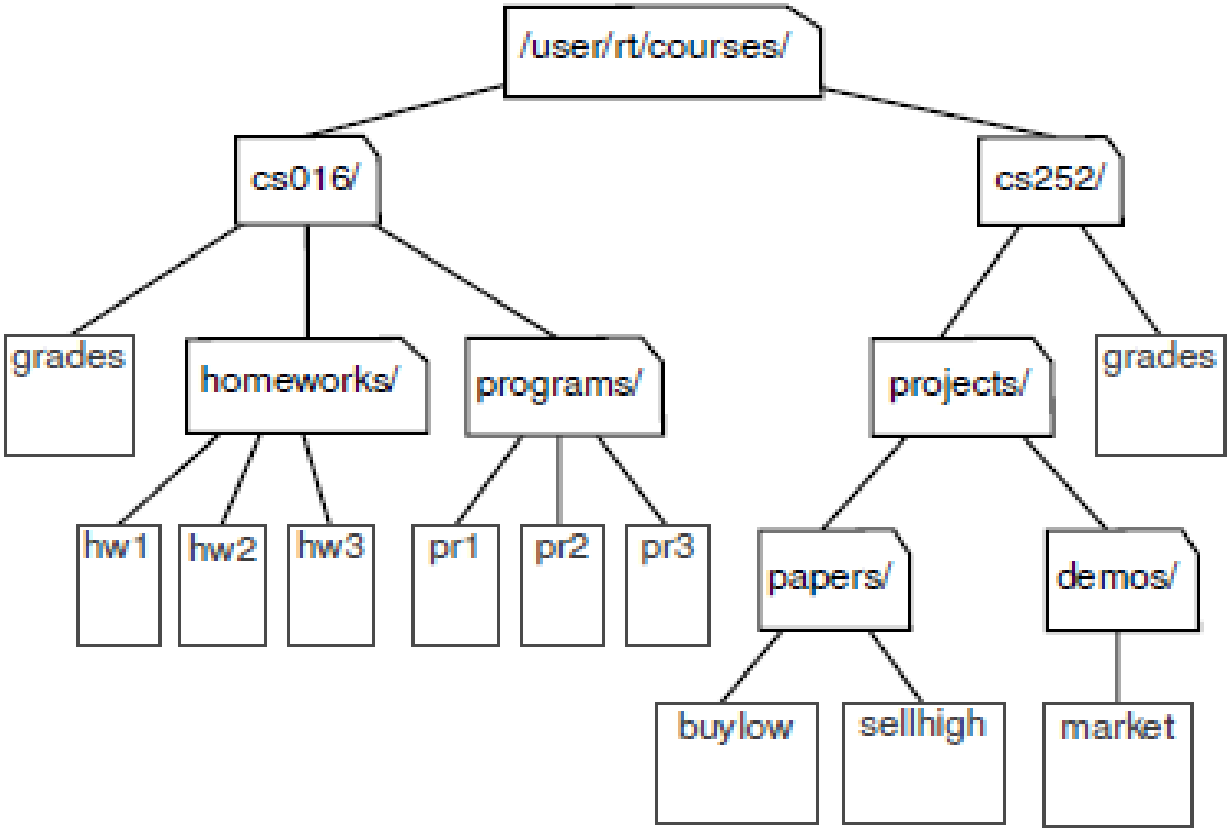
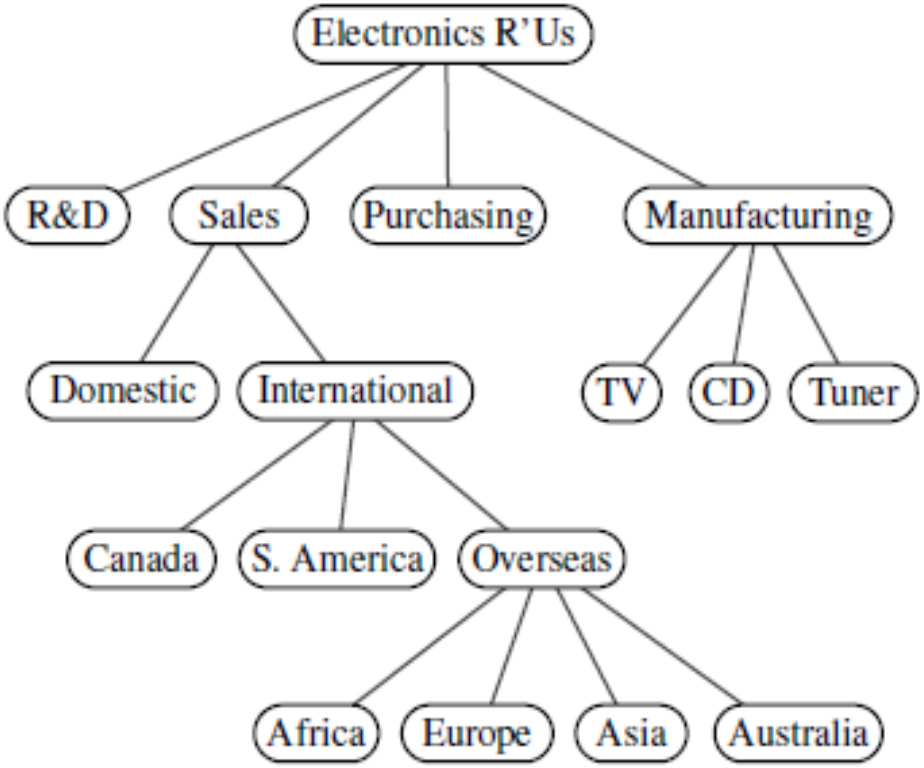
# Trees


/user/rt/courses/
cs016/
cs252/
grades
homeworks/
programs/
projects/
grades
hw1
hw2
hw3
pr1
pr2
pr3
papers/
demos/
buylow
sellhigh
market

# What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments

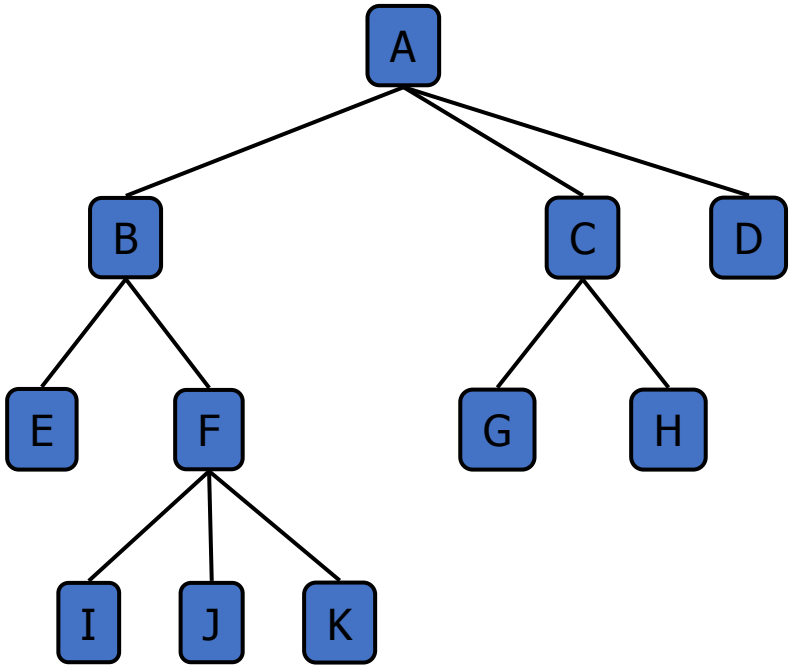


# Formal Definition: Tree

- A ***tree*** *T i*s 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*.

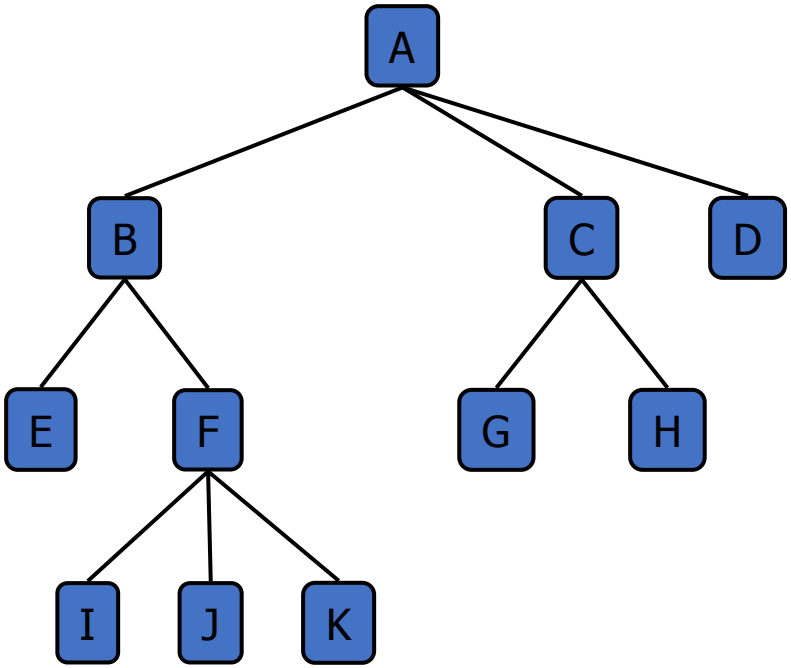
# Tree Terminology

- Root: node without parent (A)



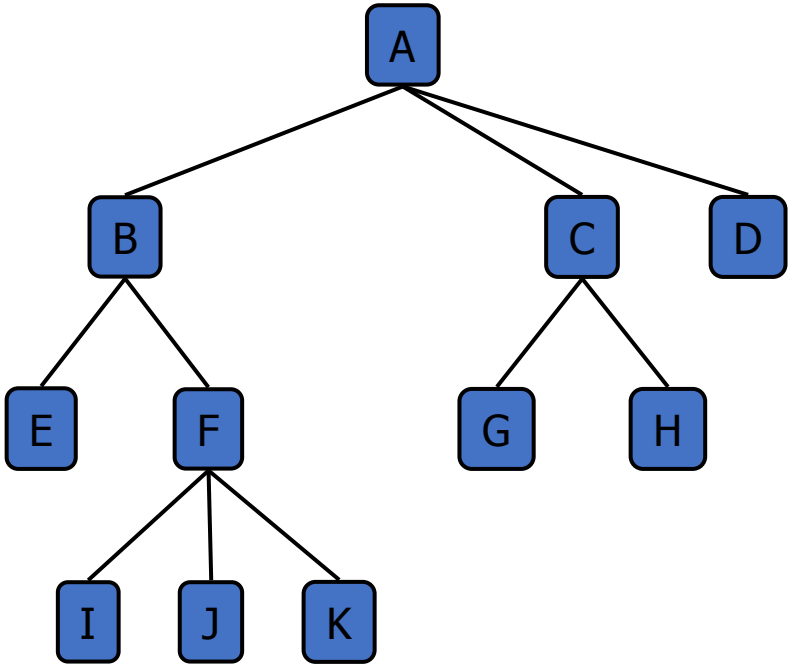
# Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)



