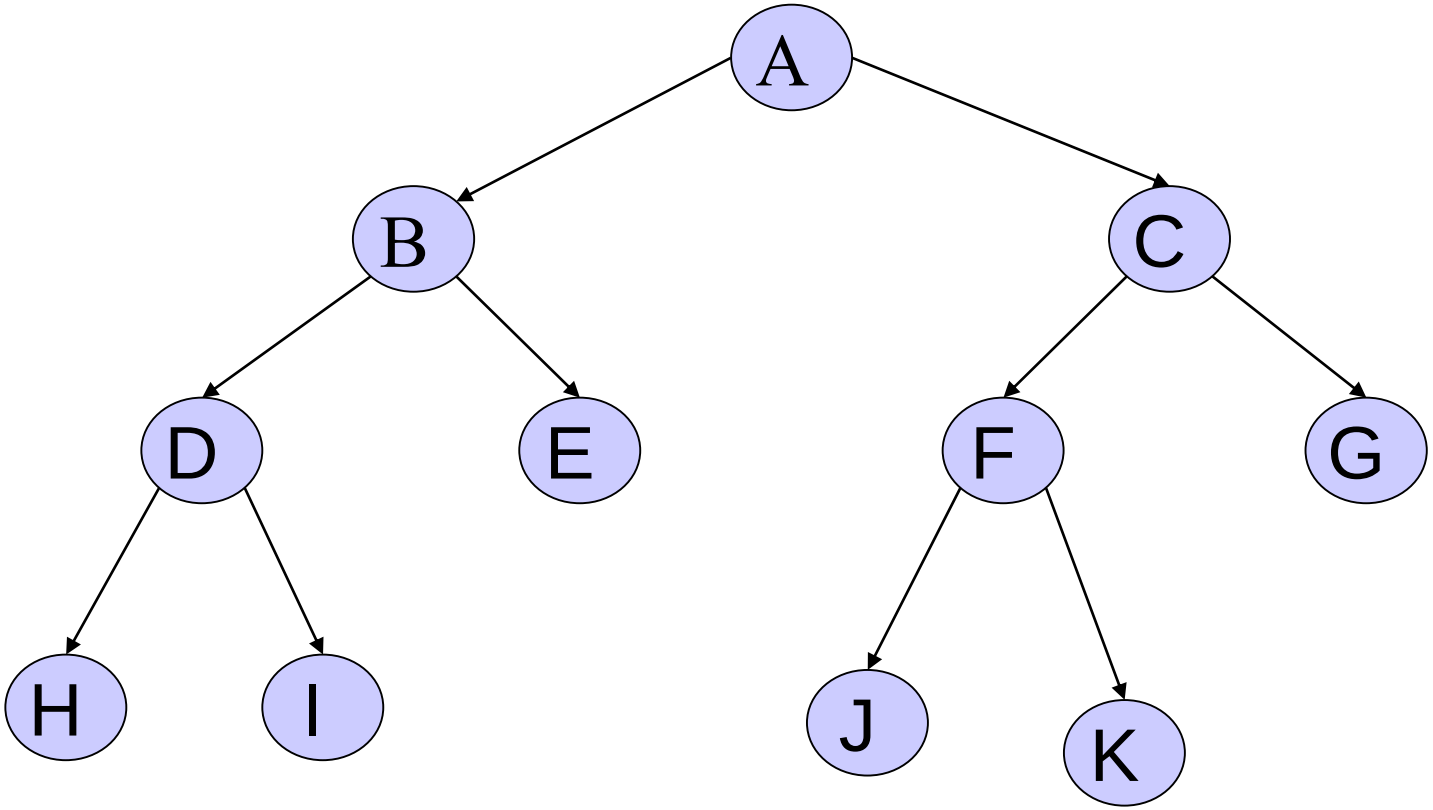
In-order traversal

The in-order listing of the nodes of T is the nodes of T_1 in in-order, followed by n,followed by the nodes $T_1, T_2, \ldots T_n$, each group of nodes in in-order.

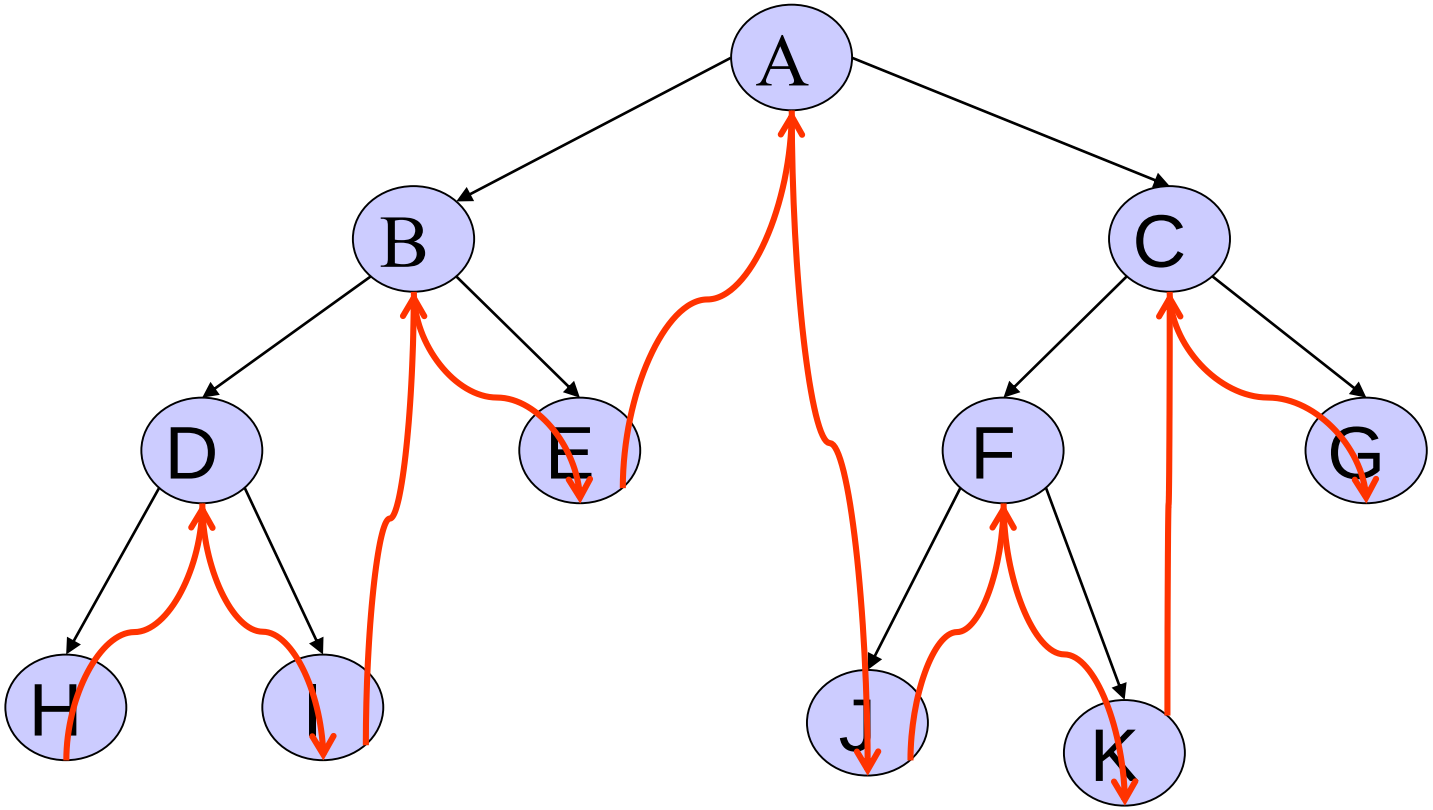
Algorithm :

- Traverse the left-subtree in in-order
- Visit the root
- Traverse the right-subtree in in-order.

