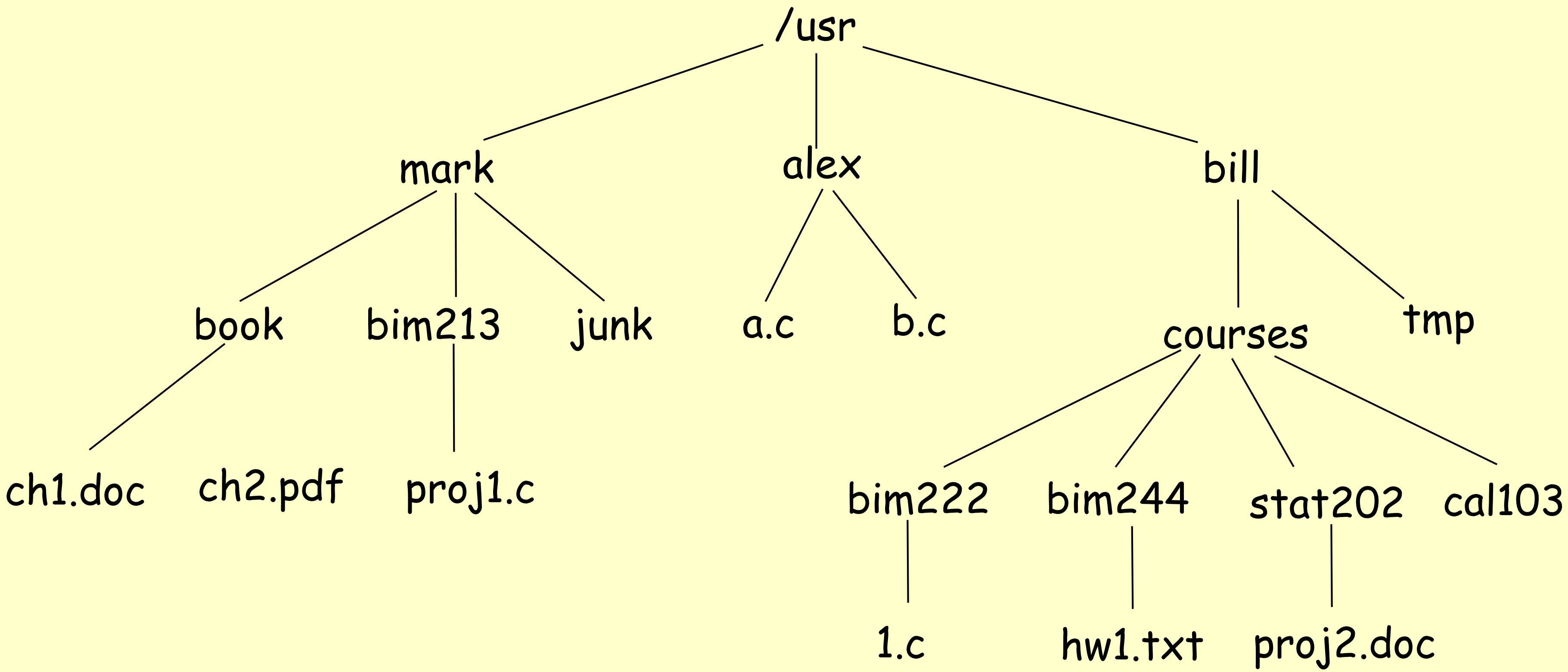
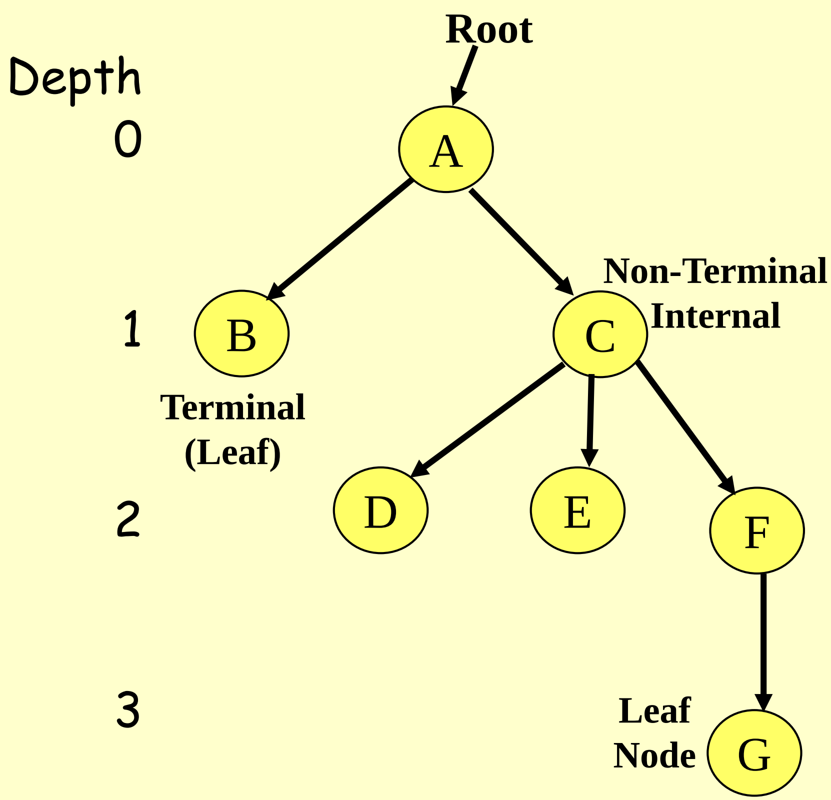
Storing Hierarchical Information

- Lists, Stacks, and Queues represent linear sequences
- Data often contain hierarchical relationships that cannot be expressed as a linear ordering
  - File directories or folders on your computer
  - Possible moves in a game (consider chess)
  - Employee hierarchies in organizations and companies
  - Family trees
  - …
  - Trees