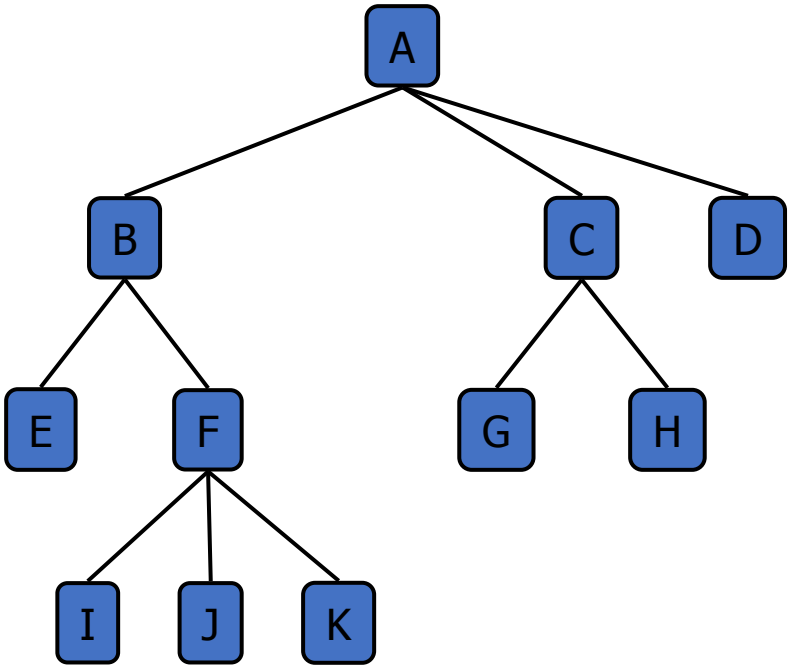
# Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf ): node without children (E, I, J, K, G, H, D)



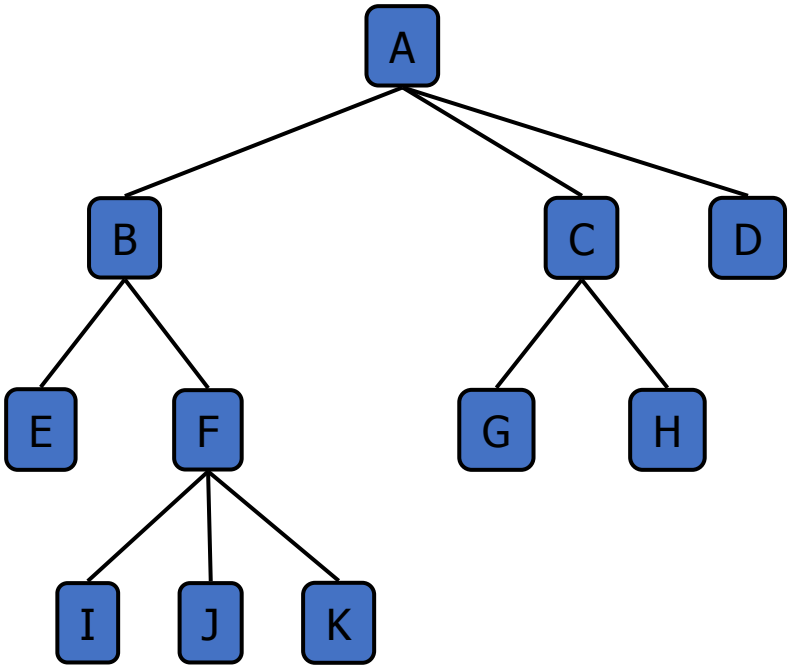
# Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf ): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.



# Tree Terminology

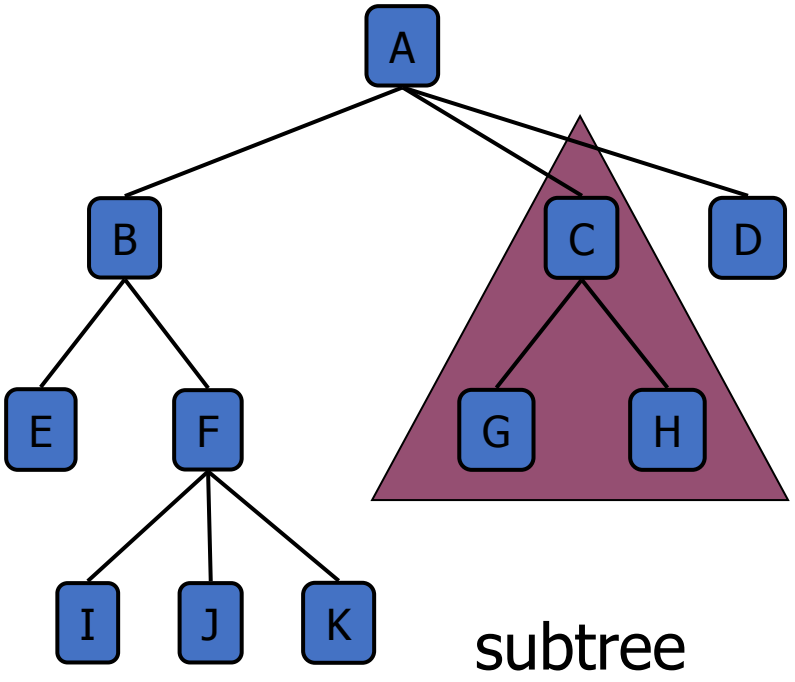
- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf ): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.



# Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf ): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendant of a node: child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



# Tree Terminology

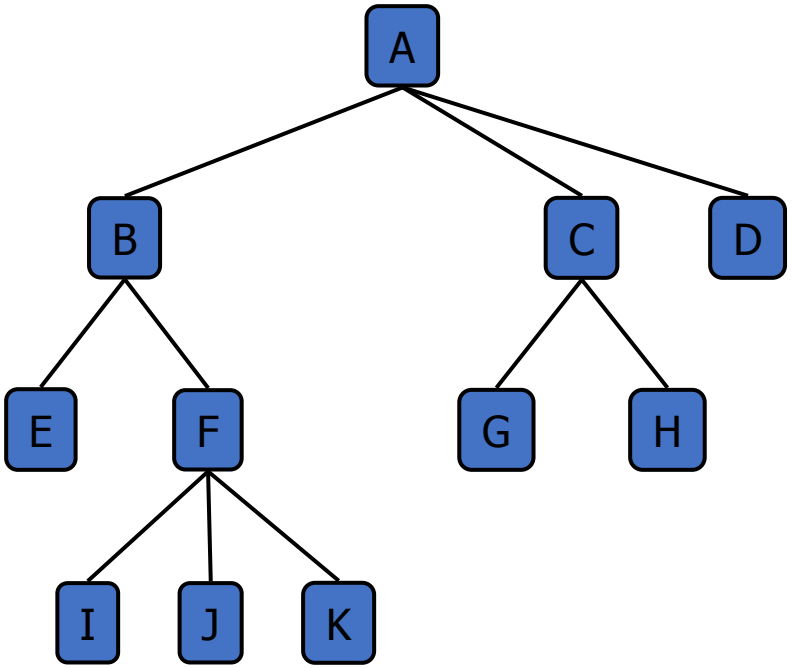
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)

- Edge : (u,v) ;

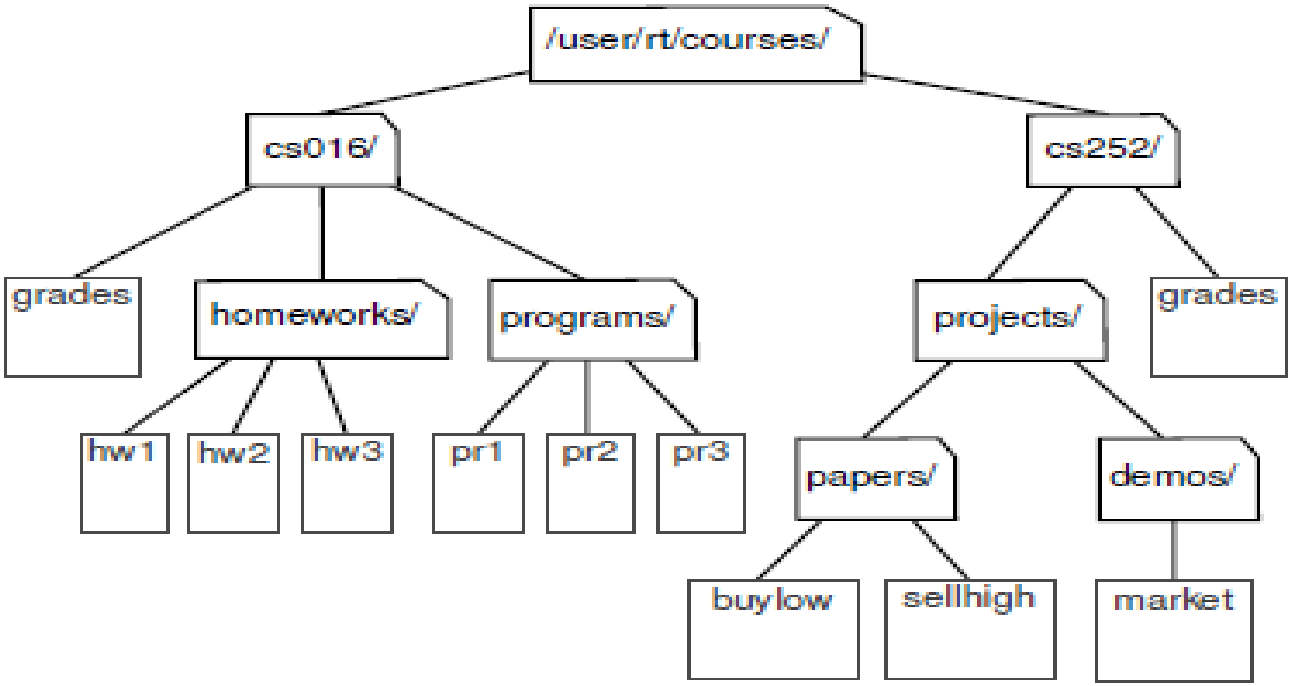
  u=parent(v) or v=parent(u)

- Path : A sequence of nodes such that any two consecutive nodes in the sequence form an edge.