7.4.2 In-Order Traversal

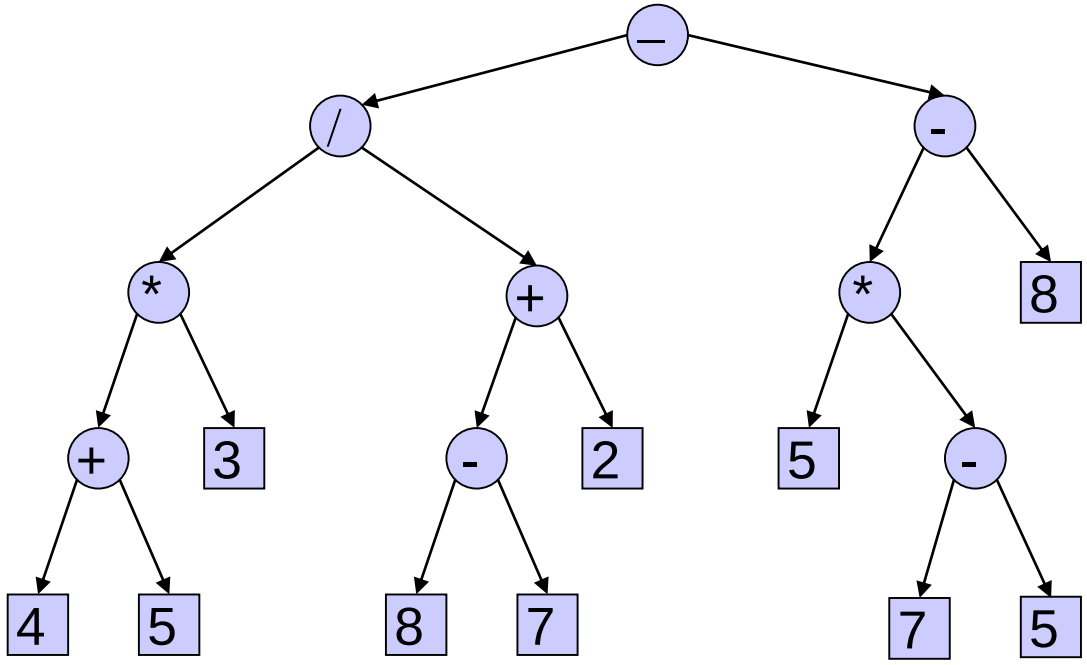


7.4.2 In-Order Traversal



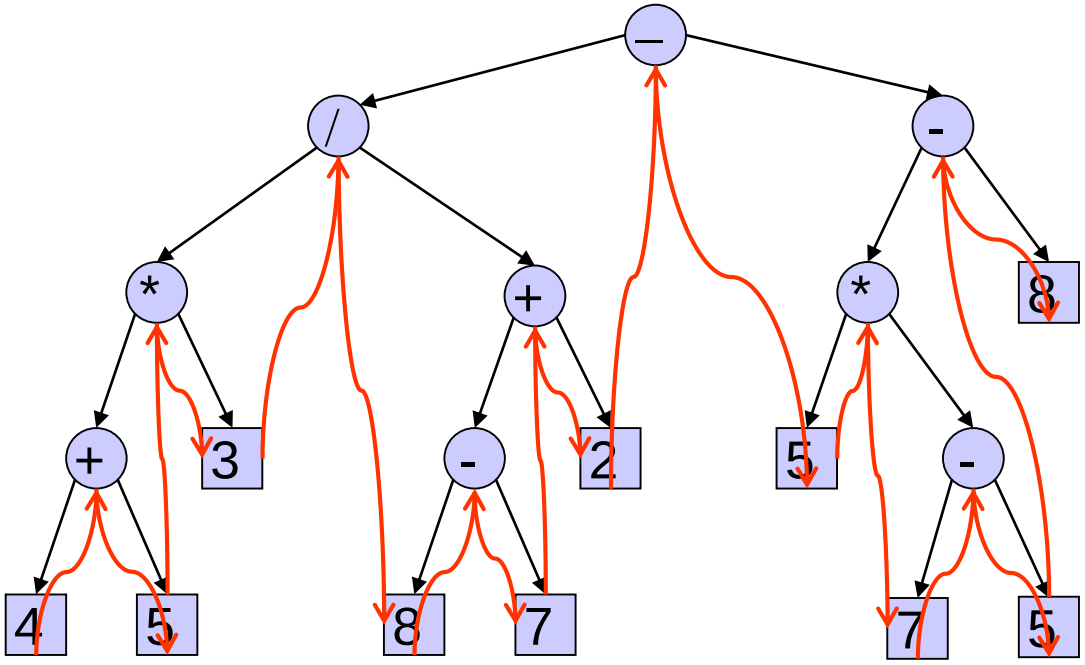
7.4.2 In-Order Traversal

- Using in-order traversal of a binary tree to solve the expression evaluation problem.

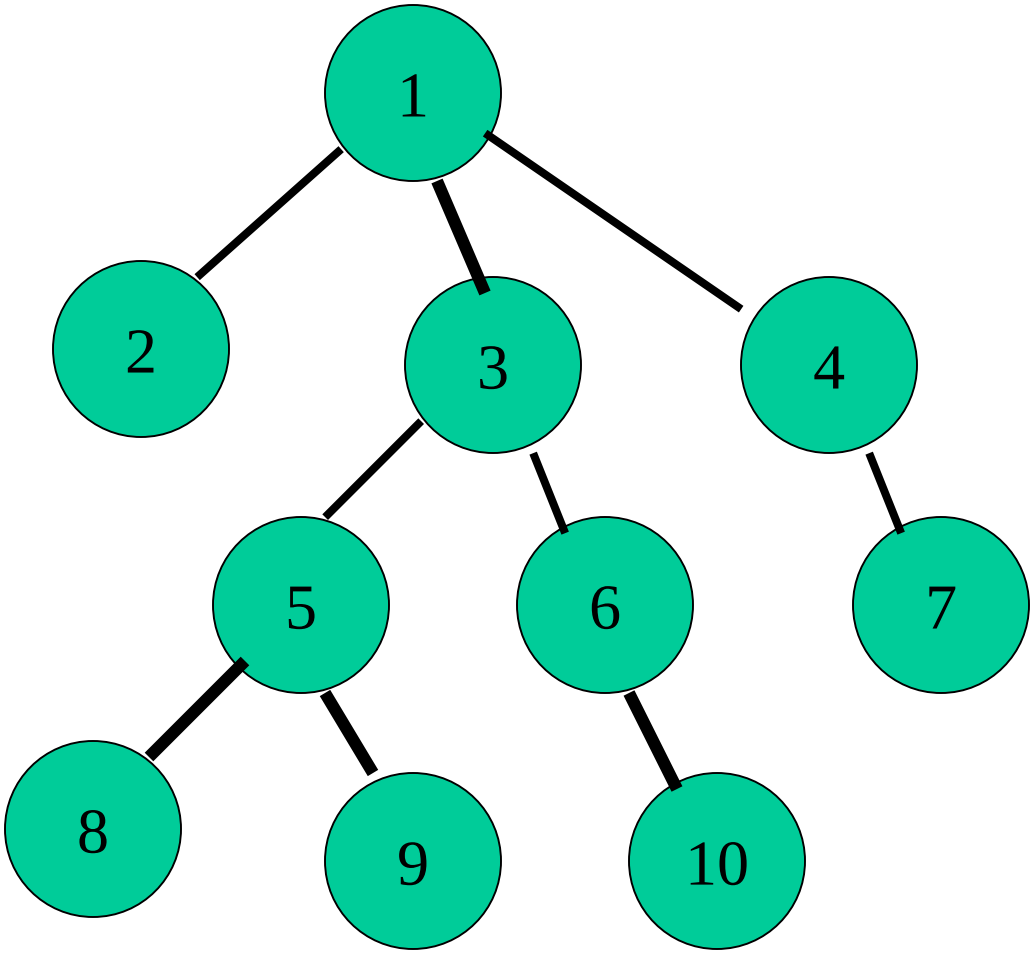


7.4.2 In-Order Traversal

- Using in-order traversal of a binary tree to solve the expression evaluation problem.



