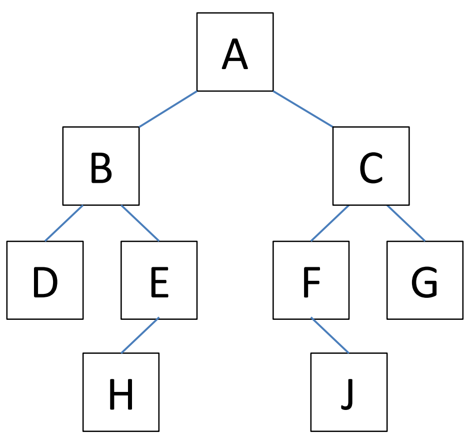
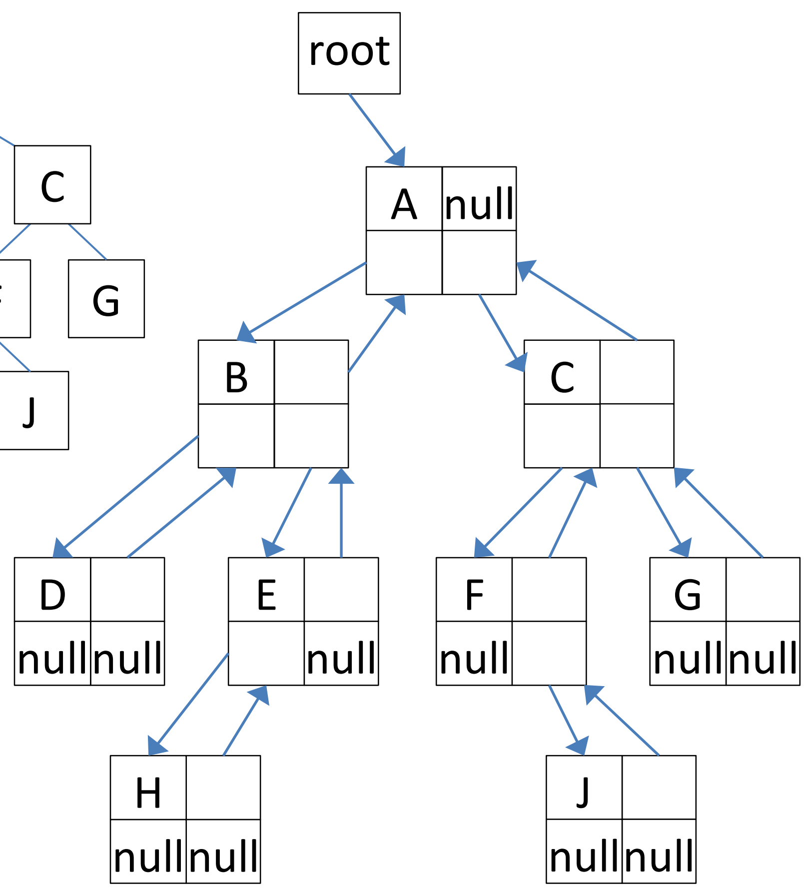
Binary Trees and Traversals

Binary Trees:

```
class Node {
        ElementType data;
        Node *parent, *leftChild, *rightChild;
}
class BinaryTree {
        Node *root;
}
```





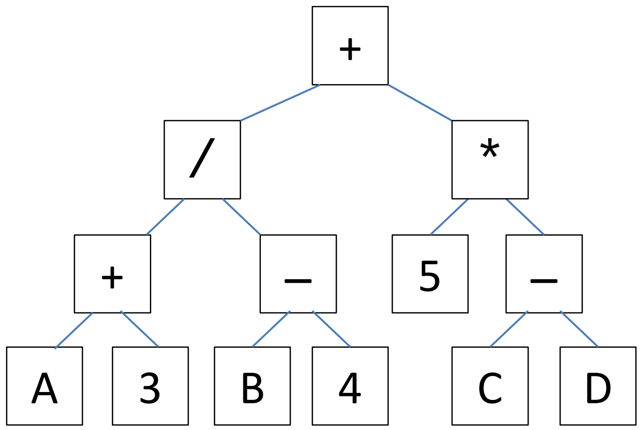
Node:

data	parent
leftC	rightC

Applications:

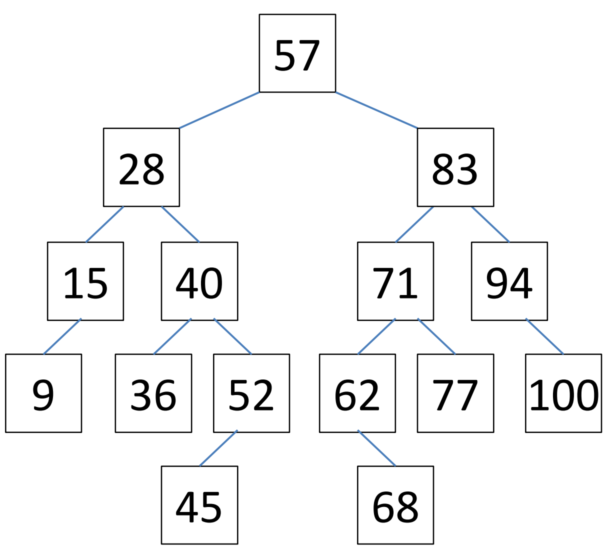
Arithmetic Expression Trees

Example: (A+3)/(B–4) + 5*(C–D)



Binary Search Trees (more about these later in course)

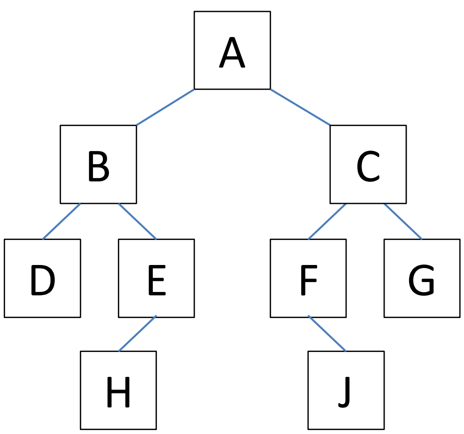
- If node X is in left subtree of node Y, then X.data ≤ Y.data
- If node Z is in right subtree of node Y, then Z.data ≥ Y.data

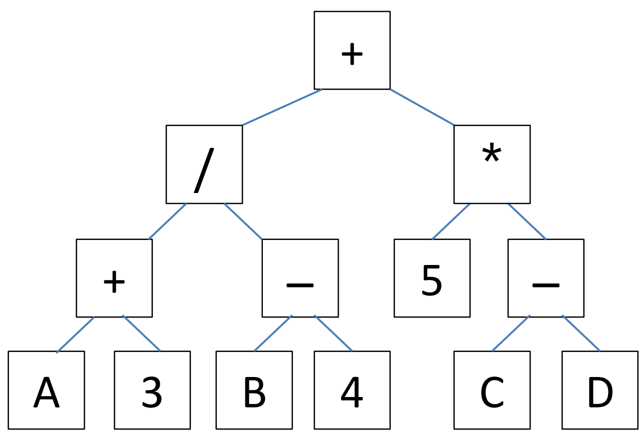


Traversals of a binary tree:

Preorder traversal

```
void preorder (BinaryTree T) {
    preorder (T->root);
}
void preorder (Node *p) {
    if (p==null) return;
    visit (p);              // for example:  print (p.data);
    preorder (p->leftChild);
    preorder (p->rightChild);
}
```