  (B,F,J)

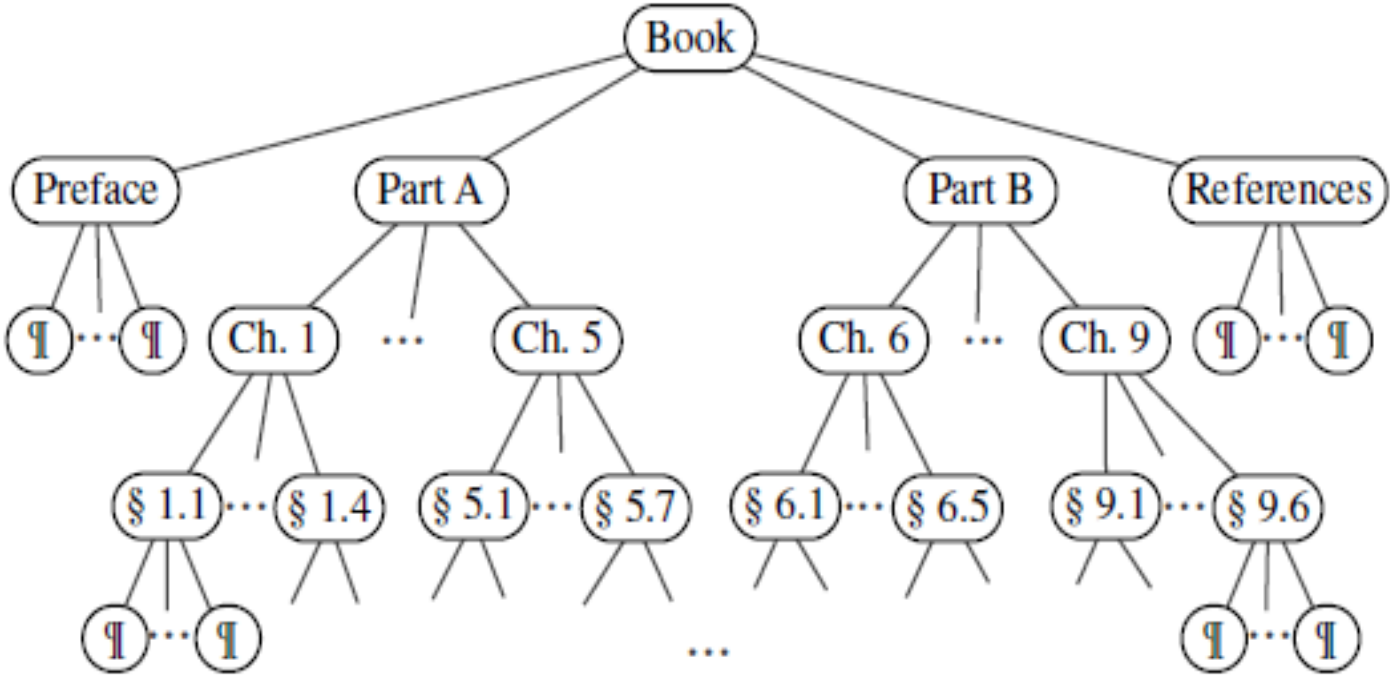


# Tree Terminology




- Depth of a node:  Height of a tree:
- Edge
- Path

# Ordered Trees




- Meaningful linear order among the children of each node

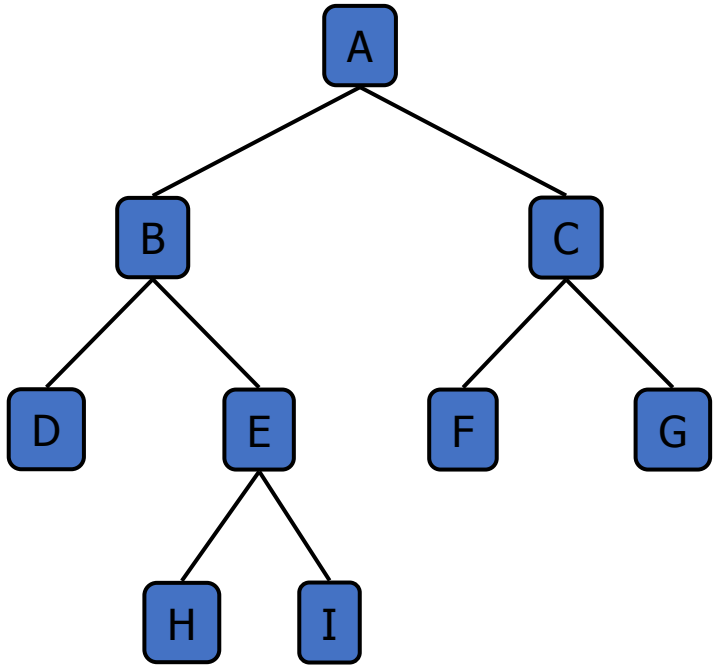
# Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - T.positions()
- Accessor methods:
  - T.root()
  - T.isroot(p)
  - Iter(T)
  - len(T)
  - T.children(p)
  - …..

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

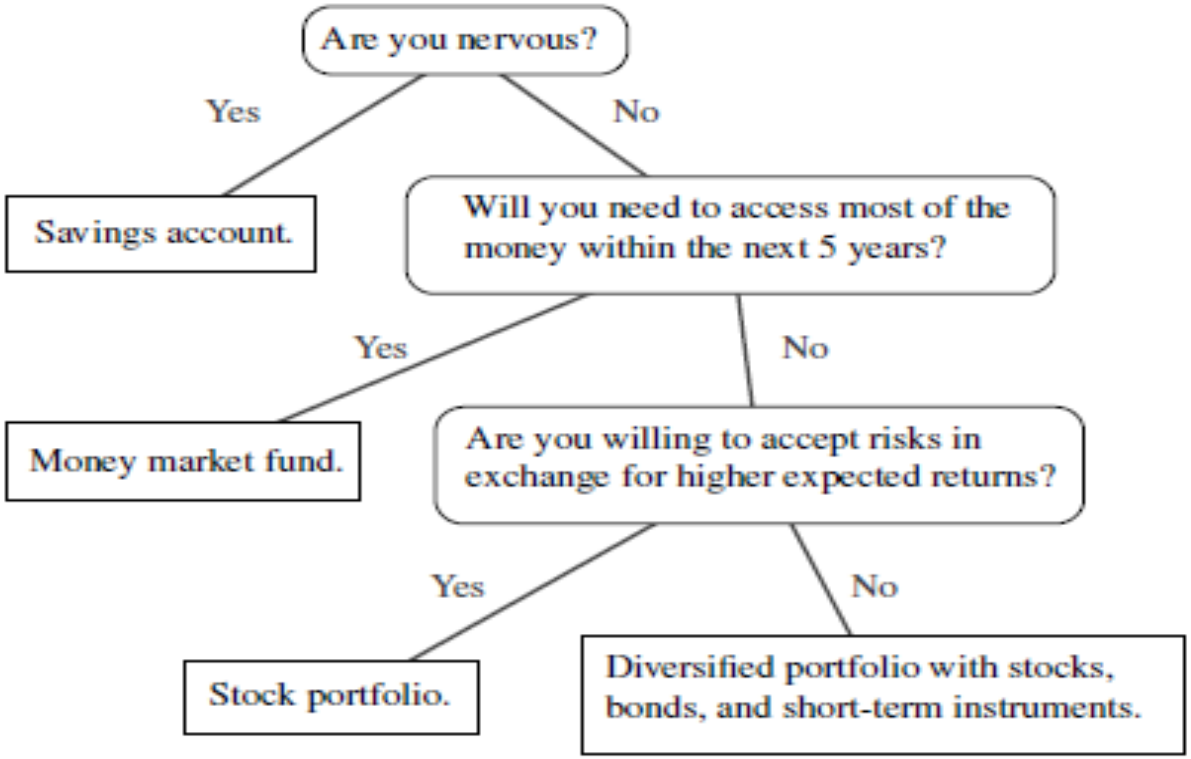
# Binary Trees

- Ordered tree with following properties
  - Every node has at most two children.
  - Each child node is labeled as being either a ***left child*** or a ***right child***.
  - A left child precedes a right child in the order of children of a node.



# Applications: Binary Trees

- arithmetic expressions
- decision processes
- searching



# Binary Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - T.positions()
- Accessor methods:
  - T.root()
  - T.isroot(p)
  - Iter(T)
  - len(T)
  - T.children(p)
  - …..
  - T.left(p)
  - T.right(p)
  - T.sibling(p)

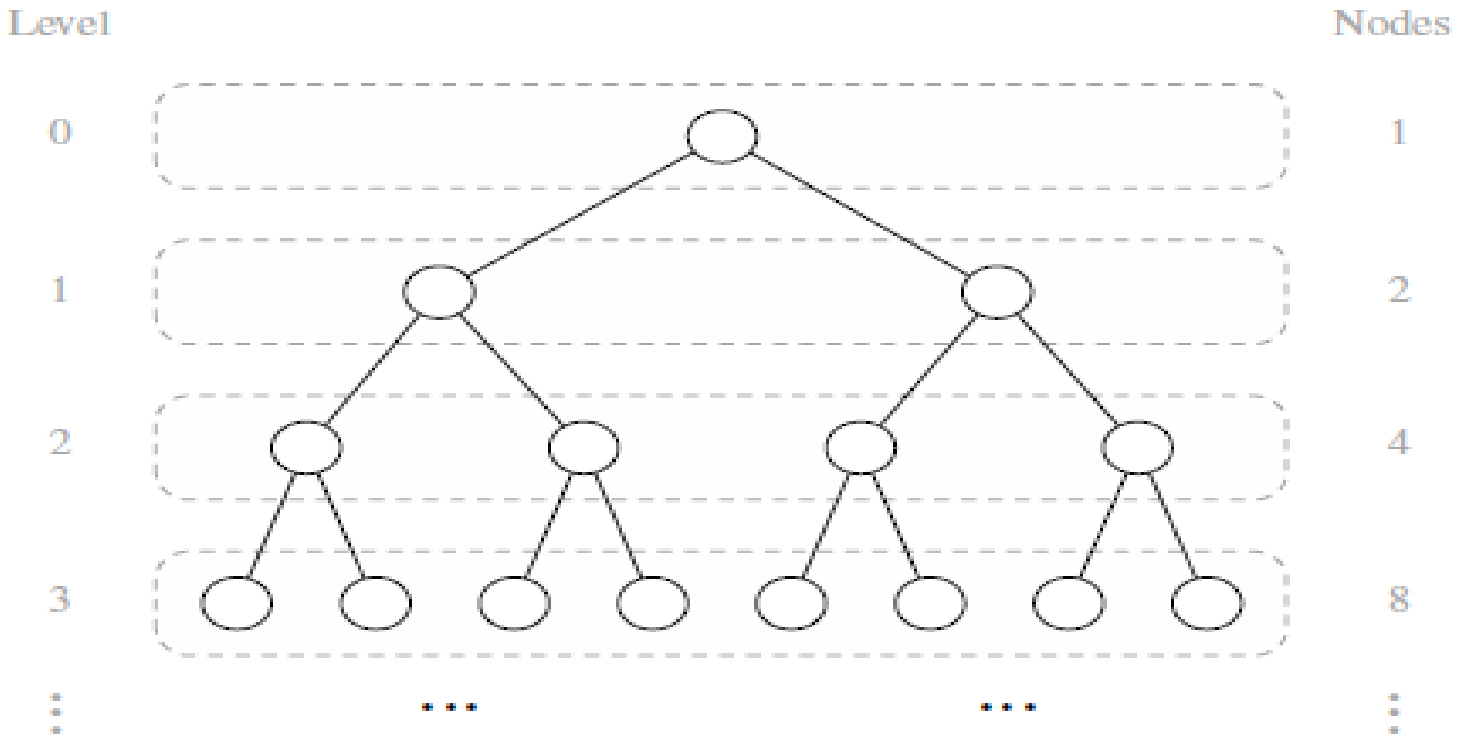
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

# BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - T.left(p)
  - T.right(p)
  - T.sibling(p)
  - boolean hasLeft(p)
  - boolean hasRight(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