In-order traversal



In-order list –2,1,8,5,9,3,10,6,7,4

7.4.3 Post-Order Traversal

- A post-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.

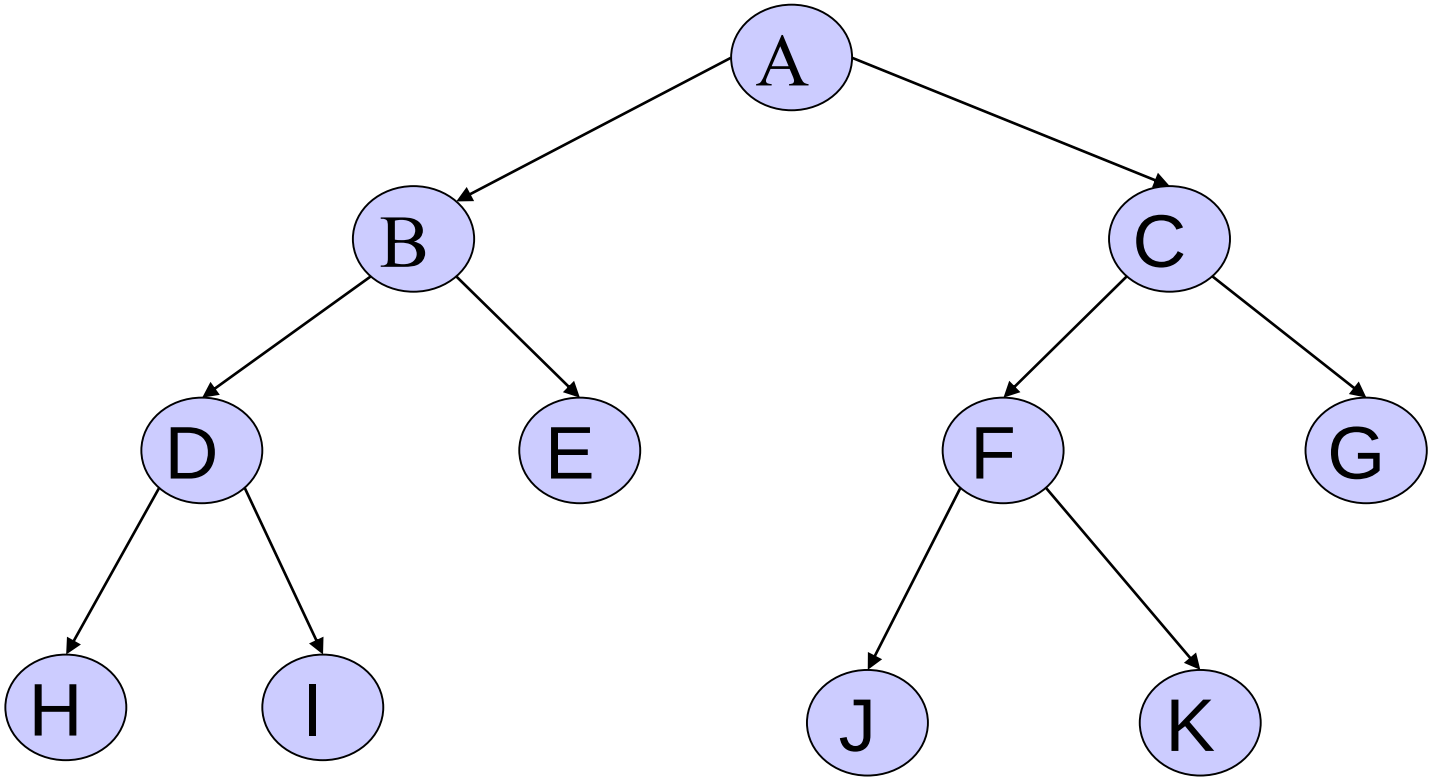
Post-order traversal

The post-order listing of the nodes of T is the nodes of T_1 in post-order, then the nodes of T_2 in post order and so-on up to T_k, all followed by n

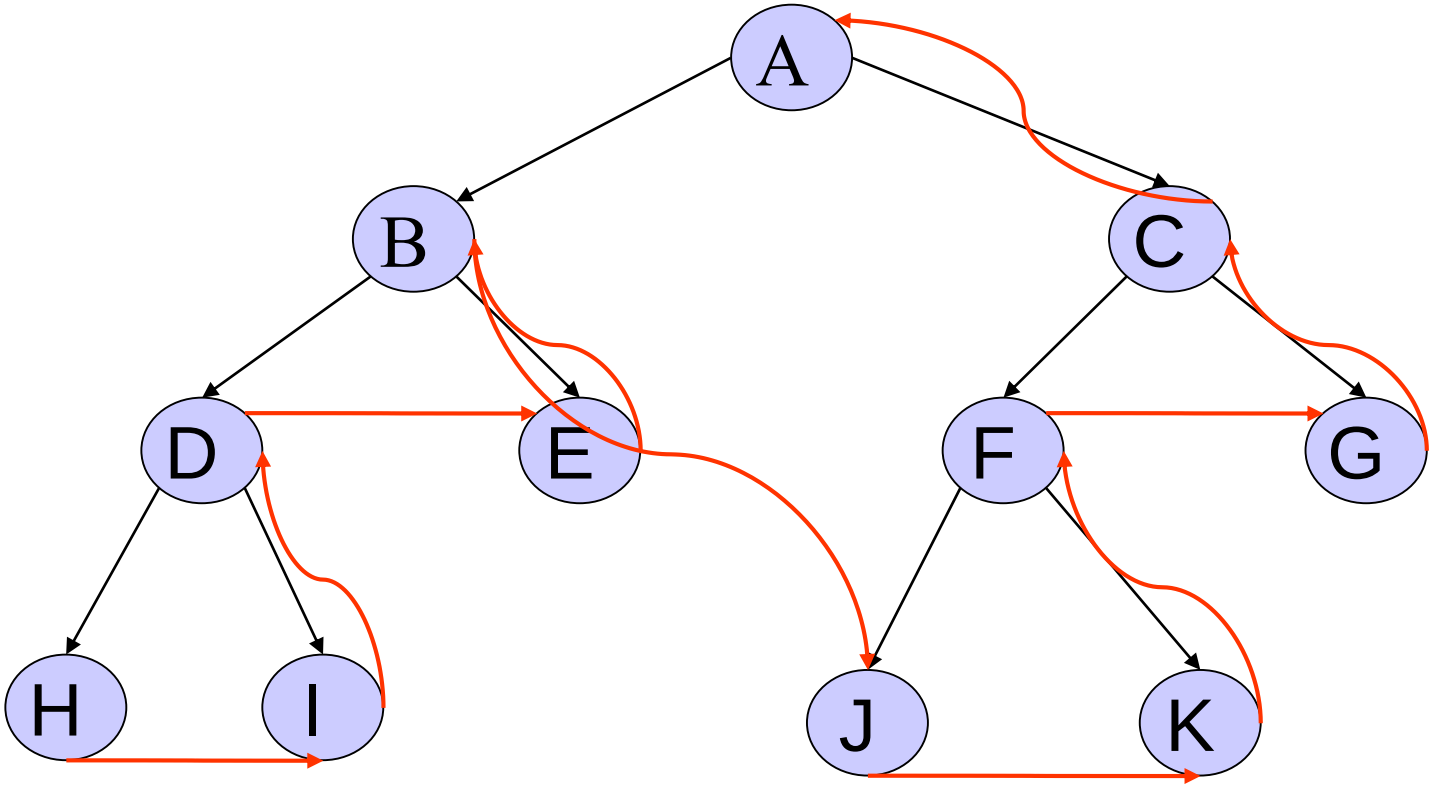
Algorithm

- Travers the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- Visit the root