Example Unix Directory Structure



Trees

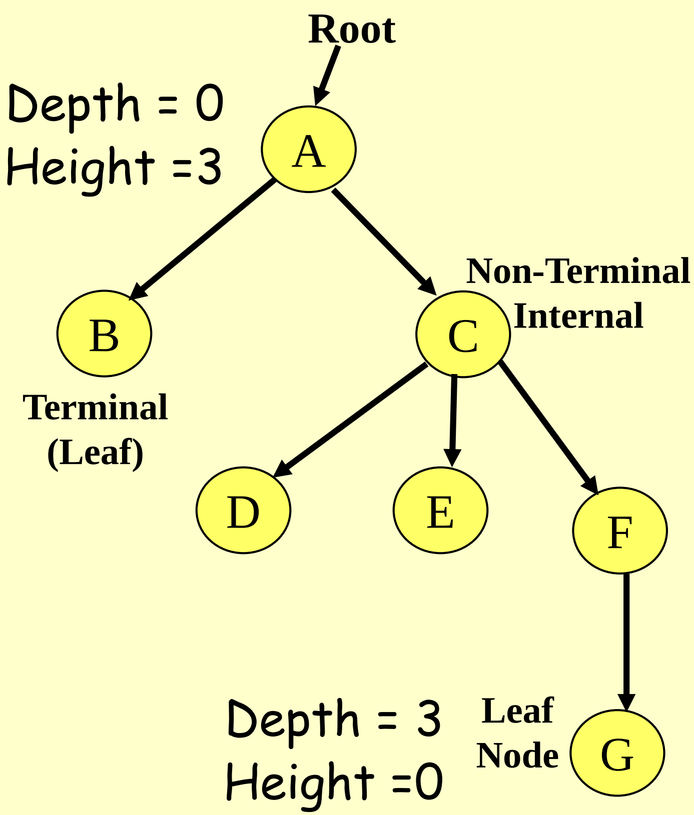


A tree of 7 nodes

• Arrows denote directed edges. Trees always contain directed edges, but arrows are usually omitted

- Basic Terminology
  - nodes and edges
  - Root
  - Subtrees
  - Parent
  - Children
  - Sblings
  - Degree = # of children of the node
  - Leaves
  - Path
  - Ancestors
  - Descendants
  - path length

More Tree Jargon

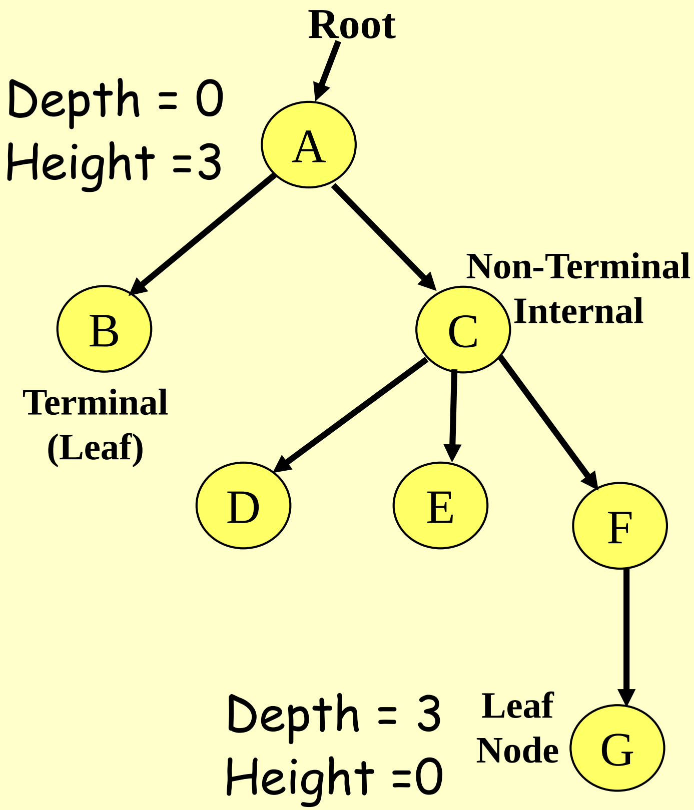


A tree of 7 nodes

Arrows denote directed edges. Trees always contain directed edges, but arrows are usually omitted

- Subtree= A subtree of a tree T, is a tree consisting of a node in T and all its descendants
- Parent= a node that has a child is called the child's parents
- Child= is a node that has a parent node
- Sblings= nodes share the same parent
- Ancestor= An ancestor of a node is any other node on the path from the node to the root
- Descendant= A descendant node of a node is any node in the path from that node to the lea
- Path= the sequence of nodes along the edges of a tree

More Tree Jargon



A tree of 7 nodes

Arrows denote directed edges. Trees always contain directed edges, but arrows are usually omitted

- Length of a path = number of edges
- Depth of a node N = length of path from root to N
- Height of node N = length of longest path from N to a leaf
- Depth and height of tree = ?

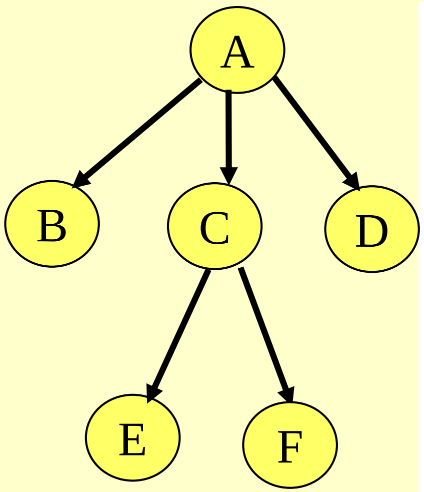
Trees

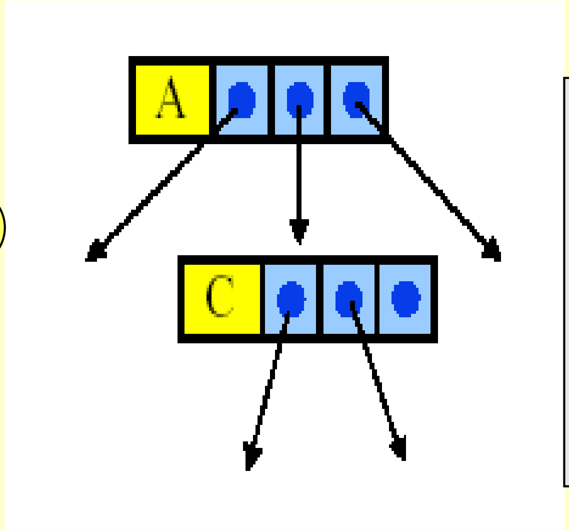
- Recursive Definition of a Tree:
  - A tree is a set of nodes that is either:
    a. an empty set of nodes, or
    b. has one node called the root from which zero or more trees ("subtrees") descend.

- A tree with N nodes always has N-1 edges
- Two nodes in a tree have at most one path between them
- Can a non-zero path from node N reach node N again?
  - No! Trees can never have cycles.