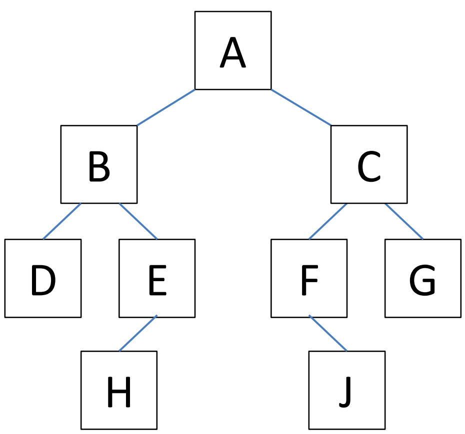
A B D E H C F J G

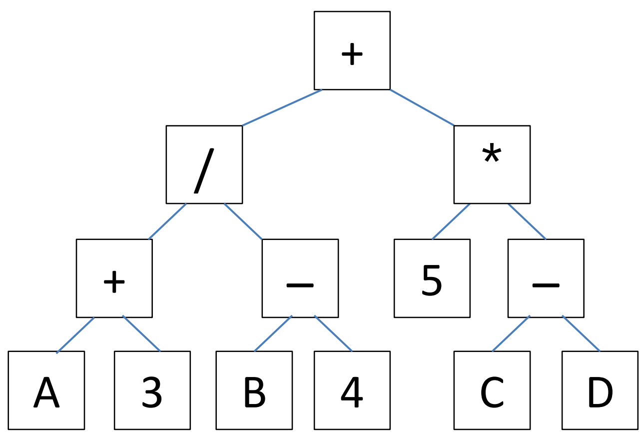
+ / + A 3 – B 4 * 5 – C D

Prefix notation

Postorder traversal

```
void postorder (BinaryTree T) {
    postorder (T->root);
}
void postorder (Node *p) {
    if (p==null) return;
    postorder (p->leftChild);
    postorder (p->rightChild);
    visit (p);
}
```





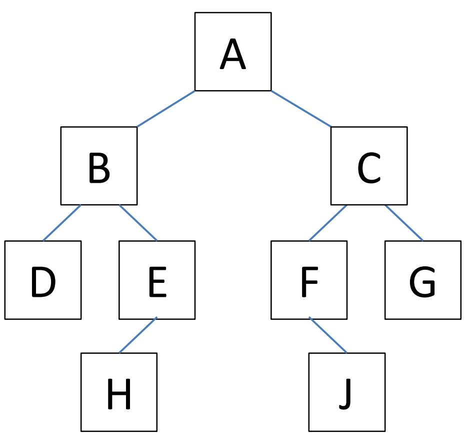
D H E B J F G C A

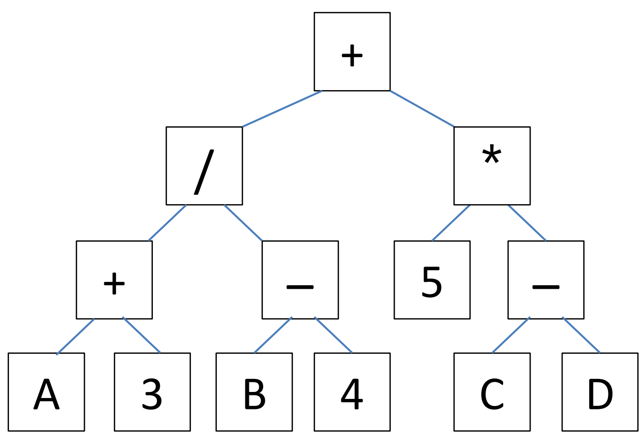
A 3 + B 4 – / 5 C D – * +

Postfix notation

Inorder traversal

```
void inorder (BinaryTree T) {
    inorder (T->root);
}
void inorder (Node *p) {
    if (p==null) return;
    inorder (p->leftChild);
    visit (p);
    inorder (p->rightChild);
}
```