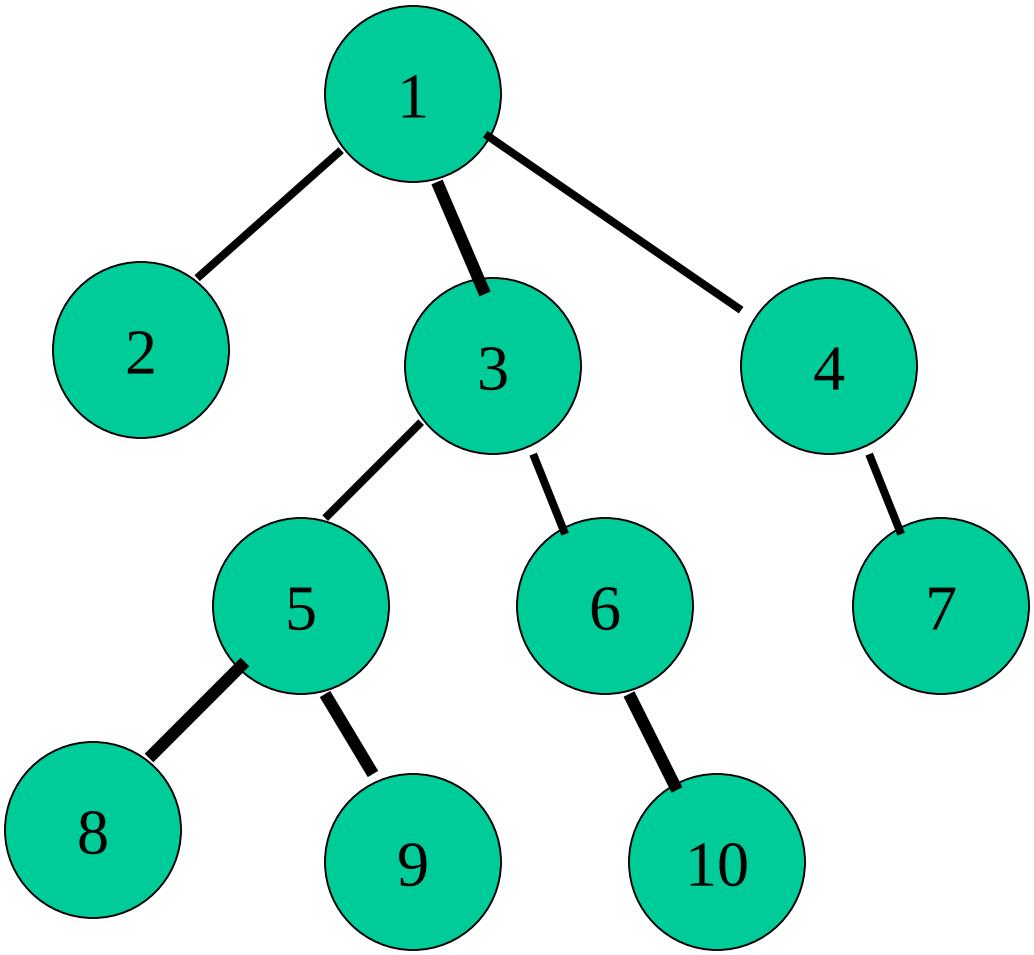
7.4.3 Post-Order Traversal



7.4.3 Post-Order Traversal



post-order traversal



Pre-order list –2,8,9,5,10,6,3,7,4,1

Preorder traversal on a binary tree.

```
1 def preorderTrav( subtree ):
2   if subtree is not None :
3     print( subtree.data )
4     preorderTrav( subtree.left )
5     preorderTrav( subtree.right )
```

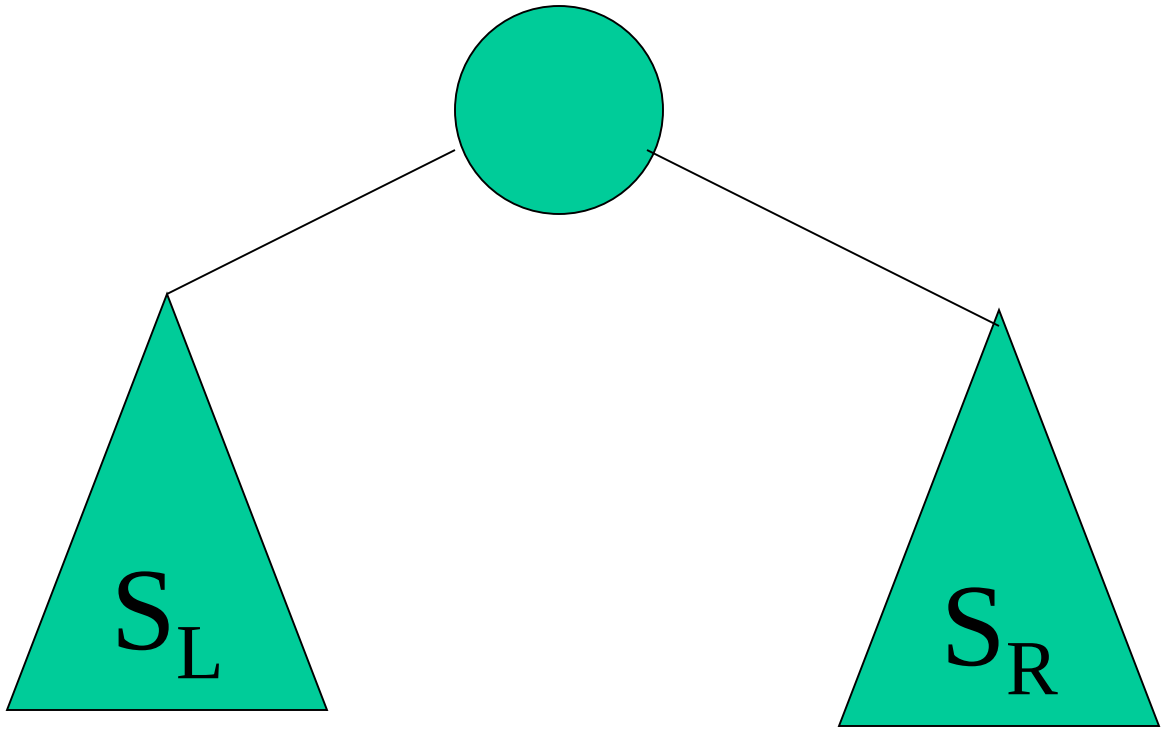
Inorder traversal on a binary tree.

```
1 def inorderTrav( subtree ):
2   if subtree is not None :
3     inorderTrav( subtree.left )
4     print( subtree.data )
5     inorderTrav( subtree.right )
```

Post order Traversal

```
1  def postorderTrav( subtree ):
2    if subtree is not None :
3      postorderTrav( subtree.left )
4      postorderTrav( subtree.right )
5      print( subtree.data )
```