Implementation of Trees: The obvious

- Obvious Implementation: Node with value and links to children





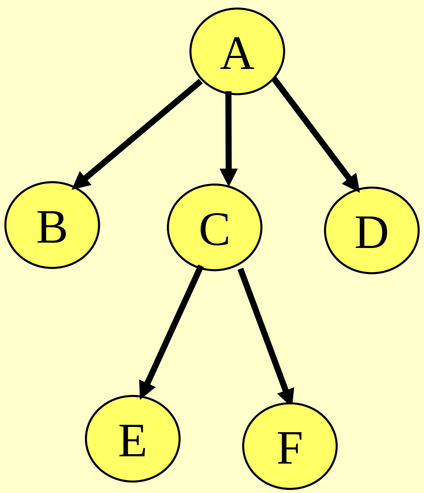
Java Declaration

```
class TreeNode {
    int key;
    TreeNode children[]=new TreeNode[3];
}
```

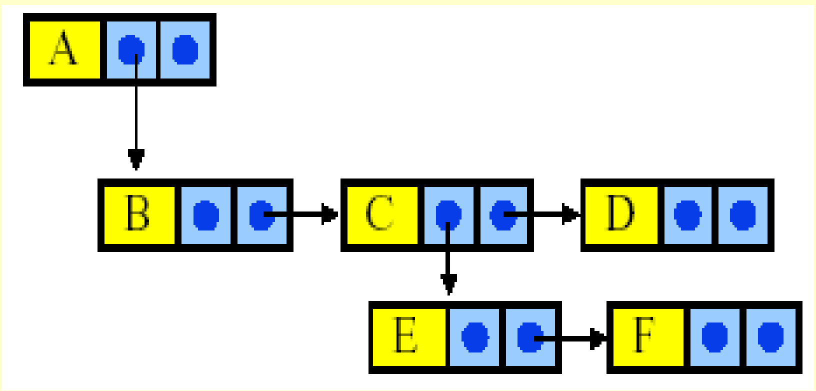
- Problem: Do not know number of children for each node in advance. Wastes space if maximum number of links assumed.

1st Child/Next Sibling Representation

- Better Implementation: 1st Child/Next Sibling Representation
  - Each node has 2 pointers: one to its first child and one to next sibling
  - Can handle arbitrary number of children



Pictorial View



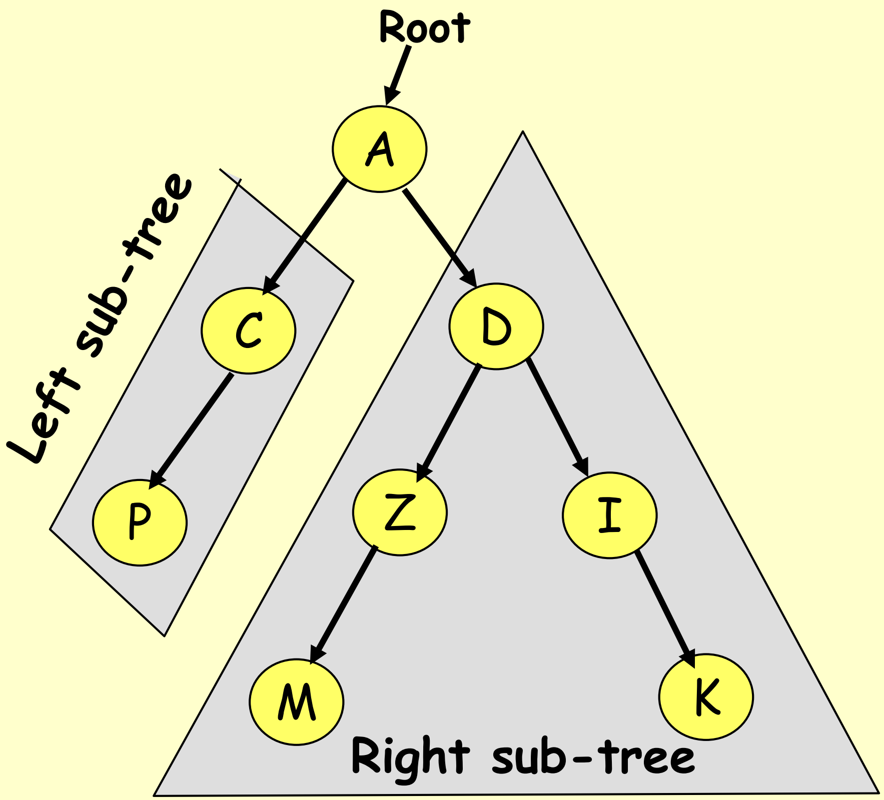
1st Child/Next Sibling Representation

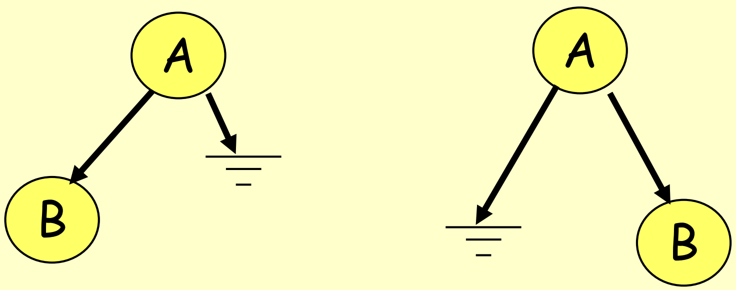
Java Declaration

```
class TreeNode {
    int key;
    TreeNode firstChild;
    TreeNode sibling;
}
```

Binary Trees

- A binary tree is an ordered-tree where the degree of each node <= 2
  - Each node has at most 2 children
    - Most popular tree in computer science

