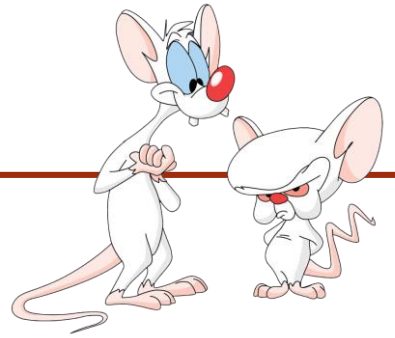
COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Week 11-3 : Binary Trees

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?

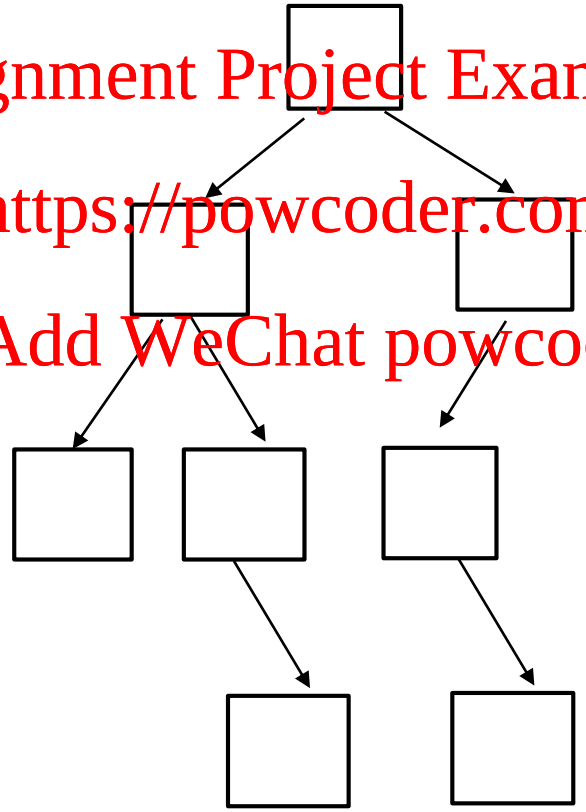


- Binary Trees