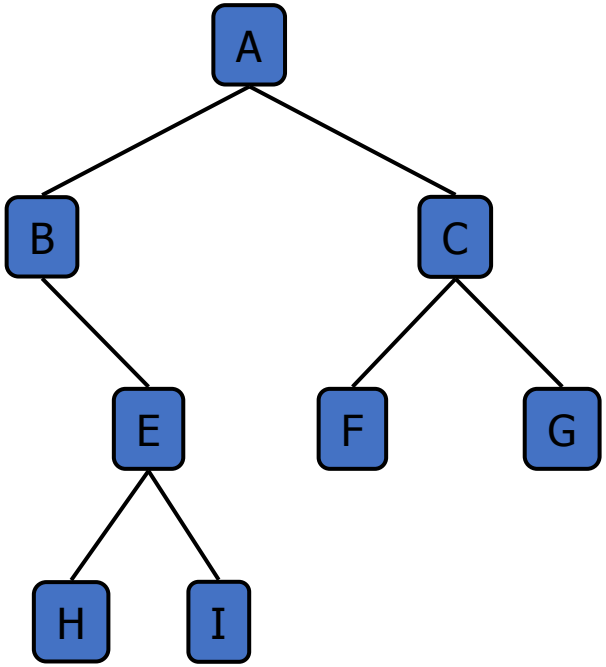
# Level of Binary Trees

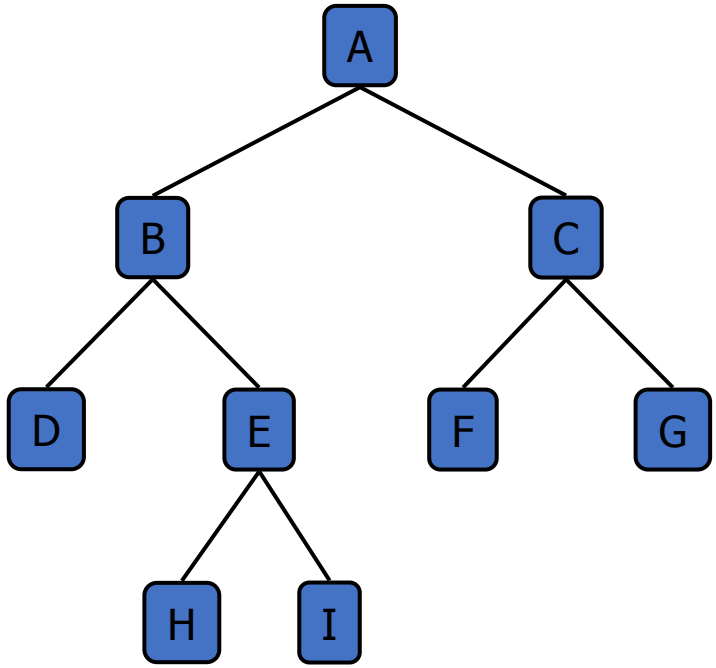
The set of all nodes of a tree $T$ at the same depth $d$ as ***level*** $d$ of $T$



# Proper Binary Trees

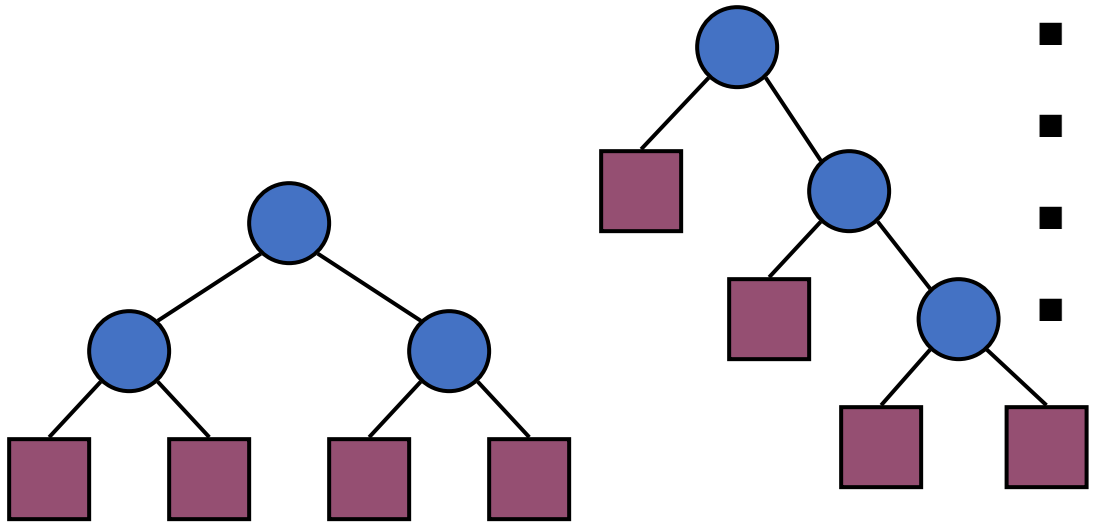
- All internal nodes have exactly 2 children



# Properties of Proper Binary Trees

- Notation
  - $n$ number of nodes
  - $e$ number of external nodes
  - $i$ number of internal nodes
  - $h$ height

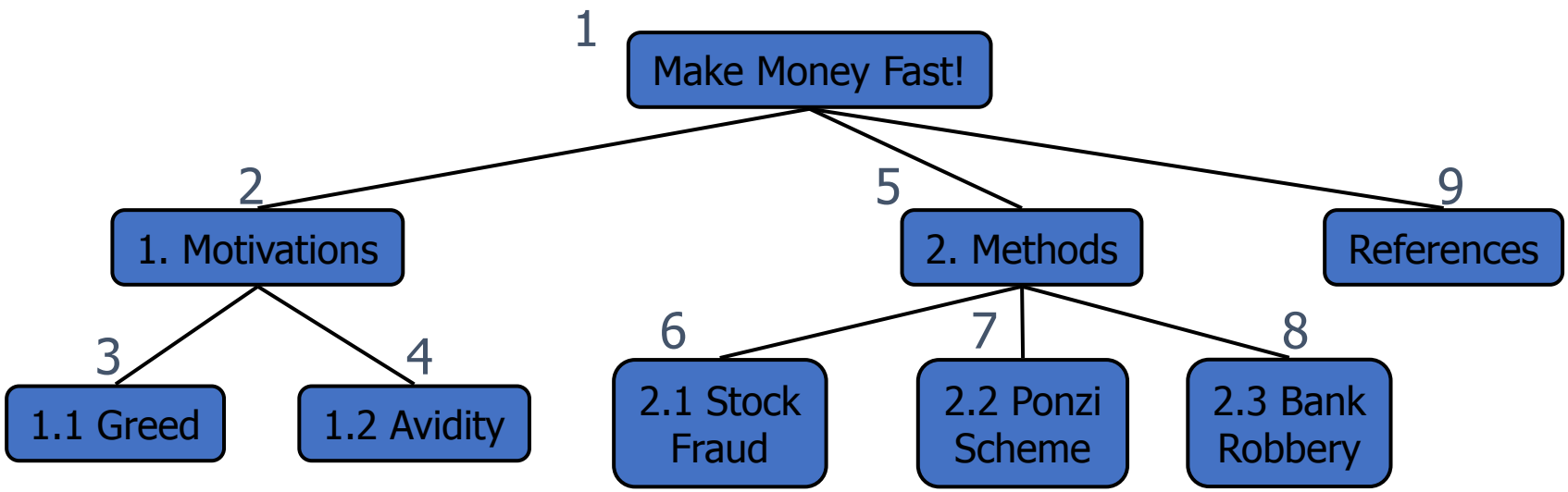


- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

# Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

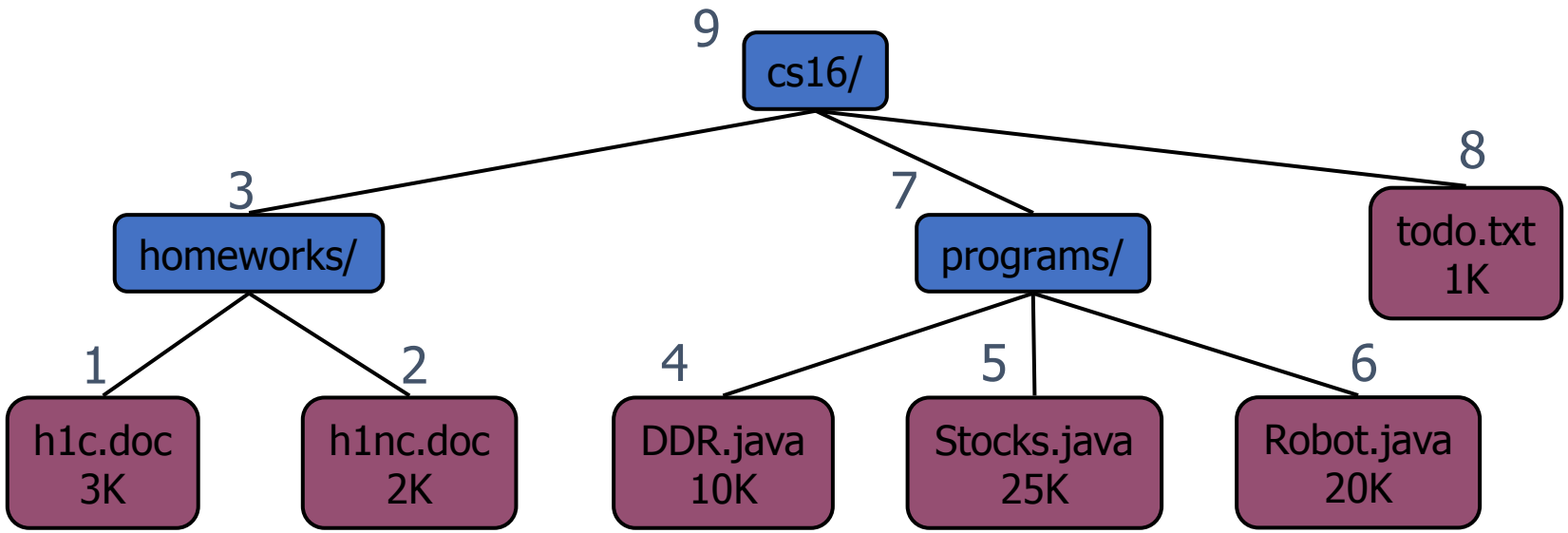
**Algorithm** *preOrder*(*v*)
    *visit*(*v*)
    **for each** child *w* of *v*
        *preorder* (*w*)



# Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

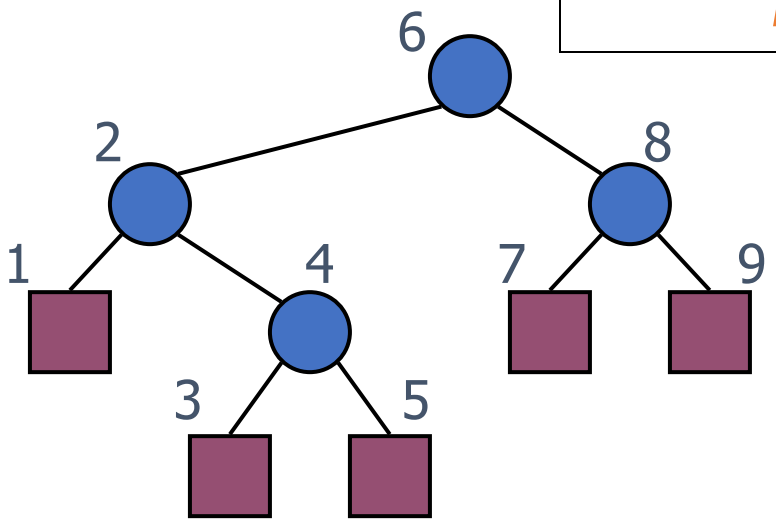
**Algorithm** *postOrder*(*v*)
    **for each** child *w* of *v*
        *postOrder* (*w*)
    *visit*(*v*)