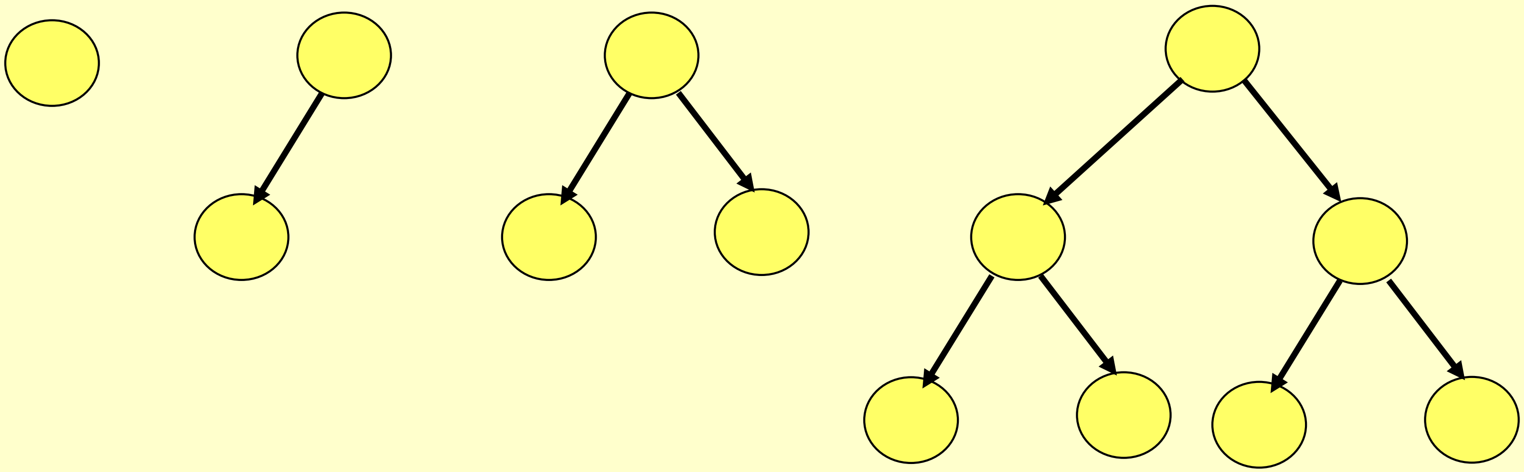




Two different binary trees

Binary Trees (more)

- Given N nodes, what is the minimum depth of a binary tree?



Depth 0: $N = 1 = 2^0$ nodes

Depth 1: $N = 2$ to 3 nodes $= 2^1$ to $2^{1+1} - 1$ nodes

At depth d, N = ?

Binary Trees (more)

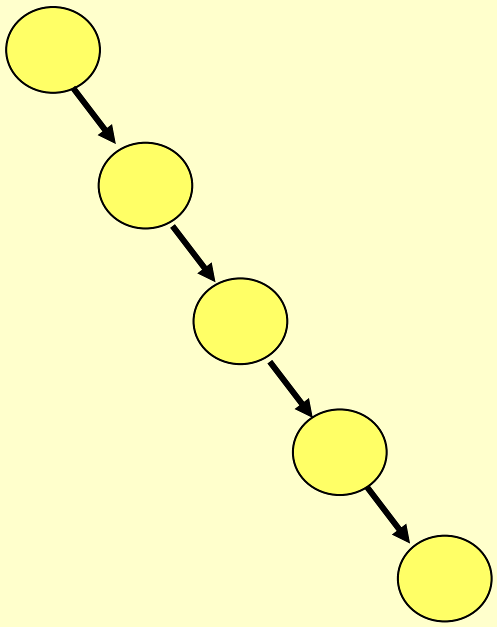
- Depth 0: $N = 1 = 2^0$ nodes
- Depth 1: $N = 2$ to 3 nodes $= 2^1$ to $2^{1+1} - 1$ nodes
- At depth d, $N = 2^d$ to $2^{d+1}-1$ nodes (a complete binary tree)

- So, minimum depth d is:

$\log(N+1)-1 \leq d \leq \log N$ or $\Theta(\log N)$

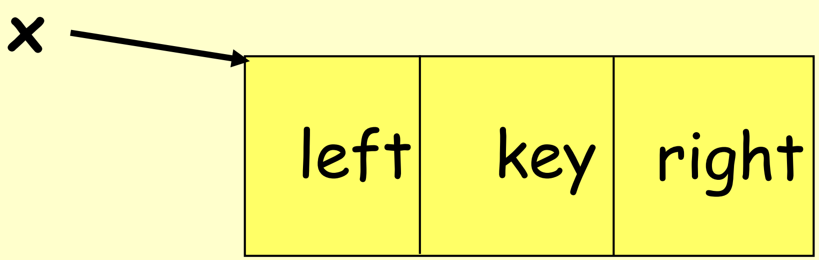
Binary Trees (more)

- Minimum depth of N-node binary tree is $\Theta(\log N)$
- What is the maximum depth of a binary tree?
  - Degenerate case: Tree is a linked list!
  - Maximum depth = N-1
- Goal: Would like to keep depth at around log N to get better performance than linked list for operations like Search

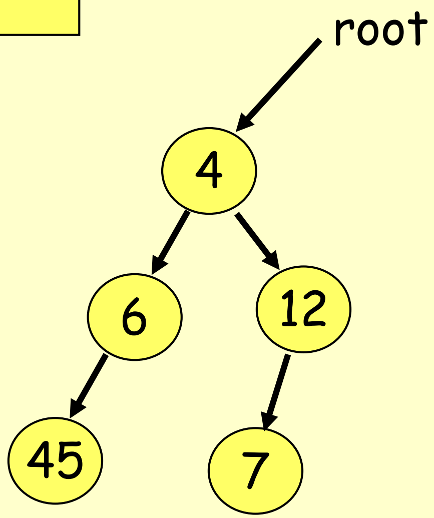


- A degenerate tree:
  - A Linked List
  - Depth = N-1

Linked Implementation of Binary Trees

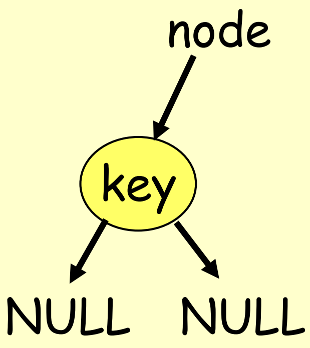



```
Java Declaration

class BinaryTreeNode {
    public BinaryTreeNode left;
    public int key;
    public BinaryTreeNode right;
};
```