  - Expression Trees

BINARY TREES

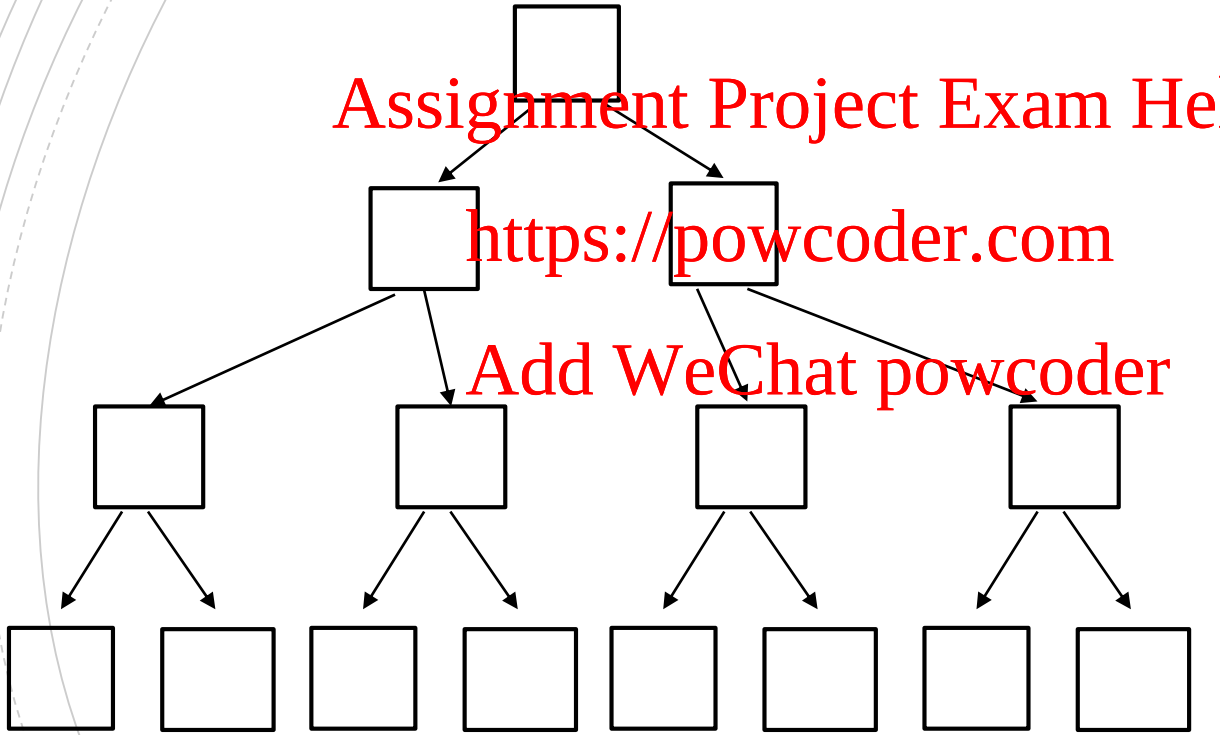
BINARY TREES

Each node has at most 2 children!



BINARY TREES

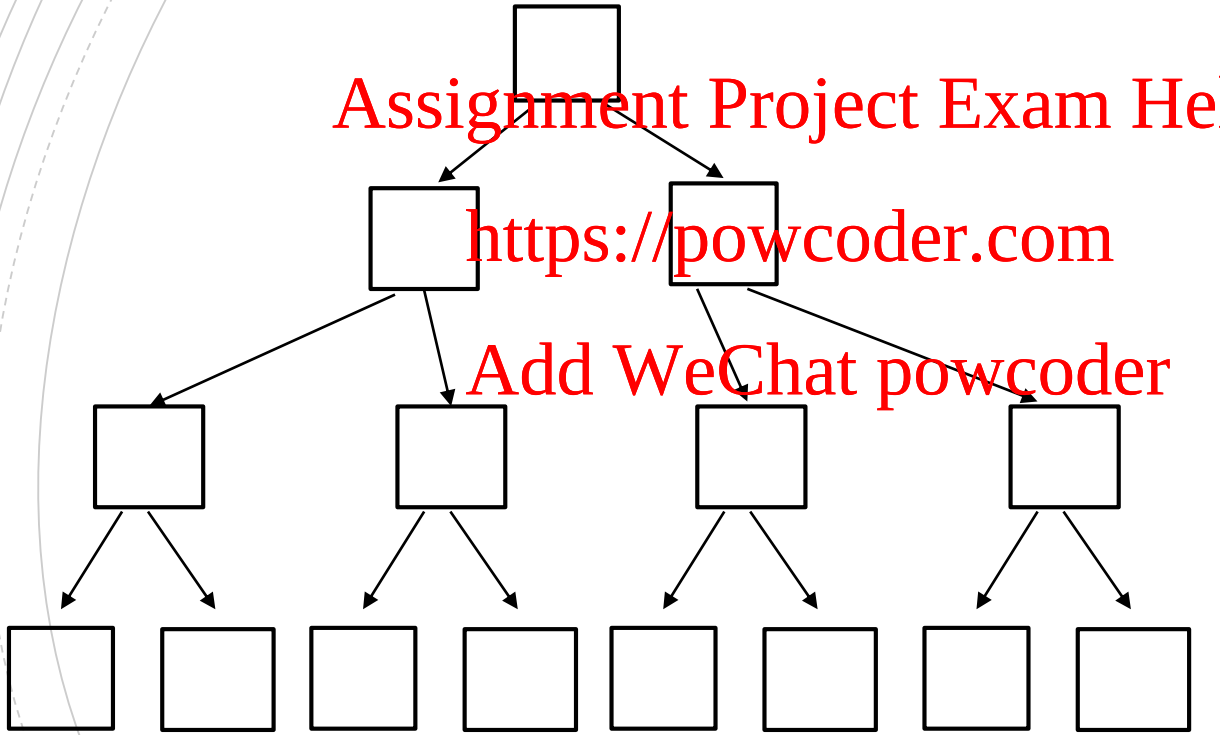
Q: What is the maximum number of nodes n in a binary tree of height h?