# Inorder Traversal

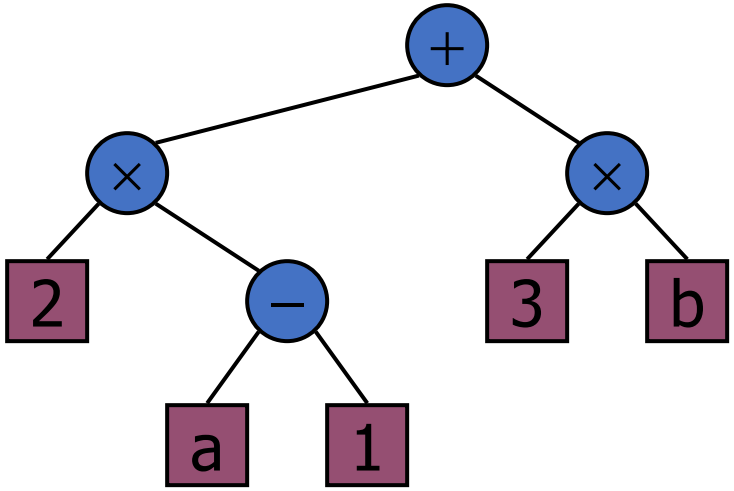
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

```
Algorithm inOrder(v)
    if hasLeft (v)
        inOrder (left (v))
    visit(v)
    if hasRight (v)
        inOrder (right (v))
```



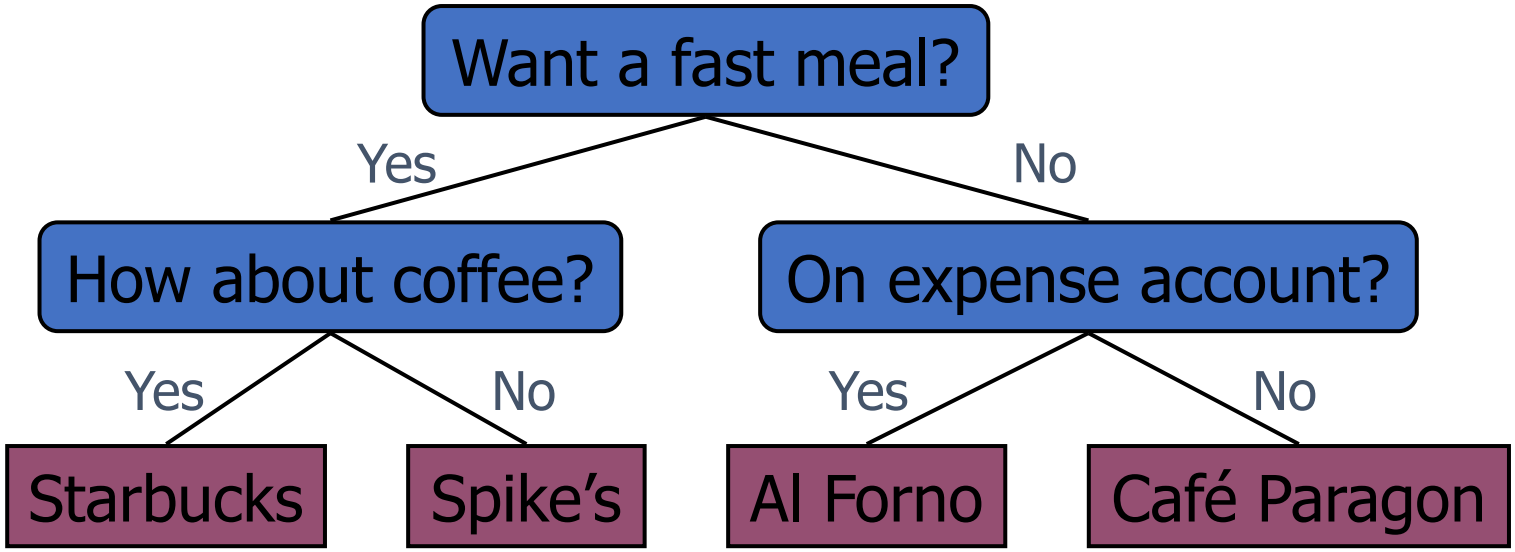
# Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



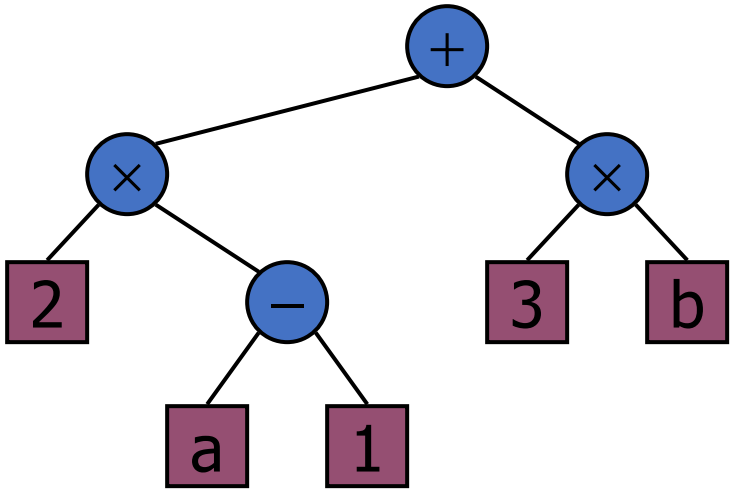
# Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



# Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

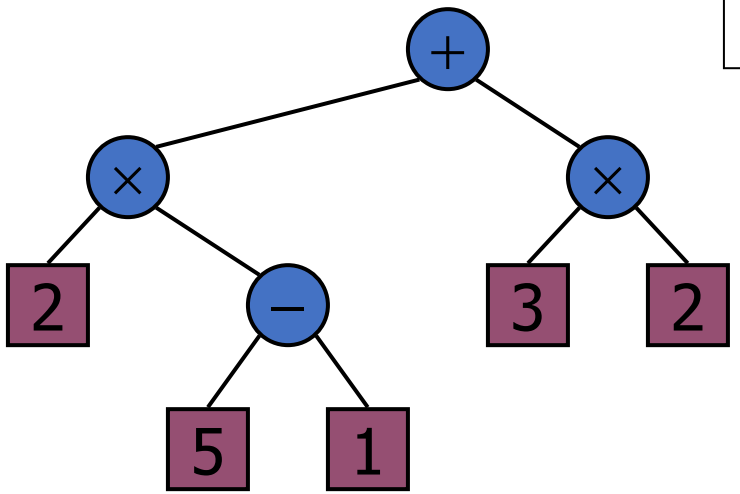



**Algorithm** *printExpression*(*v*)
  **if** *hasLeft* (*v*)
    *print*("(")
    *inOrder* (*left*(*v*))
  *print*(*v.element* ())
  **if** *hasRight* (*v*)
    *inOrder* (*right*(*v*))
    *print* (")")

$$((2 \times (a - 1)) + (3 \times b))$$

# Evaluate Arithmetic Expressions

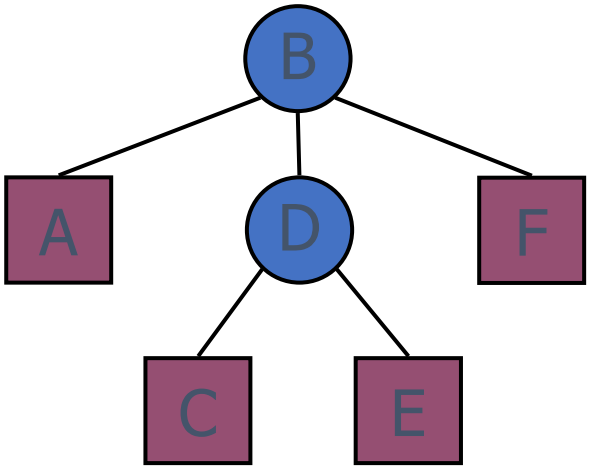
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

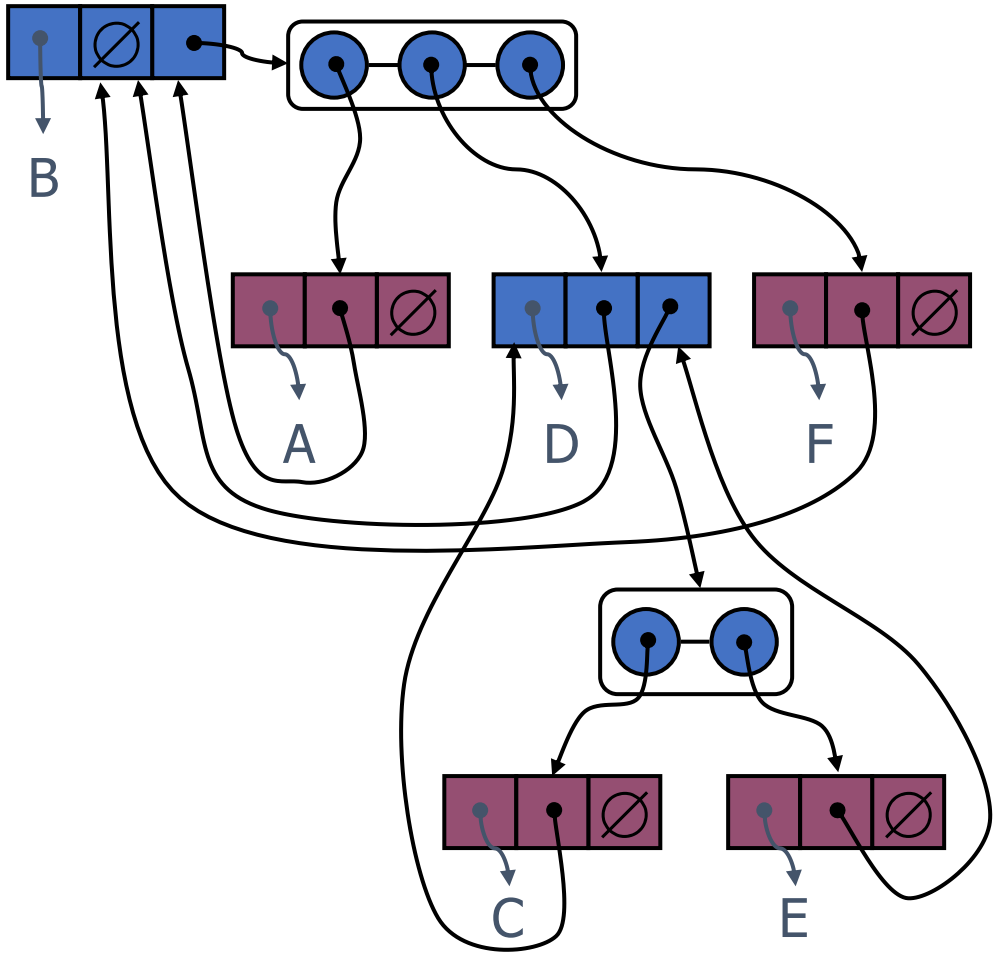



**Algorithm** *evalExpr*(*v*)
    **if** *isExternal* (*v*)
        **return** *v.element* ()
    **else**
        *x* ← *evalExpr*(*leftChild* (*v*))
        *y* ← *evalExpr*(*rightChild* (*v*))
        ◊ ← operator stored at *v*
        **return** *x* ◊ *y*