Linked Implementation of Binary Trees

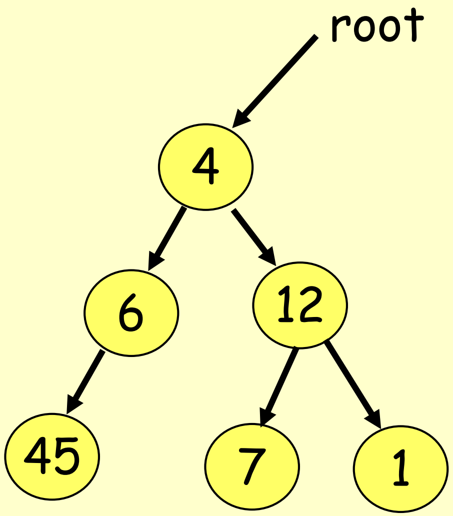
```
/* Creates, initializes and returns
 * a binary tree node
 */
BinaryTreeNode CreateNode(int key){
  BinaryTreeNode node = new BinaryTreeNode();
  node.key = key;
  node.left = null;
  node.right = null;

  return node;
} /* CreateNode */
```



- CreateNode function creates and initializes a BinaryTreeNode

Linked Implementation of Binary Trees

```
BinaryTreeNode root = null;

main(){
  root = CreateNode(4);

  root.left = CreateNode(6);
  root.right = CreateNode(12);

  root.left.left = CreateNode(45);

  root.right.left = CreateNode(7);
  root.right.right = CreateNode(1);
} /* main */
```



- Given a pointer to the root, how do you go over all tree nodes and print them out?
  - Tree traversal algorithms (preorder, inorder, postorder)

Binary Tree Traversal

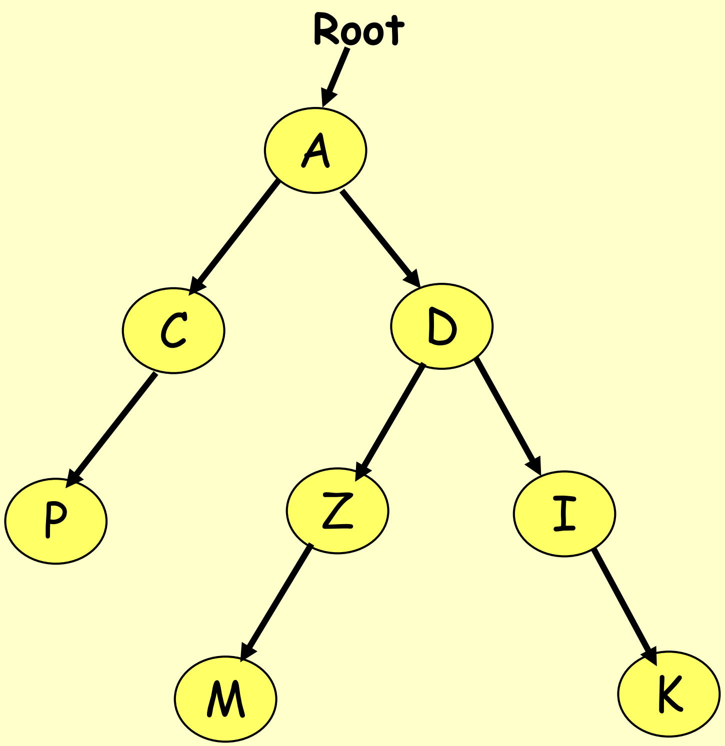
- 3 principal ways to traverse a binary tree defined with respect to the order in which the root is visited:
  - Preorder
  - Inorder
  - Postorder

Preorder Traversal
- Visit the root
- Traverse the left sub-tree
- Traverse the right sub-tree

Inorder Traversal
- Traverse the left sub-tree
- Visit the root
- Traverse the right sub-tree

Postorder Traversal
- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit the root

Example Binary Tree Traversal Results



Preorder Traversal Results

A C P D Z M I K

Inorder Traversal Results

P C A M Z D I K

Postorder Traversal Result

P C M Z K I D A

Traversal Algorithms

```
PreorderTraversal(BinaryTreeNode  root){
    if (root == null) return;
    println(root.key);
    PreorderTraversal(root.left);
    PreorderTraversal(root.right);
} //end-PreorderTraversal
```

```
InorderTraversal(BinaryTreeNode  root){
    if (root == null) return;
    InorderTraversal(root.left);
    println(root.key);
    InorderTraversal(root.right);
} //end-InorderTraversal
```

```
PostorderTraversal(BinaryTreeNode root){
    if (root == null) return;
    PostorderTraversal(root.left);
    PostorderTraversal(root.right);
    println(root.key);
} //end-PostorderTraversal
```

Preorder Traversal With a Stack

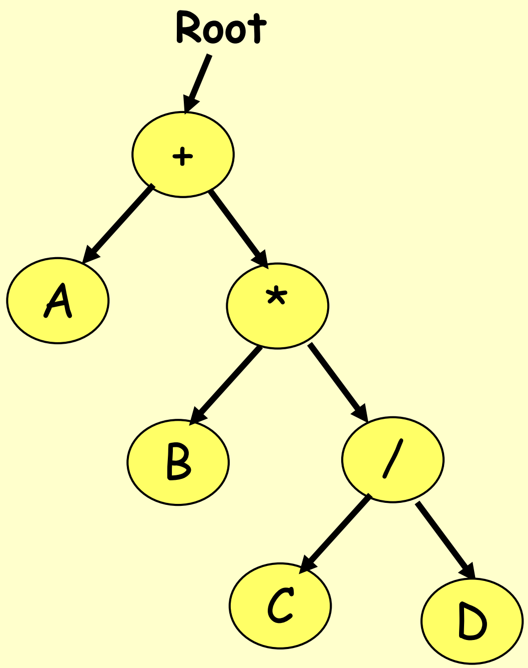
```
void StackPreorder (SBinaryTreeNode root) {
  if (root == null) return;

  Stack S = new Stack(); // Stack holds pointers to SBinaryTreeNode
  S.Push(root);

   while (!S.isEmpty()) {
      BinaryTreeNode x = S.Pop();
      println(" " + x.key);

      if (x.right != null) S.Push(x.right);
      if (x.left != null) S.Push(x.left);
    } //end-while
} //end-StackPreorder
```

Level-by-Level Traversal With a Queue

```
void LevelByLevelTraversal (SBinaryTreeNode root) {
  if (root == null) return;

  Queue Q = new Queue();   // Queue holds pointers to SBinaryTreeNode
  Q.Enqueue(root);

   while (!Q.isEmpty()) {
      BinaryTreeNode x = Q.Dequeue();
      println(" "+ x.key);

      if (x.left != null) Q.Enqueue(x.left);
      if (x.right != null) Q.Enqueue(x.right);
    } //end-while
} //end-LevelByLevelTraversal
```