Height h
(e.g. 3)

BINARY TREES

Q: What is the maximum number of nodes n in a binary tree of height h?

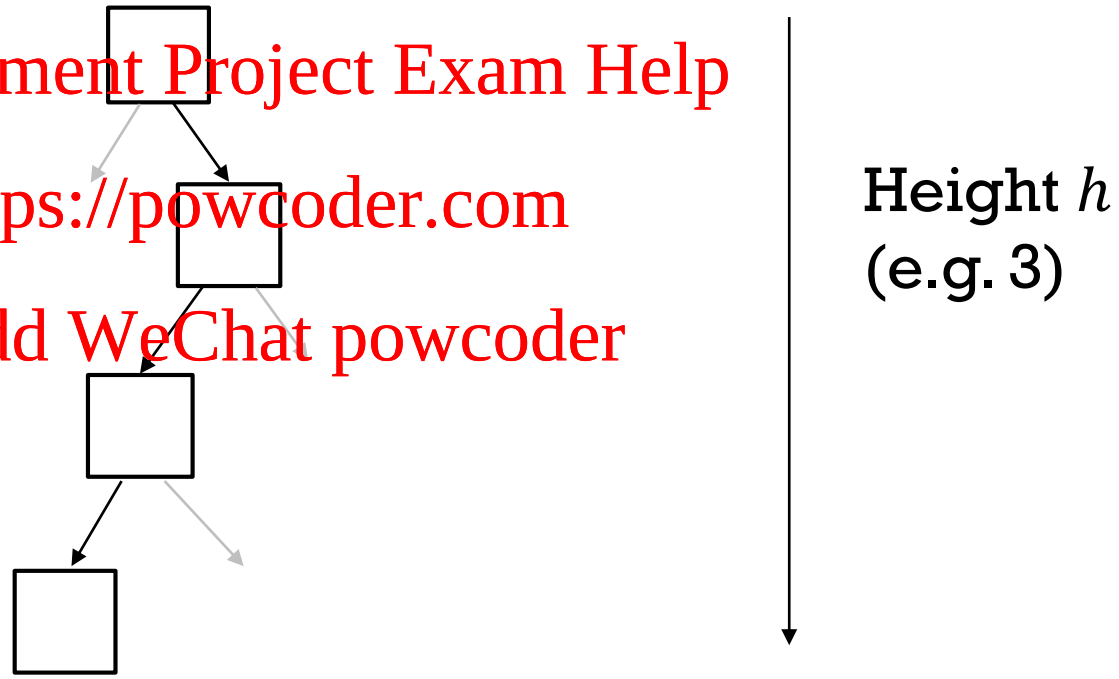


Height h
(e.g. 3)