D B H E A F J C G

A + 3 / B – 4 + 5 * C – D

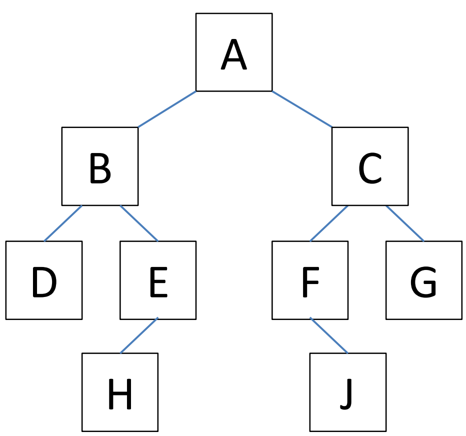
Infix notation

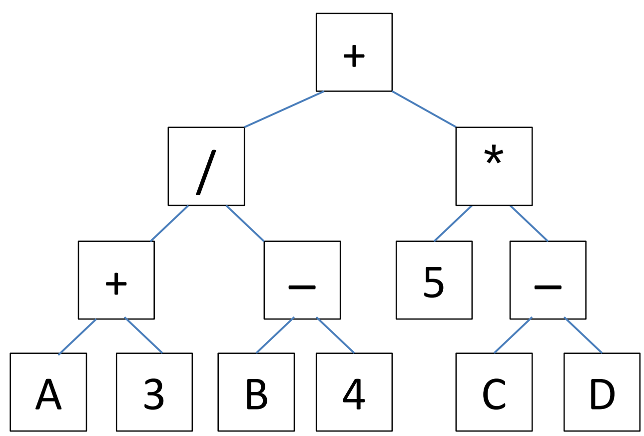
((A + 3) / (B – 4)) + (5 * (C – D))

Parenthesized infix

Level-order traversal

```
void levelOrder (BinaryTree T) {
    Queue Q( );
    Q.enqueue (T.root);
    while (not Q.isEmpty( )) {
            Node *p = Q.dequeue( );
            visit (p);
            if (p->leftChild != null)  Q.enqueue (p->leftChild);
            if (p->rightChild != null)  Q.enqueue (p->rightChild);
    }
}
```





A B C D E F G H J

+ / * + – 5 – A 3 B 4 C D

Analysis: Let n = number of nodes in the tree. Each kind of traversal spends $\theta(1)$ time at each node of the tree, so each traversal has $\theta(n)$ total running time.

Evaluating a postfix expression (use stack or recursion)

```
evaluatePostfix( ) {
	Stack S( );
	while (op = read( )) {
		if (op is operand)
			S.push (op);
		else if (op is operator) {
			right = S.pop( );
			left = S.pop( );
			S.push (apply (op, left, right));
		}
	}
	return S.pop( );
}
```

Example: 3 4 + 9 1 – *

op		Stack (from bottom to top)
3		3
4		3 4
+	apply (+, 3, 4)	7
9		7 9
1		7 9 1
–	apply (–, 9, 1)	7 8
*	apply (*, 7, 8)	56

Evaluating a prefix expression (use stack or recursion)

```
evaluatePrefix( ) {
    op = read( );
    if (op is operand)
        return op;
    else if (op is operator) {
        left = evaluatePrefix( );
        right = evaluatePrefix( );
        return apply (op, left, right);
    }
}
```

Example: * + 3 4 – 9 1

op = *

Recursively read and evaluate + 3 4 $\Rightarrow$ left = 7

Recursively read and evaluate – 9 1 $\Rightarrow$ right = 8

apply (*, 7, 8) $\Rightarrow$ return 56

Another application of stacks: Matching Paired Symbols

Examples of paired symbols: ( ) , [ ] , { } , < >

Example of input string: { ([ ]<>) { ([ ]) }<{} ( )>}

```
bool isBalanced (string input) {
        Stack S( );
        for (k=0; k<input.length( ); k++) {
                c = input[k];
                if (c is left symbol of a pair)
                        S.push (c);
                else if (c is right symbol of a pair) {
                        if (S.isEmpty( )) return false;
                        b = S.pop( );
                        if (b and c do not form a matching pair)
                                return false;
                }
        return S.isEmpty( ));
}
```

Trace this algorithm using the above input string