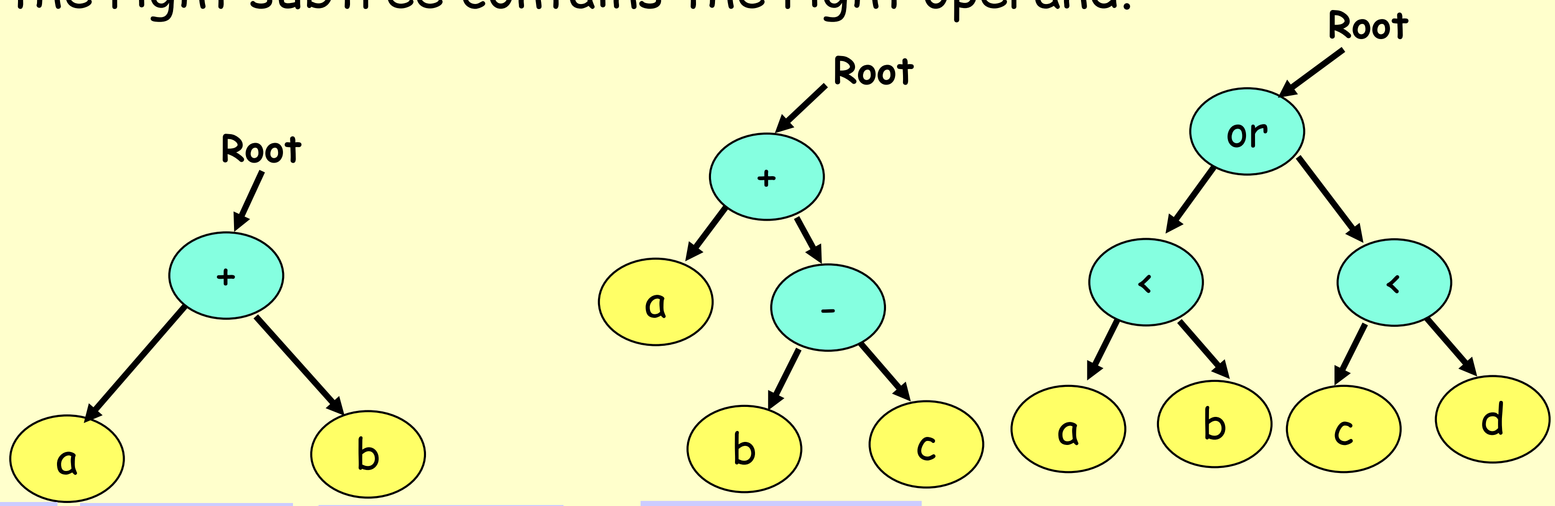
Using Binary Trees: Expression Trees

- Example Arithmetic Expression
  - A + (B * (C / D))
- Tree for the above expression:
  - Leaves = operands (constants/variables)
  - Non-leaf nodes = operators
- Used in most compilers
- No parenthesis needed – use tree structure
- Can speed up calculations e.g. replace / node with C/D if C and D are known
- Also allows optimization



Using Binary Trees: Expression Trees

- For a binary operator such as +, -, *, /
  - the root contains the operator,
  - the left subtree contains the left operand
  - the right subtree contains the right operand.



In: a + b

Pre: +a b

Post: a b +

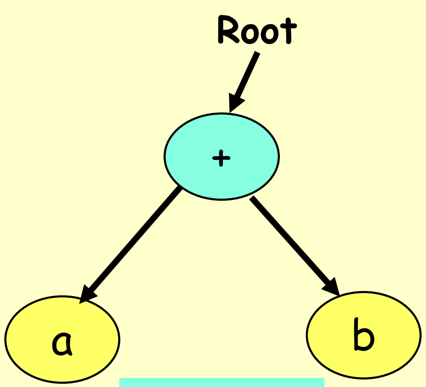
In: a + (b-c)

In: (a < b) or (c < d)

- Names of the traversal methods are related to Polish Form of the expression:
  - Pre-order traversal: Prefix form
  - In-order traversal: Infix form
  - Post-order traversal: Postfix form or Reverse Polish notation

Using Binary Trees: Expression Trees

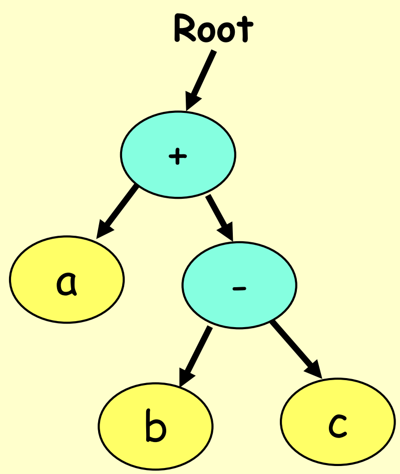
- Names of the traversal methods are related to Polish Form of the expression:
  - Pre-order traversal: Prefix Form
  - In-order traversal: Infix form
  - Post-order traversal produces Post-fix form