A: $n = 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$

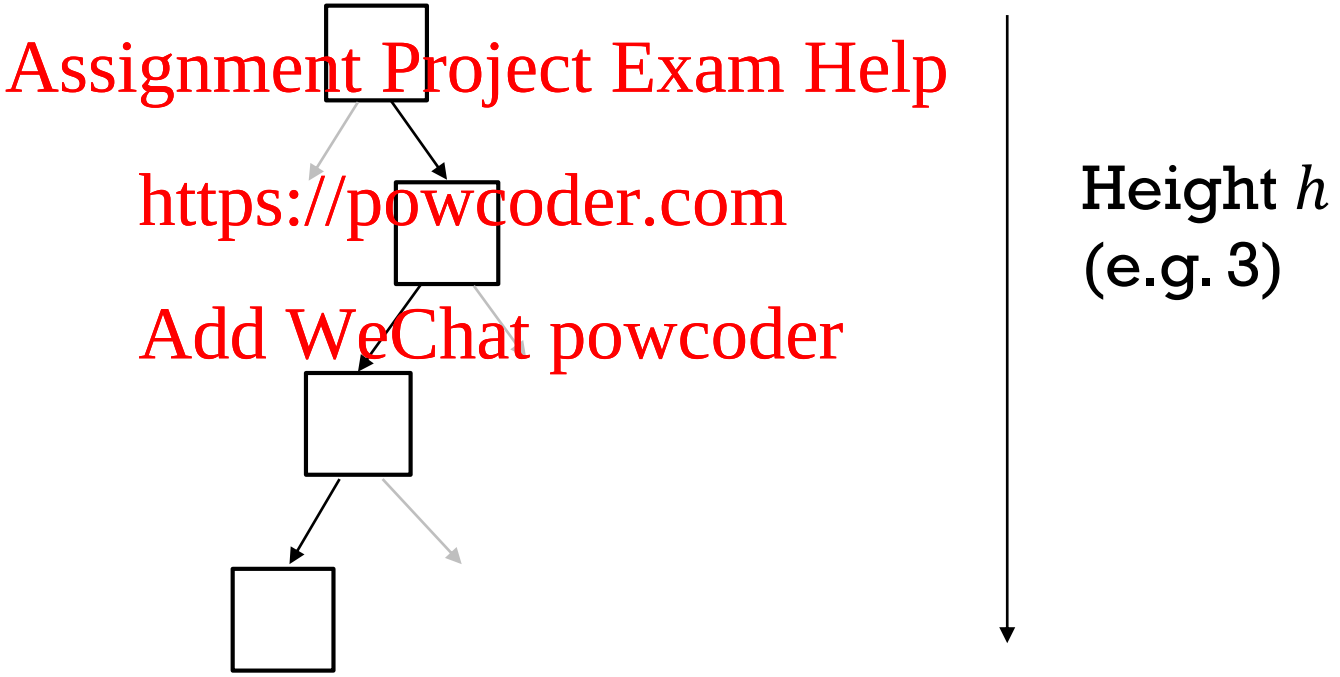
BINARY TREES

Q: What is the minimum number of nodes n in a binary tree of height h?



BINARY TREES

Q: What is the minimum number of nodes n in a binary tree of height h?



A: $n = h + 1$

BINARY TREES - IMPLEMENTATION

```
class  BTree<T>{
   BTNode<T>   root;
      :
   class BTNode<T>{
      T element;
      BTNode<T> leftchild;
      BTNode<T> rightchild;
            :
   }
}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

Rooted Tree
(last lecture)

Binary Tree


```
depthFirst(root){
   if (root is not empty){
      visit root
      for each child of root
         depthFirst( child )
   }
}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

Rooted Tree (last lecture)

```
preorder(root){
    if (root is not empty){
        visit root
        for each child of root
            preorder( child )
    }
}
```

Binary Tree

```
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

```
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```

```
postorderBT(root){

}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

```
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```

```
postorderBT(root){
    if (root is not empty){
        postorderBT(root.left)
        postorderBT(root.right)
        visit root
    }
}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

```
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```

```
postorderBT(root){
    if (root is not empty){
        postorderBT(root.left)
        postorderBT(root.right)
        visit root
    }
}
```



```
inorderBT(root){




}
```

BINARY TREE TRAVERSAL (DEPTH FIRST)