# Linked Structure for Trees

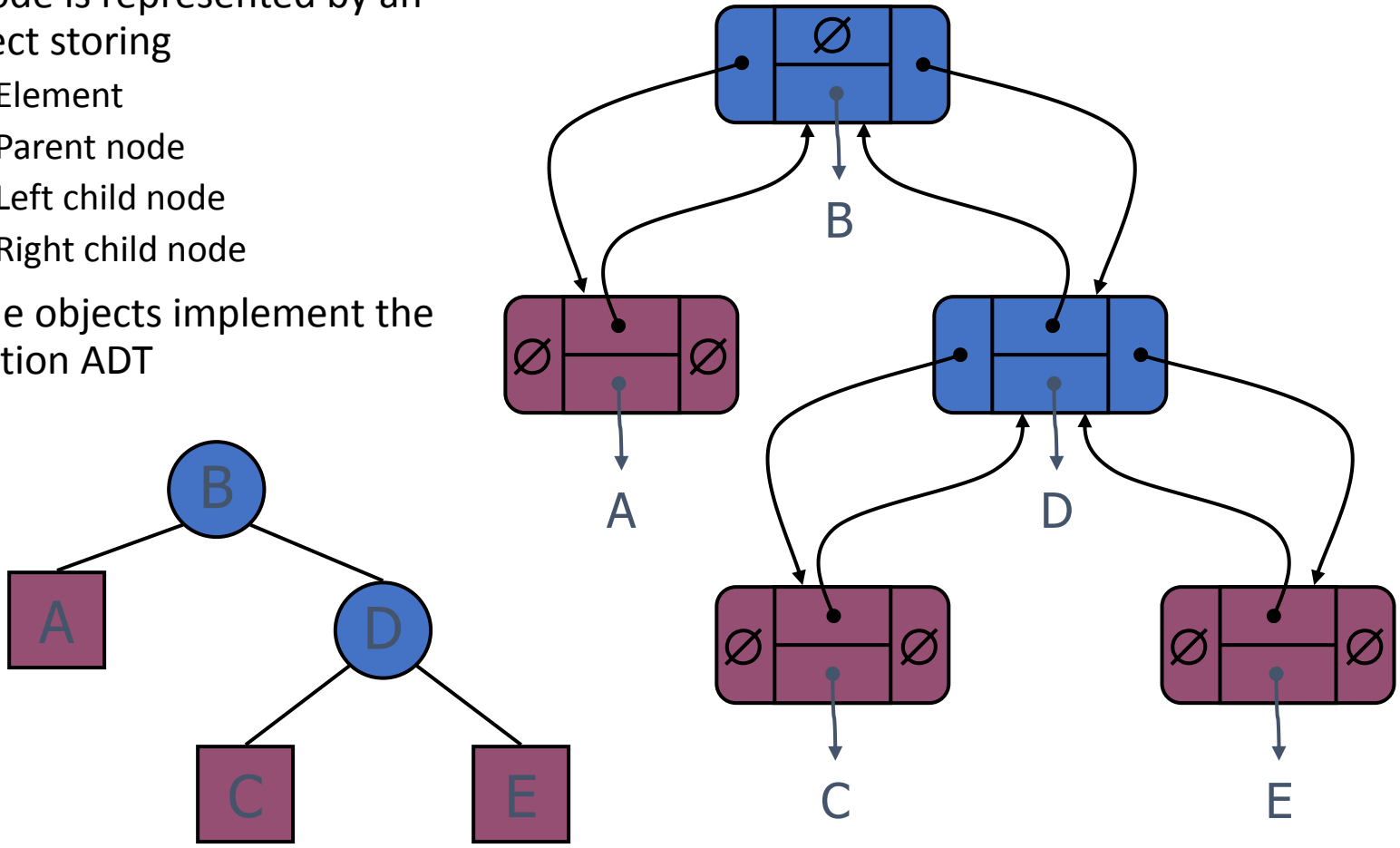
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



# Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



# Operations : Linked Binary Trees

T.add root(e): Create a root for an empty tree, storing e as the element, and return the position of that root; an error occurs if the tree is not empty.

T.add left(p, e): Create a new node storing element e, link the node as the left child of position p, and return the resulting position; an error occurs if p already has a left child.

T.add right(p, e): Create a new node storing element e, link the node as the right child of position p, and return the resulting position; an error occurs if p already has a right child.

**Time Complexity : O(1)**

# Operations : Linked Binary Trees

T.replace(p, e)**:** Replace the element stored at position p with element e, and return the previously stored element.

T.delete(p)**:** Remove the node at position p, replacing it with its child, if any, and return the element that had been stored at p; an error occurs if p has two children.

T.attach(p, T1, T2)**:** Attach the internal structure of trees T1 and T2, respectively, as the left and right subtrees of leaf position p of T, and reset T1 and T2 to empty trees; an error condition occurs if p is not a leaf.

**Time complexity : O(1)**

# Operations and Time complexity

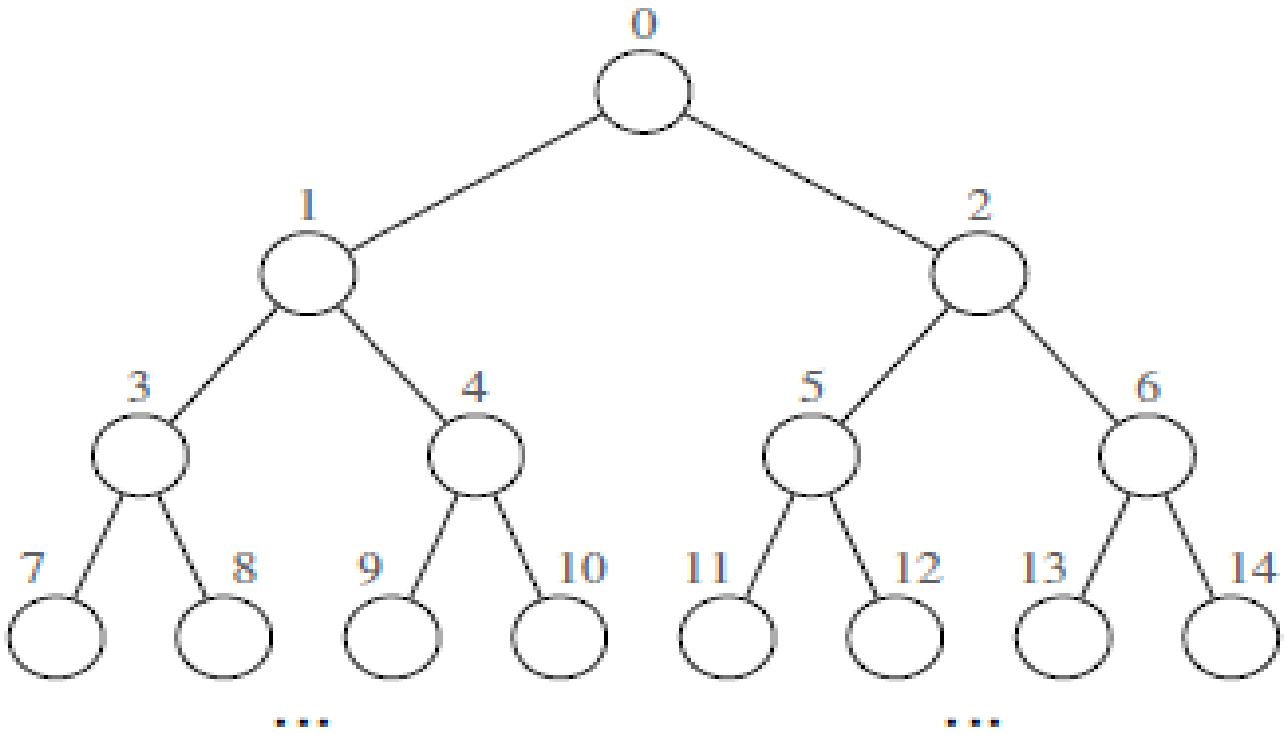
| Operation | Running Time |
|---|---|
| len, is_empty | $O(1)$ |
| root, parent, left, right, sibling, children, num_children | $O(1)$ |
| is_root, is_leaf | $O(1)$ |
| depth(p) | $O(d_p+1)$ |
| height | $O(n)$ |
| add_root, add_left, add_right, replace, delete, attach | $O(1)$ |

# Array-Based Representation of Binary Trees

- nodes are stored in an array
- f(p) : level numbering function of the positions in a binary tree

- If $p$ is the root of $T$,
  then $f(p) = 0$.
- If $p$ is the left child of position $q$,
  then $f(p) = 2f(q)+1$.
- If $p$ is the right child of position $q$,
  then $f(p) = 2f(q)+2$.

# Array Representation

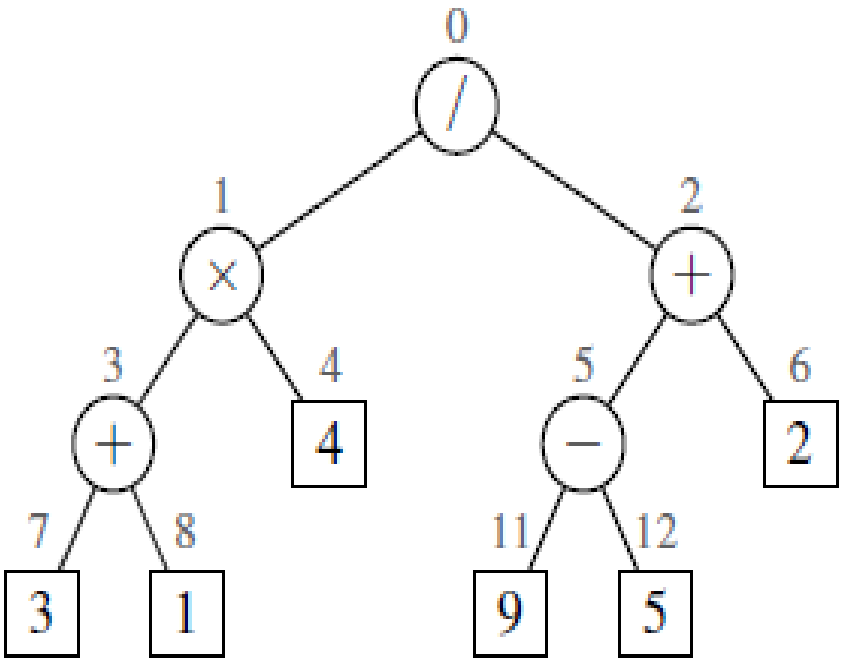



# Drawback of Arrays

- Not efficient for Sparse Tree
- Deletion and Addition of nodes has O(N) time complexity

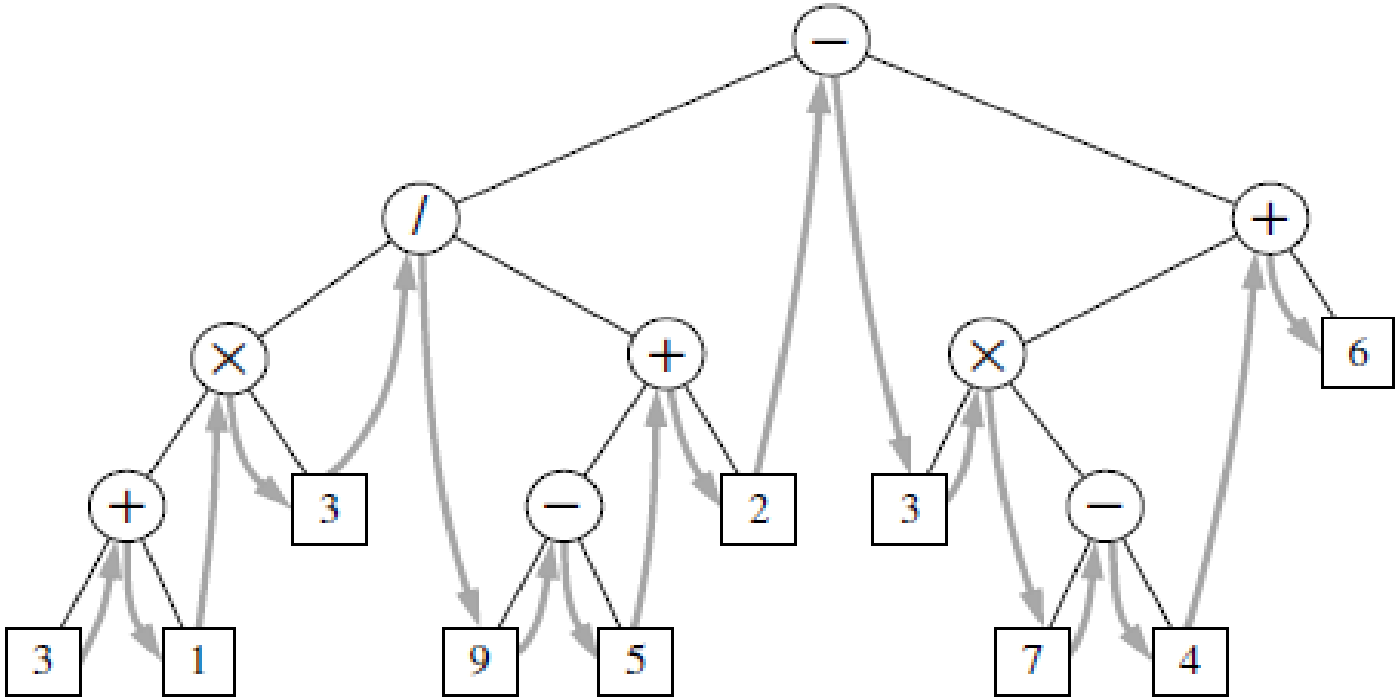
- Array size = N = 2^n -1 ; n : height of tree