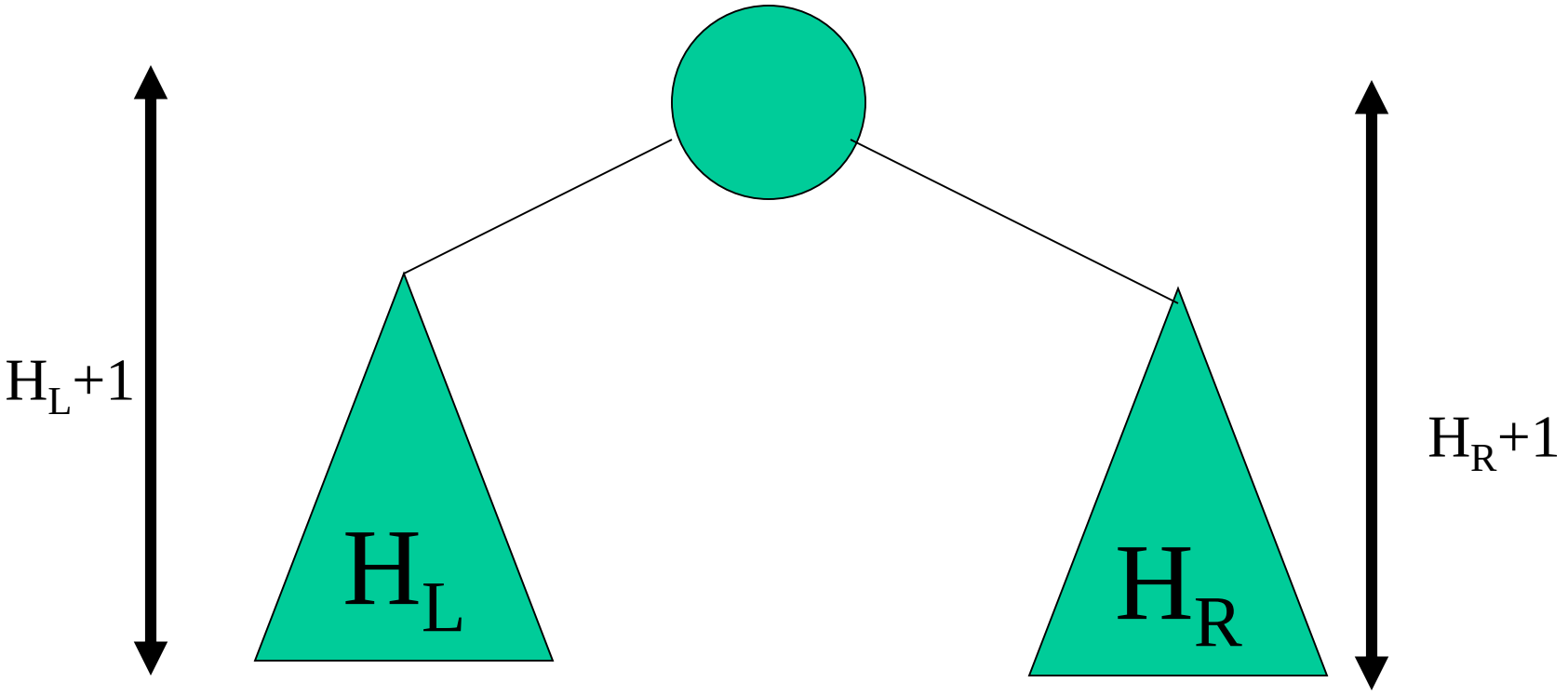
Recursive view used to calculate the size of a tree : $S_T=S_L+S_R+1$



Pseudo Code Algorithm to calculate the size of the binary tree

- //* Return the size of the binary tree rooted at t.
- size (BinaryNode t)
- if (t = = null)
- return 0
- else
- return 1+size(t.left)+ size(t.right)
- endif

Recursive view used to calculate the height of a tree : $H_T=\max(H_L+1, H_R+1)$



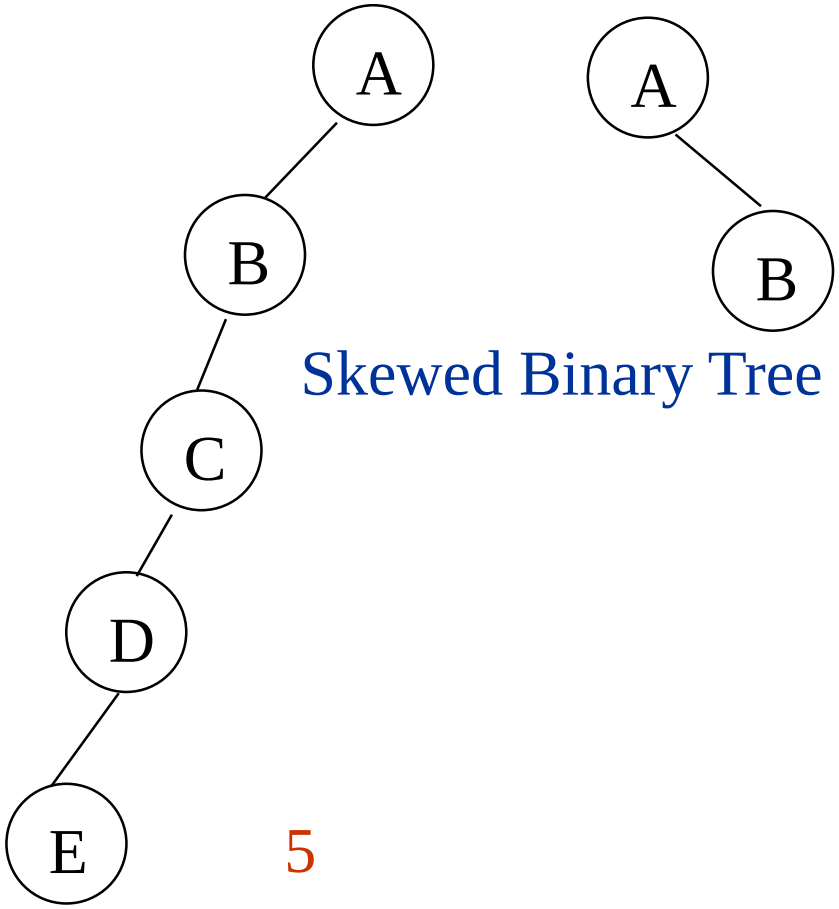
Routine to compute the height of a node (figure 1.2)