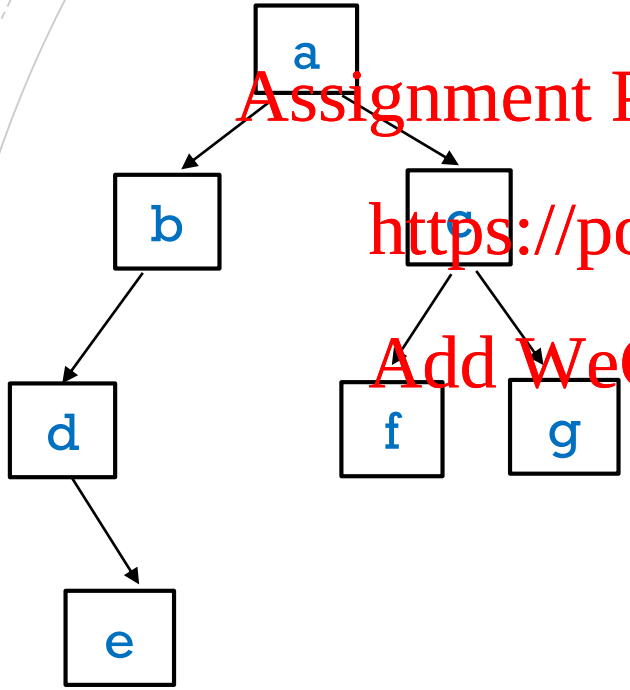
```
preorderBT(root){
    if (root is not empty){
        visit root
        preorderBT(root.left)
        preorderBT(root.right)
    }
}
```

```
postorderBT(root){
    if (root is not empty){
        postorderBT(root.left)
        postorderBT(root.right)
        visit root
    }
}
```



```
inorderBT(root){
    if (root is not empty){
        inorderBT(root.left)
        visit root
        inorderBT(root.right)
    }
}
```

EXAMPLE

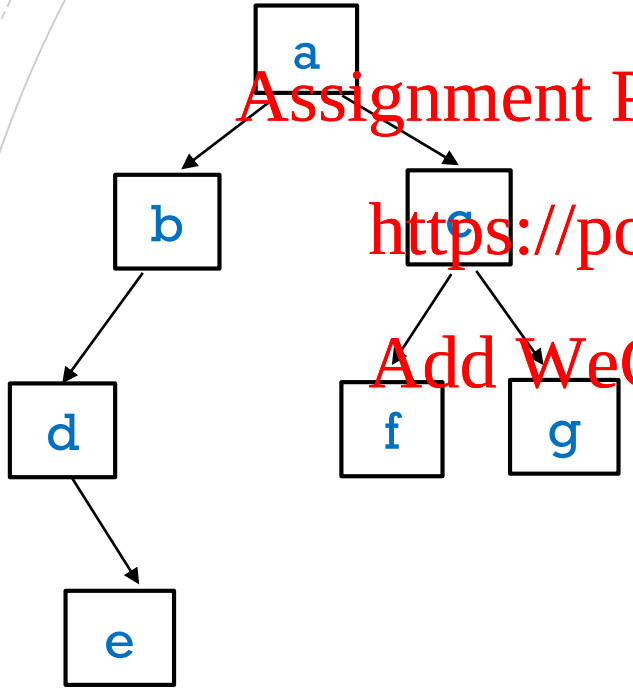


Pre order:

In order:

Post order:

EXAMPLE

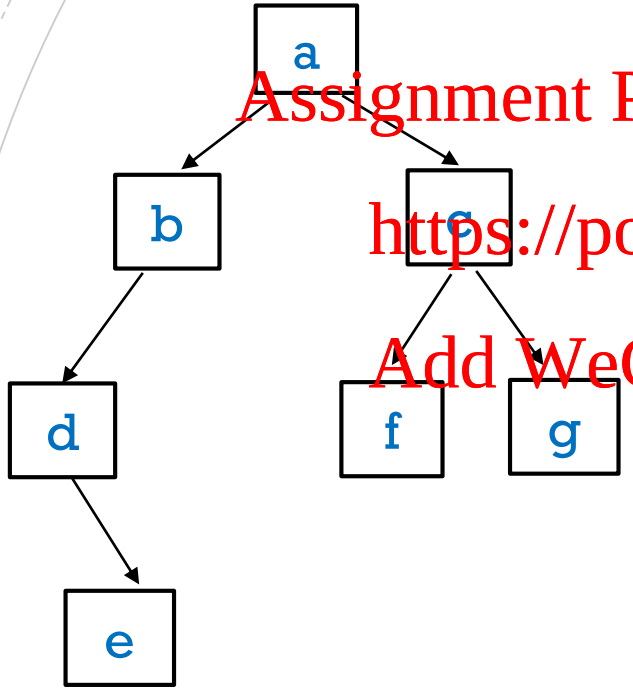


Pre order: a b d e c f g

In order:

Post order:

EXAMPLE

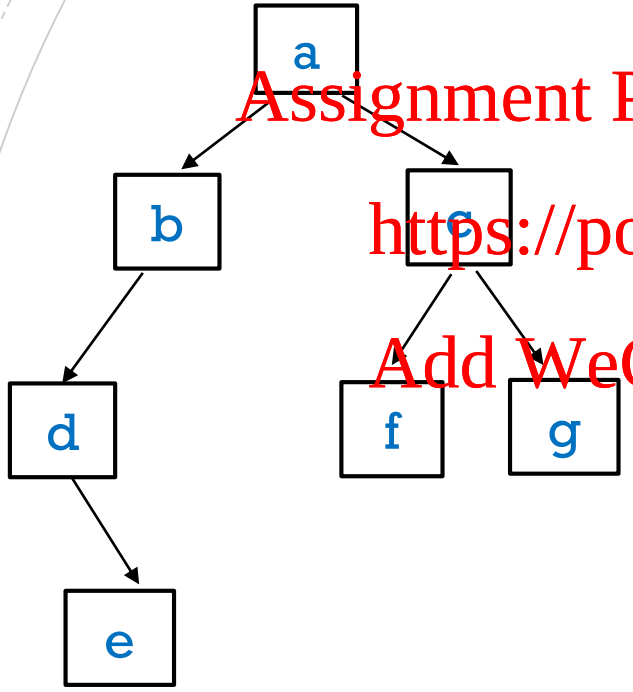


Pre order: a b d e c f g

In order: d e b a f c g

Post order:

EXAMPLE



Pre order: a b d e c f g

In order: d e b a f c g

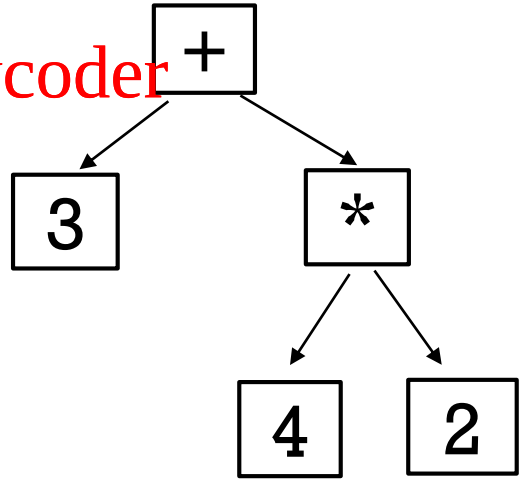
Post order: e d b f g c a

EXPRESSIONS TREES

EXPRESSION TREES

e.g. 3 + 4 * 2

3 + (4 * 2)