# Array Representation

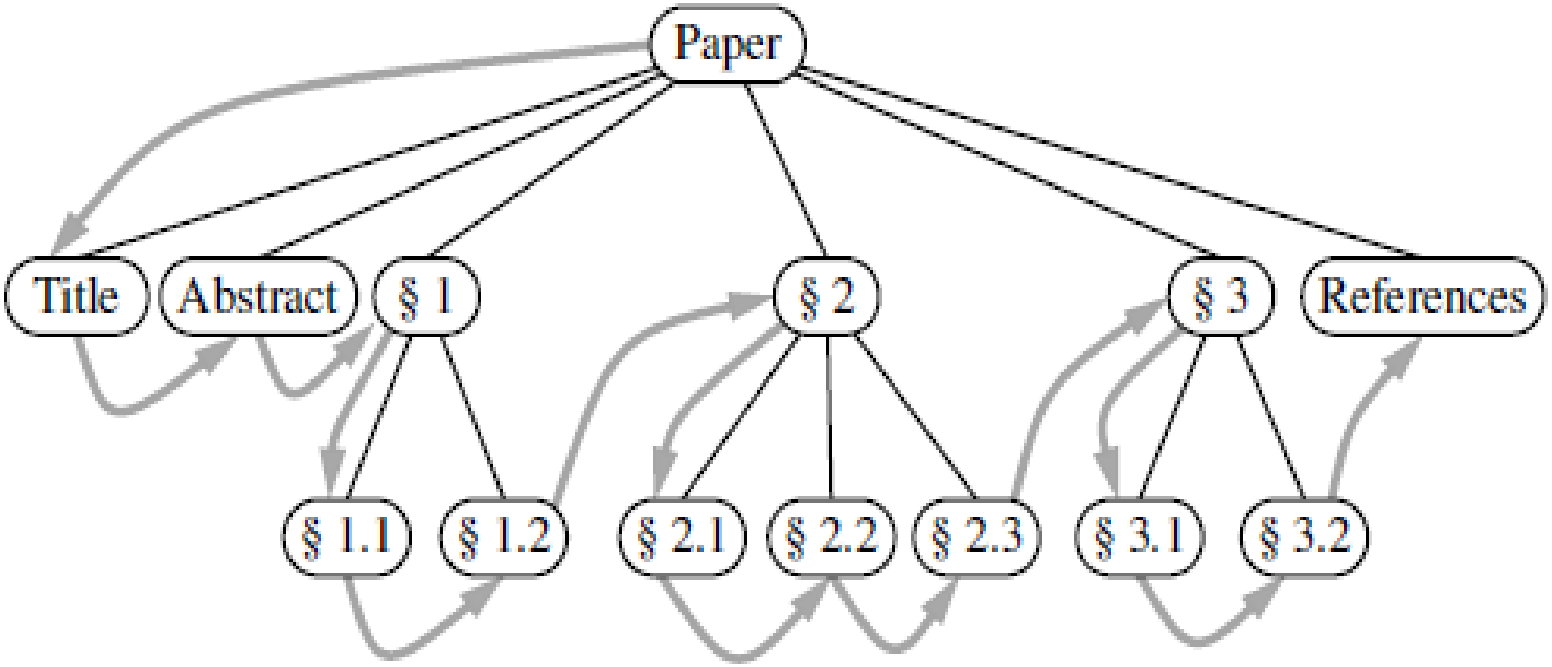



| / | × | + | + | 4 | − | 2 | 3 | 1 | | | 9 | 5 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

# Example



# Example



# Tree Traversals in Python

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

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

# Computing Depth of Node Define Recursively

If $p$ is the root,

the depth of $p$ is 0.

Else

the depth of $p$ is one plus the depth of the parent of $p$.

```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

# Depth of a Node

```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

Time Complexity : O(dp+1) → O(n)

# Computing Height of Tree

The ***height*** of a position $p$ in a tree $T$ is also defined recursively:

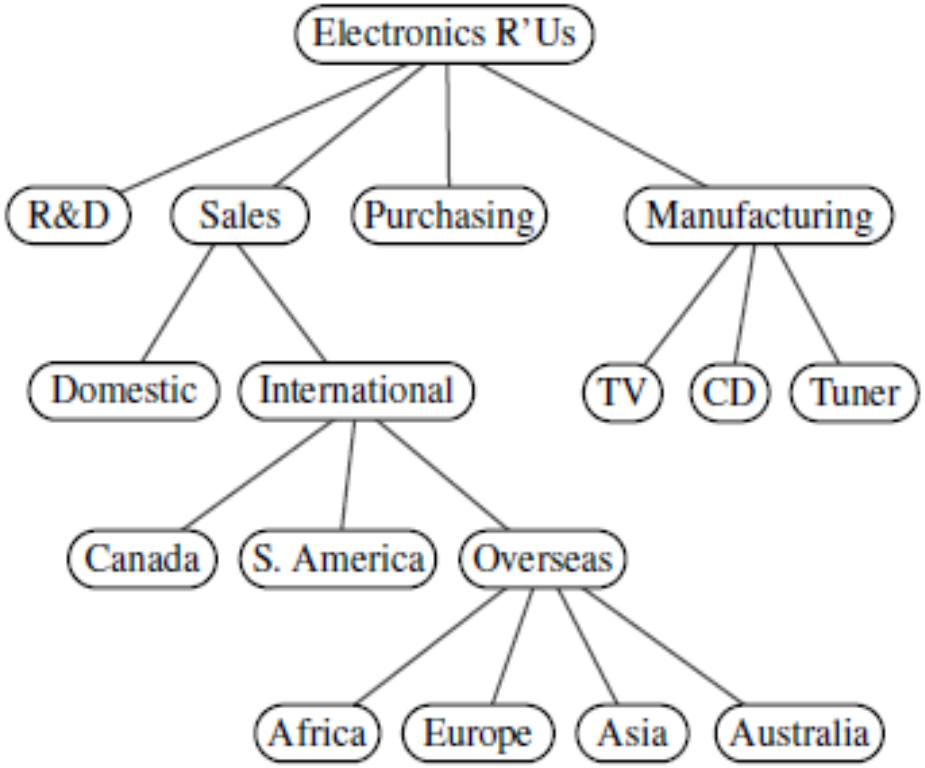
- If $p$ is a leaf, then the height of $p$ is 0.
- Otherwise, the height of $p$ is one more than the maximum of the heights of $p$'s children.

```
def height1(self):
    """Return the height of the tree."""
    return max(self.depth(p) for p in self.positions( ) if self.is leaf(p))
```

# Computing Height of Tree

```
def height2(self, p):
 """Return the height of the subtree rooted at
Position p."""
 if self.is leaf(p):
            return 0
 else:
            return 1 + max(self. height2(c) for c in
self.children(p))
```

# Parenthetic Representations of Tree



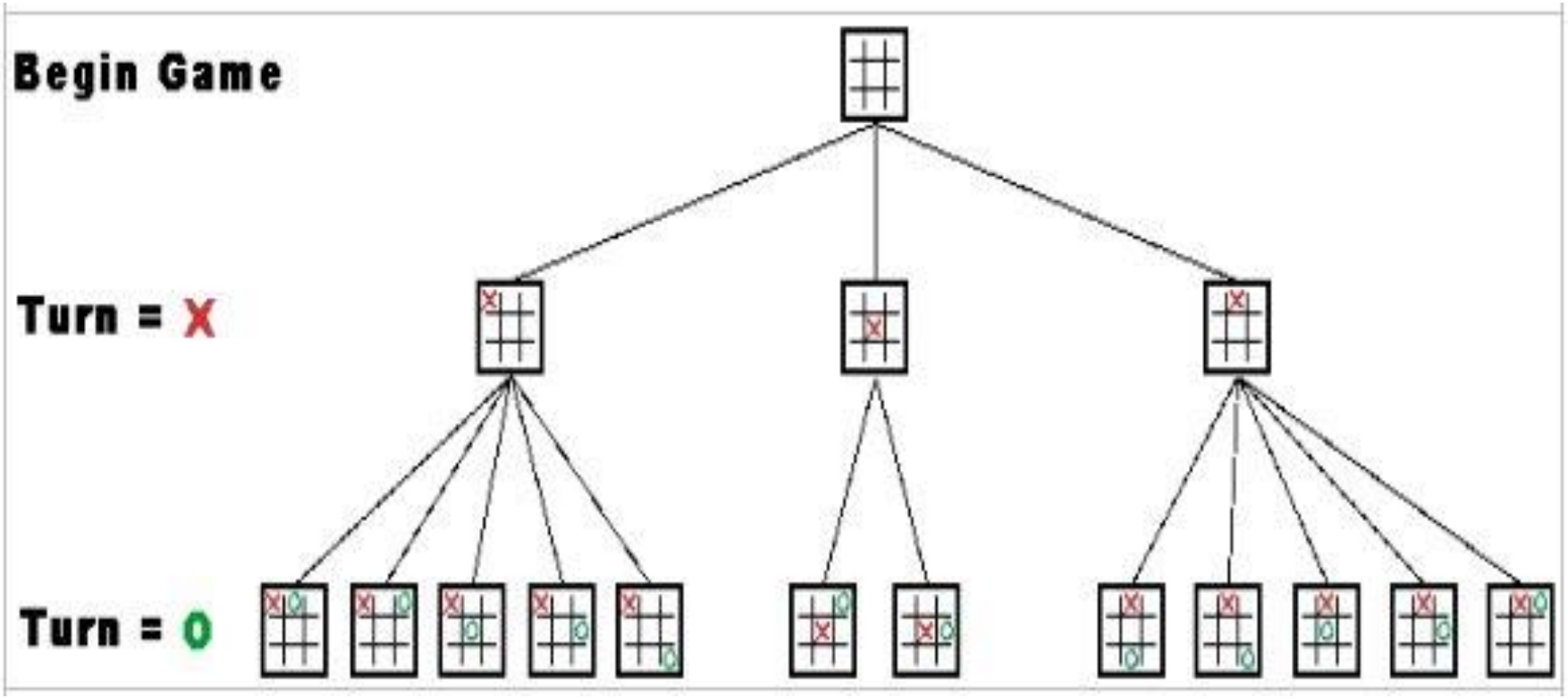

```
Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))
```

# Algorithm : Parenthesize

```
1  def parenthesize(T, p):
2    """Print parenthesized representation of subtree of T rooted at p."""
3    print(p.element( ), end='')          # use of end avoids trailing newline
4    if not T.is_leaf(p):
5      first_time = True
6      for c in T.children(p):
7        sep = ' (' if first_time else ', '          # determine proper separator
8        print(sep, end='')
9        first_time = False                # any future passes will not be the first
10       parenthesize(T, c)                # recur on child
11     print(')', end='')                  # include closing parenthesis
```

**Code Fragment 8.25:** Function that prints parenthetic string representation of a tree.

# Board Games : TIC-TAC-TOE



# Board Games : TIC-TAC-TOE

Node : Each node in the tree represents a potential state of a Tic-Tac-Toe game.