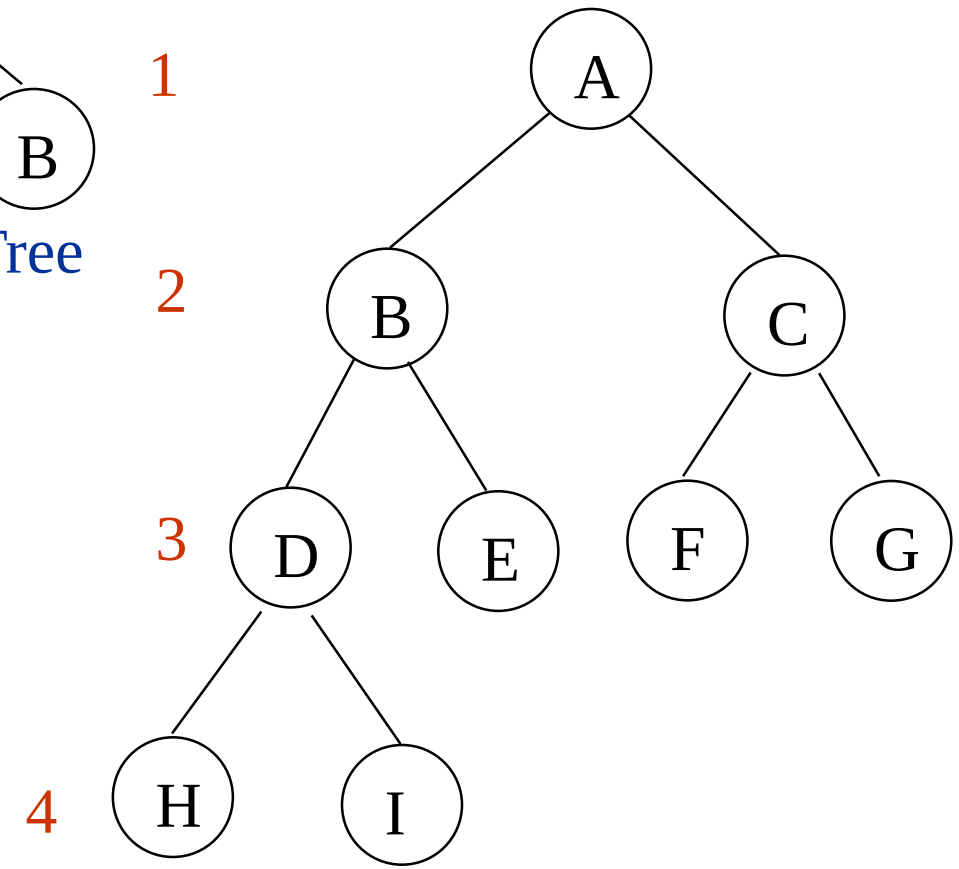
- //* Return the height of the binary tree rooted at t.
- height (BinaryNode t)
- If (t = = null)
- return 0;
- else
- return 1+math.max(height(t.left),height(t.ight);
- }

Tree ADT

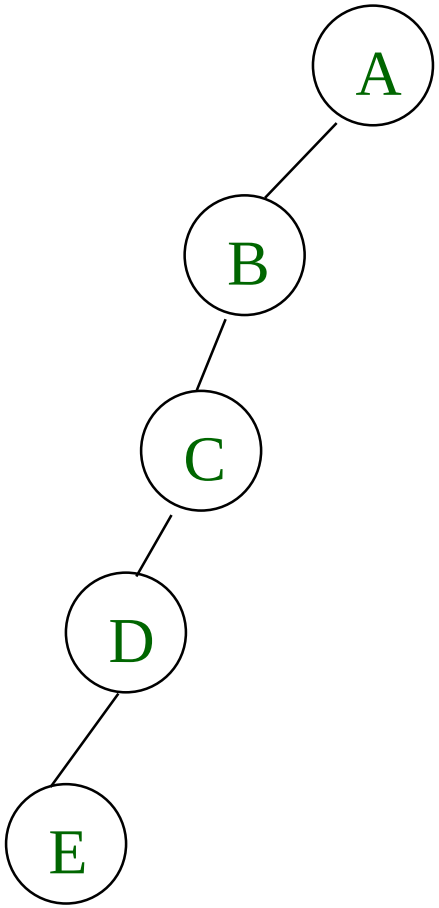
- Objects: any type of objects can be stored in a tree
- Methods:
- accessor methods
  - root() – return the root of the tree
  - parent(p) – return the parent of a node
  - children(p) – returns the children of a node
- query methods
  - size() – returns the number of nodes in the tree
  - isEmpty() - returns true if the tree is empty
  - elements() – returns all elements
  - isRoot(p), isInternal(p), isExternal(p)

Samples of Trees



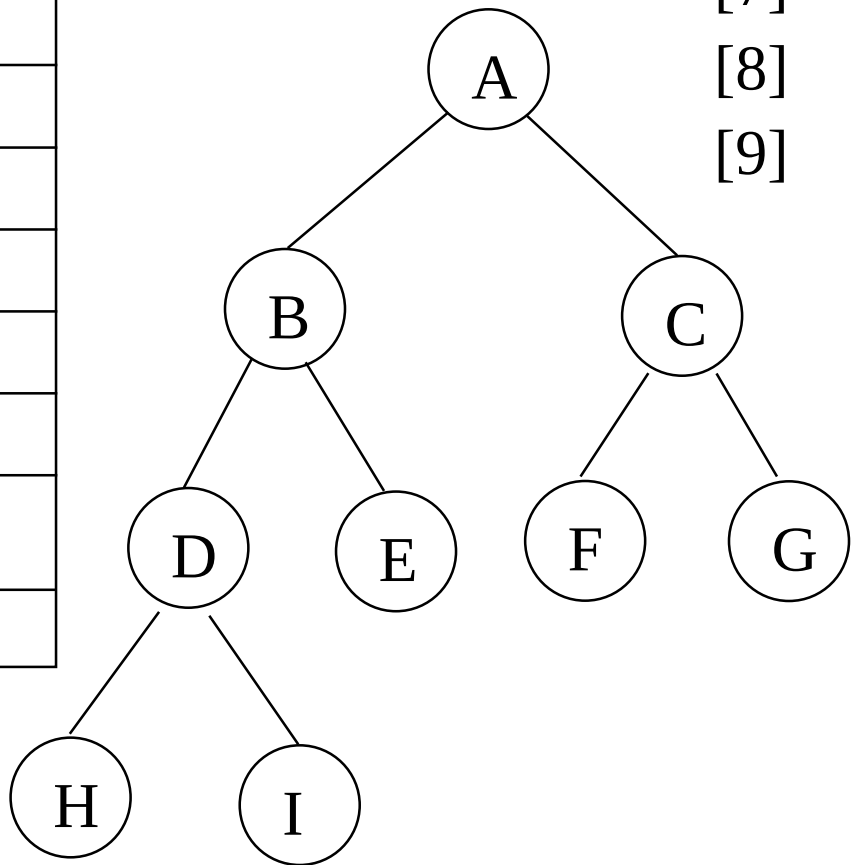


Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space
(2) insertion/deletion problem

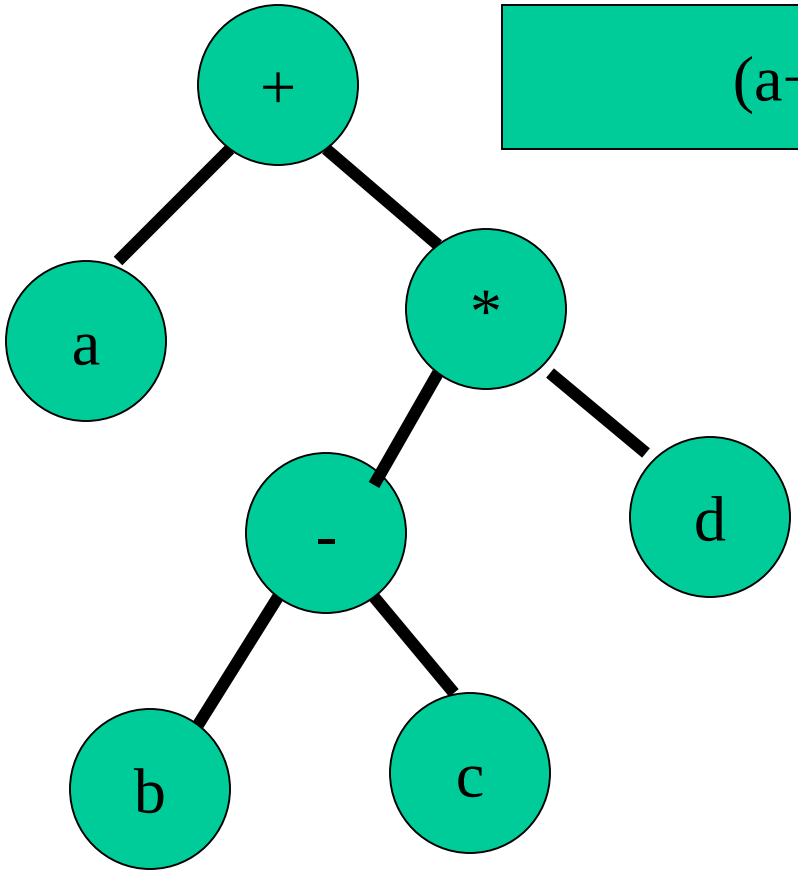


[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Expression trees

- One use of the binary tree is in the expression tree, which is central data structure in compiler design.
- The structure of the expression tree is based on the order in which the operators are evaluated. The operator in each internal node is evaluated after both its left and right subtrees have been evaluated.