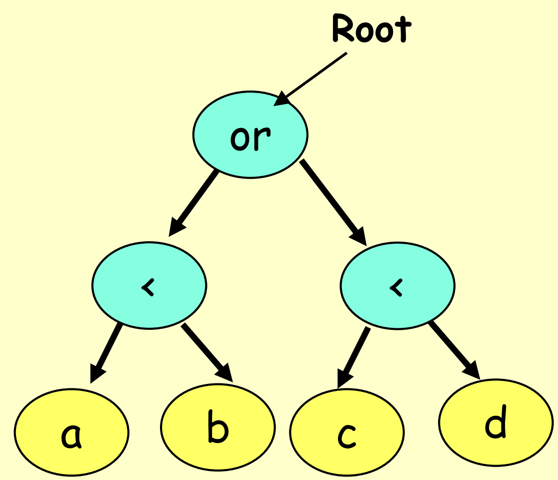
In: a + b

Pre: +a b

Post: a b +

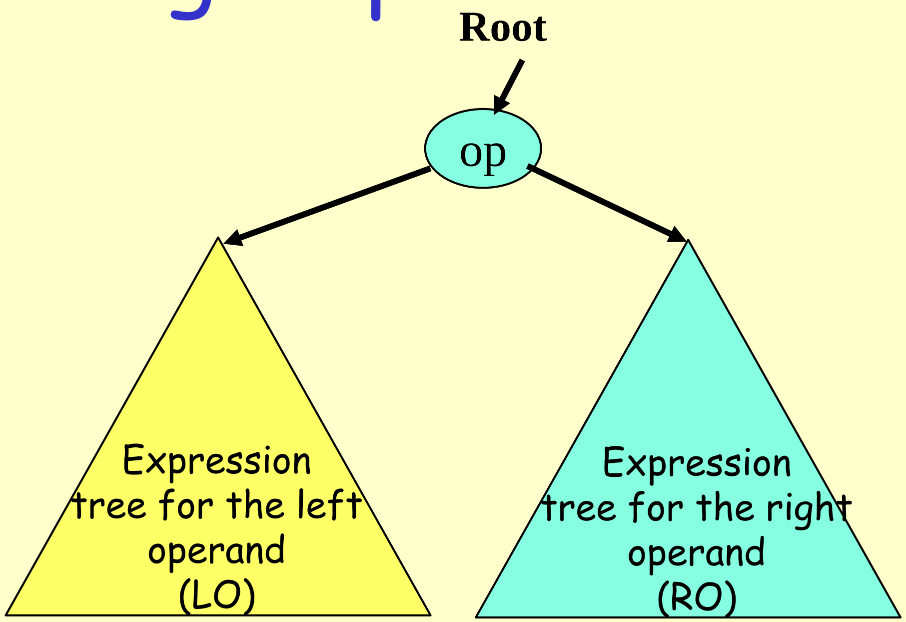


In: a + (b-c)



In: (a < b) or (c < d)

Evaluating Expression Trees



(LO) op (RO)

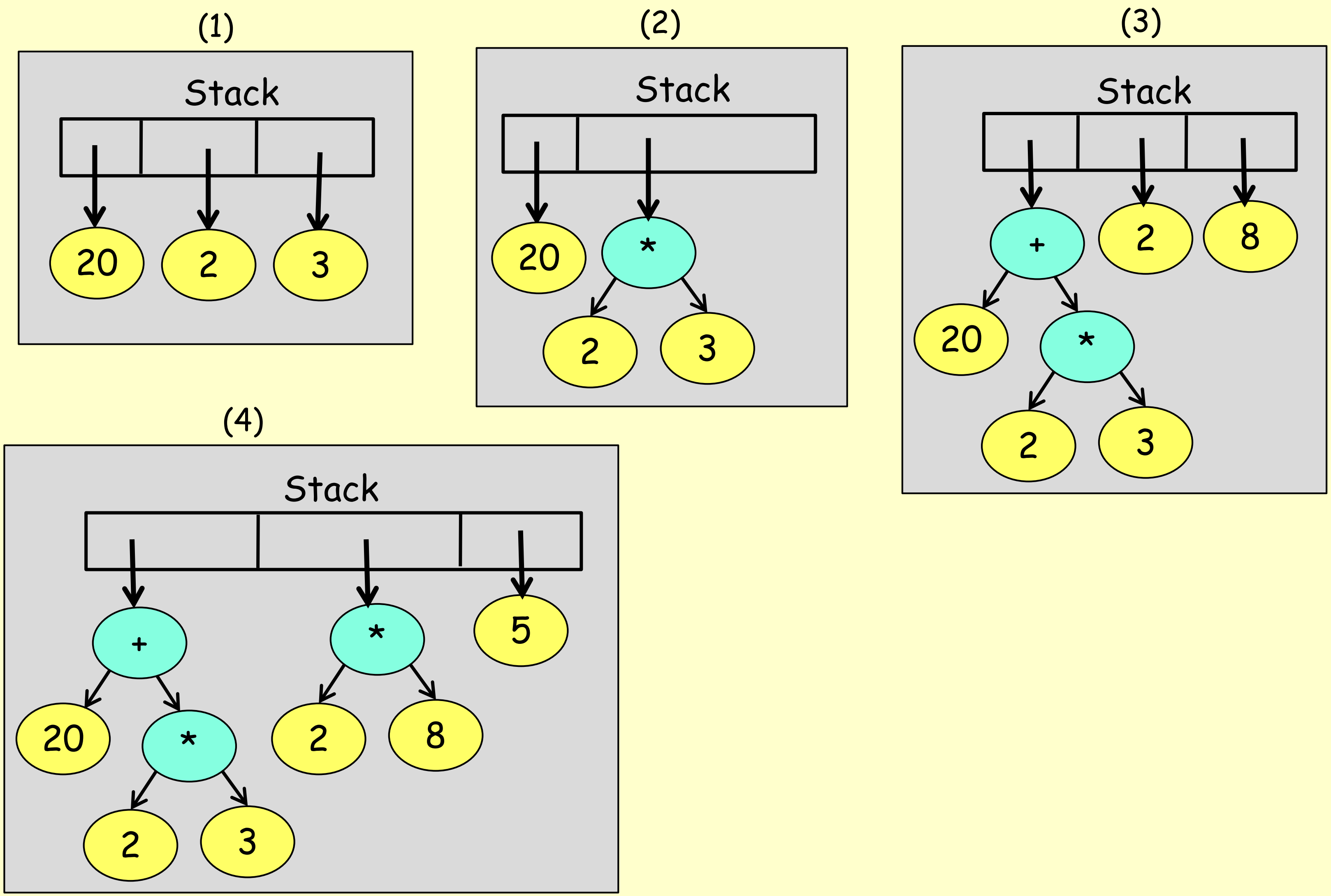
- Observation: To evaluate the above expression consisting of a binary operator and 2 operands do:
  - Evaluate the left operand (LO)
  - Evaluate the right operand (RO)
  - Apply the operator
- This is precisely a post-order traversal of the expression tree.