Each Node has

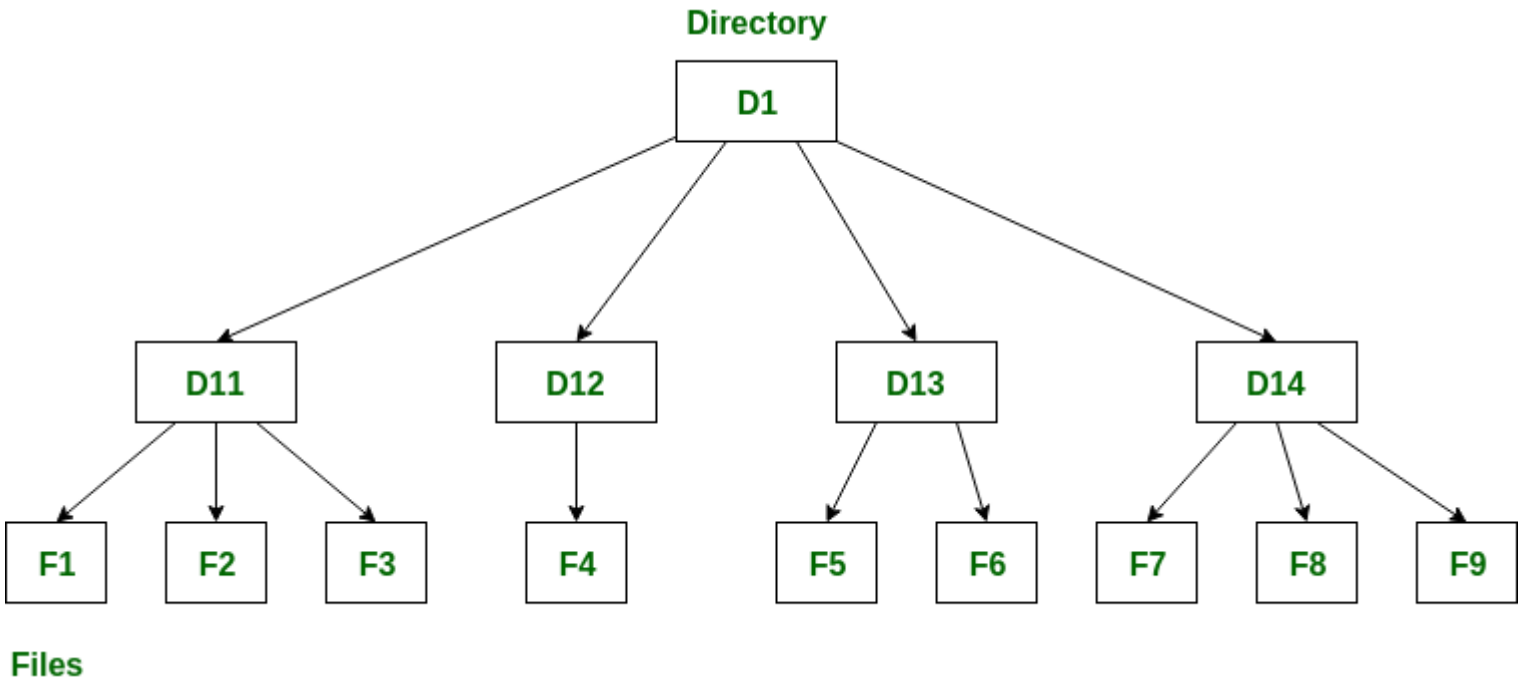
- A Board object with the current game state it represents,
- A marker for which player last played a turn on the board,
- A rating of either 'X Wins', 'O Wins', or 'Tie'
- A list of its child nodes.

# Algorithm

1. Generate the full Game Tree: start by creating the root node and populating its Board object with the empty Board, then create its children, populating their Board objects appropriately, and so on until you reach the leaf nodes. A leaf node is a node whose board represents a game in which some player has won or a tie has occurred.

2. Identify the winning strategy at each node of the game tree, which amounts to appropriately setting the rating variable of each node in the tree. To do so, you will implement the following recursive ranking algorithm:
    1. If a node *N* is a leaf node, set the node's rating as Marker.X, Marker O, or Marker.NONE to indicate that X has won, O has won, or there was a tie, respectively, in the node's associated board.
    2. If a node *N* is neither a leaf node nor a root node, then set its rating as follows:
        1. if at least one of *N*'s children has a rating that is opposite to *N*'s turn identifier, rate *N* with the value opposite to its turn identifier. For example, if N's turn identifier is Marker.X, and N has a child with rating Marker.O, then rate N with Marker.O.
        2. if all of *N*'s children have a rating equal to *N*'s turn identifier, rate *N* with its turn identifier.
        3. otherwise, rate this node as a tie (i.e., Marker.NONE).

# Directory Structure



# Is File Empty ?

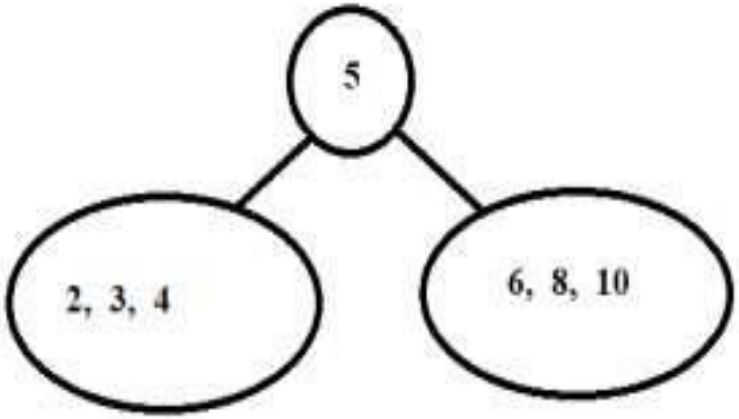
# Disk Usage

```
1  def disk_space(T, p):
2    """Return total disk space for subtree of T rooted at p."""
3    subtotal = p.element( ).space( )          # space used at position p
4    for c in T.children(p):
5      subtotal += disk_space(T, c)            # add child's space to subtotal
6    return subtotal
```

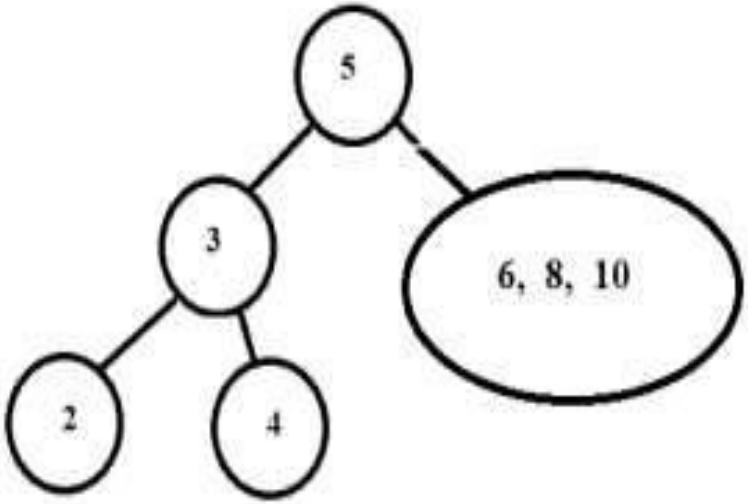
# Tree from Inorder and Preorder Traversal

Inorder : 2-3-4-5-6-8-10
Preorder : 5-3-2-4-8-6-10

Step 1



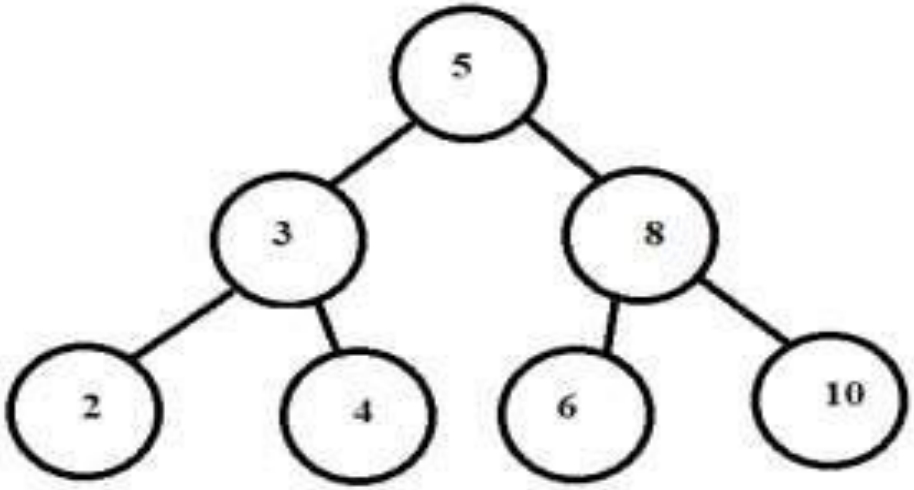

Step 2



# Tree from Inorder and Preorder Traversal

Inorder : 2-3-4-5-6-8-10
Preorder : 4-3-2-5-8-6-10

Step 3



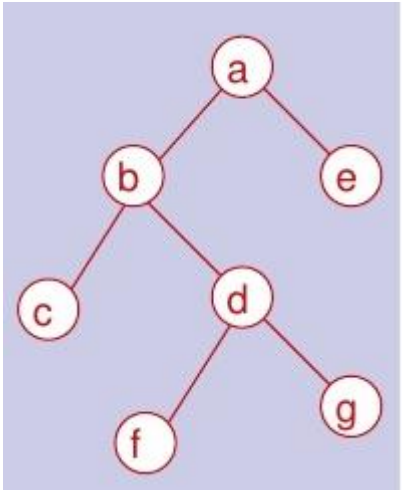
# Tree from Inorder and Preorder Traversal

Inorder : D-B-E-A-F-C
Preorder : A-B-D-E-C-F

ABCDEF

# Tree from Inorder and Preorder Traversal

Preorder : a-b-c-d-f-g-e
Inorder : c-b-f-d-g-a-e



# Algorithm : buildTree()

- Pick an element from Preorder.
- Increment a Preorder Index Variable (*preIndex*) to pick next element in next recursive call.
- Create a new tree node *tNode* with the data as picked element.
- Find the picked element's index in Inorder. Let the index be *inIndex*.
- Call buildTree for elements before *inIndex* and make the built tree as left subtree of *tNode*.
- Call buildTree for elements after *inIndex* and make the built tree as right subtree of *tNode*.
- return *tNode*.