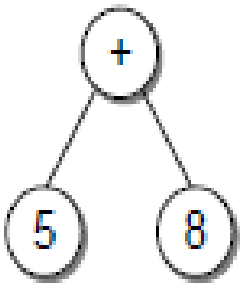
Example :Expression tree



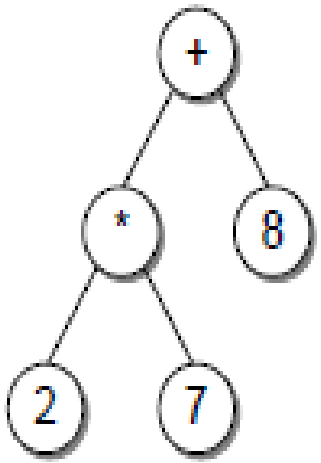
(a+((b-c)*d))

The leaves of an expression tree are operands, such as constant, variable names. The other nodes contain operators.

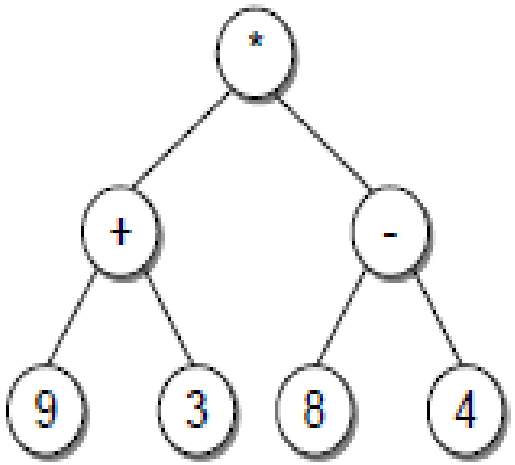
Examples :Expression tree



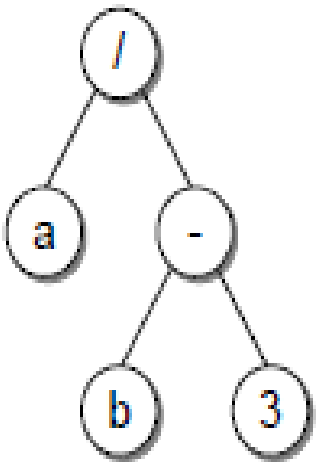
5 + 8



2 * 7 + 8

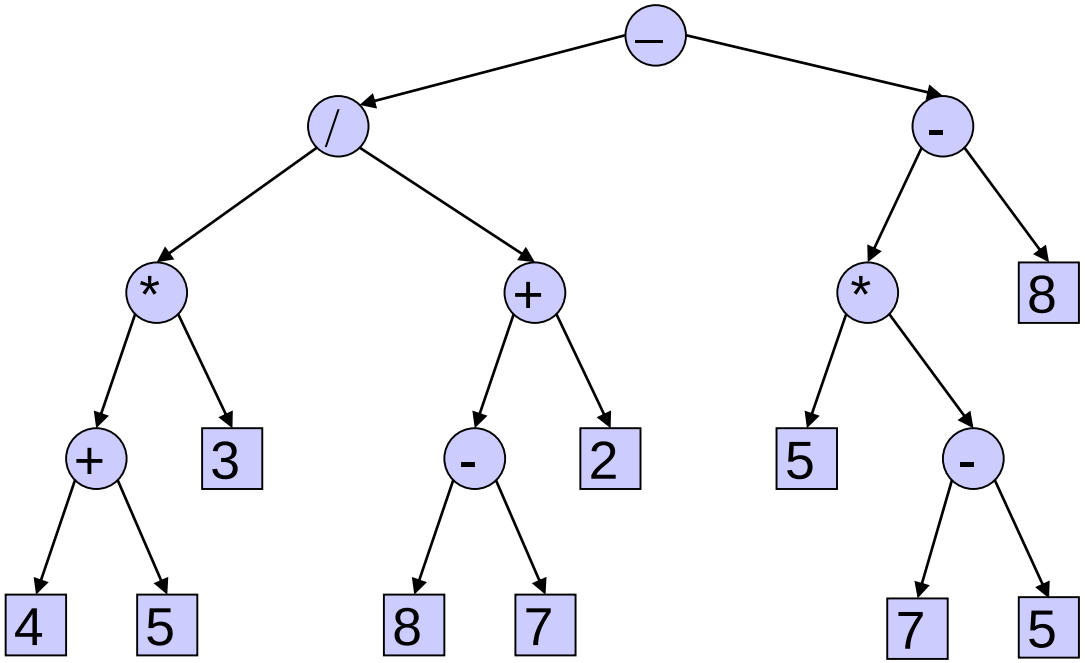


(9 + 3) * (8 - 4)



a / (b - 3)

Given a expression (((4+5)*3)/((8-7)+2))-((5*(7-5))-8), Its expression tree is as follows:



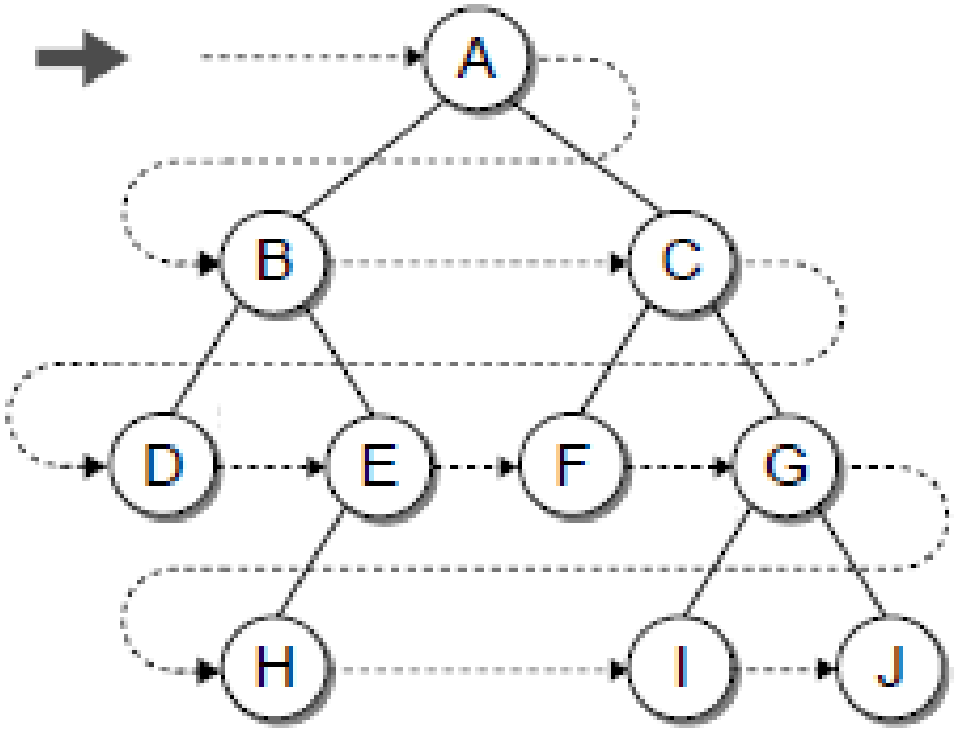
Breadth-First Traversal

- The preorder, inorder, and postorder traversals are all examples of a depth-first traversal.
- That is, the nodes are traversed deeper in the tree before returning to higher-level nodes.