Postfix – 2- Expression Tree (1)

- It is very easy to create an expression tree from a postfix expression with the help of a stack
  - E.g. 20 + 2*3 + (2*8+5) *4
  - Postfix equivalent of this expression is
  - 20 2 3 * + 2 8 * 5 + 4 * +
  - Now convert this postfix expression to an expression tree with the help of a stack

Postfix – 2- Expression Tree (2)

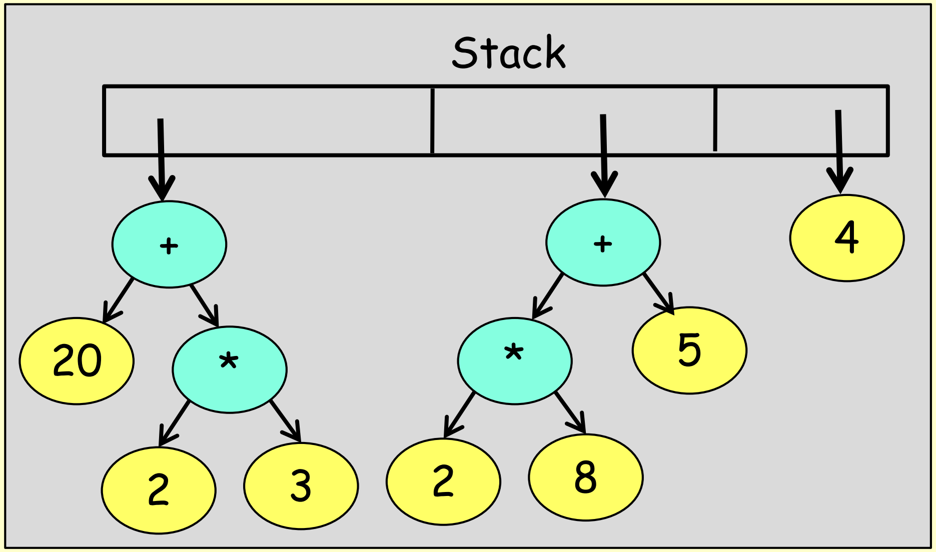
- Postfix Expr: “20 2 3 * + 2 8 * 5 + 4 * +”



Postfix – 2- Expression Tree (3)

- Postfix Expr: “20 2 3 * + 2 8 * 5 + 4 * +”

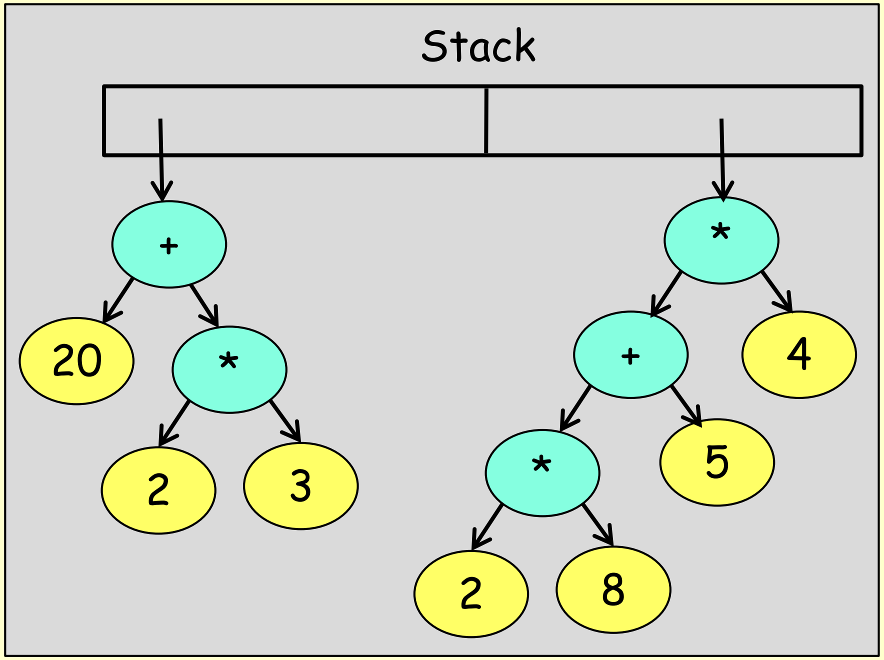
(5)



Postfix – 2- Expression Tree (4)

- Postfix Expr: “20 2 3 * + 2 8 * 5 + 4 * +”

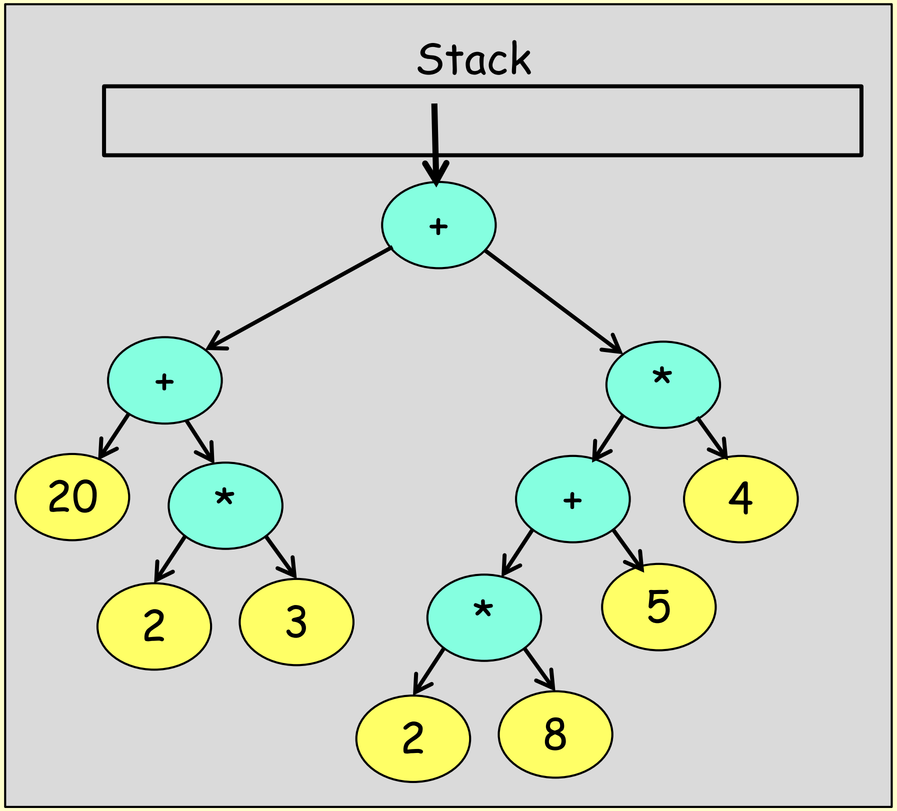
(6)



Postfix – 2- Expression Tree (5)

- Postfix Expr: “20 2 3 * + 2 8 * 5 + 4 * +”

(7)



Expression Trees: Last Word

- Look at the book for the details of the expression manipulation algorithms
  - Chapter 3 has an algorithm to convert an infix expression to a postfix expression using a stack
  - Chapter 4 has an algorithm to construct an expression tree from a postfix expression