Breadth-First Traversal

- Another type of traversal that can be performed on a binary tree is the breadth-first traversal.
- In a breadth-first traversal, the nodes are visited by level, from left to right.

The logical ordering of the nodes with a breadth-first traversal



Implementation of Breadth-first-traversal

- In the above example tree, we would visit node A followed by nodes B and C, which is the correct ordering.
- The best way to save a node's children for later access is to use a queue.
- We can then use an iterative loop to move across the tree in the correct node order to produce a breadth-first traversal.

Implementation of Breadth-First-Traversal

- BFT uses a queue to implement the breadth-first traversal.
- The process starts by saving the root node and in turn readying the iterative loop. During each iteration, we remove a node from the queue, visit it, and then add its children to the queue.
- The loop terminates after all nodes have been visited.

Breadth-first traversal on a binary tree.

```
1  def breadthFirstTrav( bintree ):
2     # Create a queue and add the root node to it.
3    Queue q
4    q.enqueue( bintree )
5
6     # Visit each node in the tree.
7    while not q.isEmpty() :
8       # Remove the next node from the queue and visit it.
9      node = q.dequeue()
10     print( node.data )
11
12      # Add the two children to the queue.
13     if node.left is not None :
14       q.enqueue( node.left )
15     if node.right is not None :
16       q.enqueue( node.right )
```

Expression Tree Construction

Now let's look at how to construct the tree given an infix expression. For simplicity, we assume the following:

(1) the expression is stored in string with no white space;

(2) the supplied expression is valid and fully parenthesized;

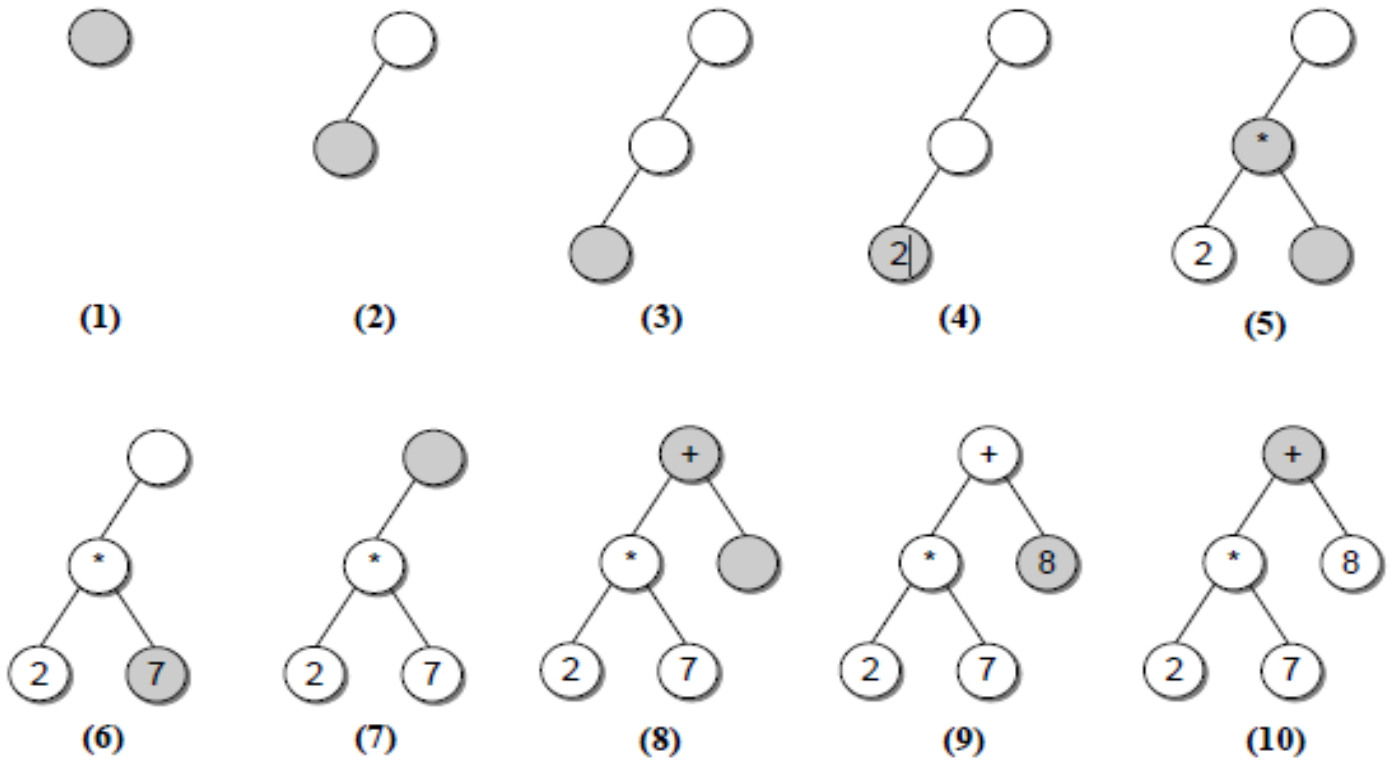
(3) each operand will be a single-digit or single-letter variable; and

(4) the operators will consist of +, -, *, /, and %.

Expression Tree Construction

- Consider the following Figure, which illustrates the steps required to build the tree for the expression ((2*7)+8).
- The steps illustrated in the Figure are described below:

Steps for building an expression tree for ((2 7) + 8).



Steps for building an expression tree for ((2 7) + 8).

1. Create an empty root node and mark it as the current node.
2. Read the left parenthesis: add a new node as the left child and descend down to the new node.
3. Read the next left parenthesis: add a new node as the left child and descend down to the new node.
4. Read the operand 2: set the value of the current node to the operand and move up to the parent of the current node.
5. Read the operator *: set the value of the current node to the operator and create a new node linked as the right child. Then descend down to the new node.
6. Read the operand 7: set the value of the current node to the operand and move up to the parent of the current node.
7. Read the right parenthesis: move up to the parent of the current node.
8. Read the operator +: set the value of the current node to the operator and create a new node linked as the right child; descend down to the new node.

Constructing an expression tree

```
1  class ExpressionTree :
2  # ...
3    def _buildTree( self, expStr ):
4       # Build a queue containing the tokens in the expression string.
5      expQ = Queue()
6      for token in expStr :
7        expQ.enqueue( token )
8
9       # Create an empty root node.
10     self._expTree = _ExpTreeNode( None )
11      # Call the recursive function to build the expression tree.
12     self._recBuildTree( self._expTree, expQ )
13
14   # Recursively builds the tree given an initial root node.
15   def _recBuildTree( self, curNode, expQ ):
16      # Extract the next token from the queue.
17     token = expQ.dequeue()
18
19      # See if the token is a left paren: '('
20     if token == '(' :
21       curNode.left = _ExpTreeNode( None )
22       buildTreeRec( curNode.left, expQ )
23
24        # The next token will be an operator: + - / * %
25       curNode.data = expQ.dequeue()
26       curNode.right = _ExpTreeNode( None )
27       self._buildTreeRec( curNode.right, expQ )
28
29        # The next token will be a ), remove it.
30       expQ.dequeue()
31
32      # Otherwise, the token is a digit that has to be converted to an int.
33     else :
34       curNode.element = token
```