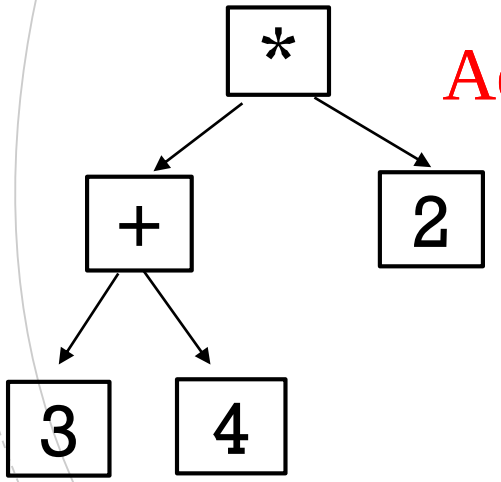


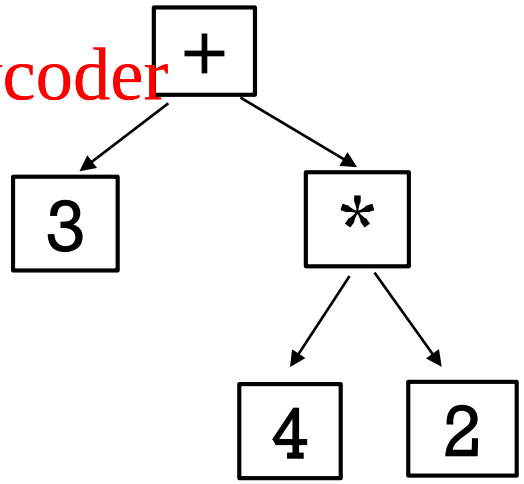
EXPRESSION TREES

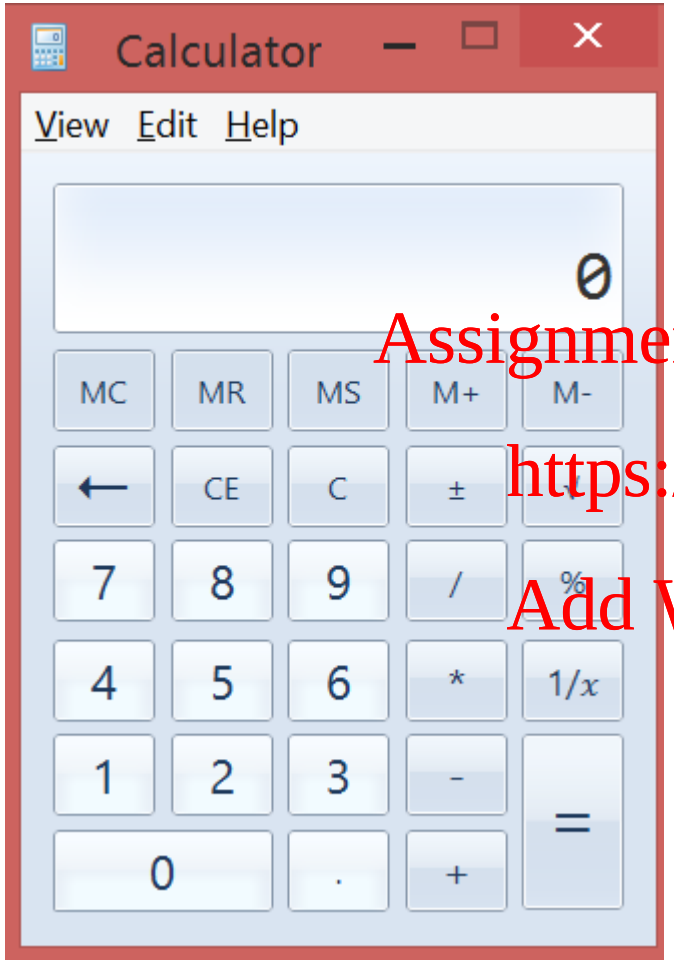
e.g. 3 + 4 * 2

(3 + 4) * 2

3 + (4 * 2)







My Windows calculator says

3 + 4 * 2 = 14.

Why? (3 + 4) * 2 = 14.

Whereas....

if I google "3+4*2", I get 11.

3 + (4*2) = 11.

EXPRESSIONS

We can make expressions using binary operators +, -, *, /, ^

e.g. a – b / c + d * e ^ f ^ g

^ is exponentiation: e ^ f ^ g = e ^ (f ^ g)

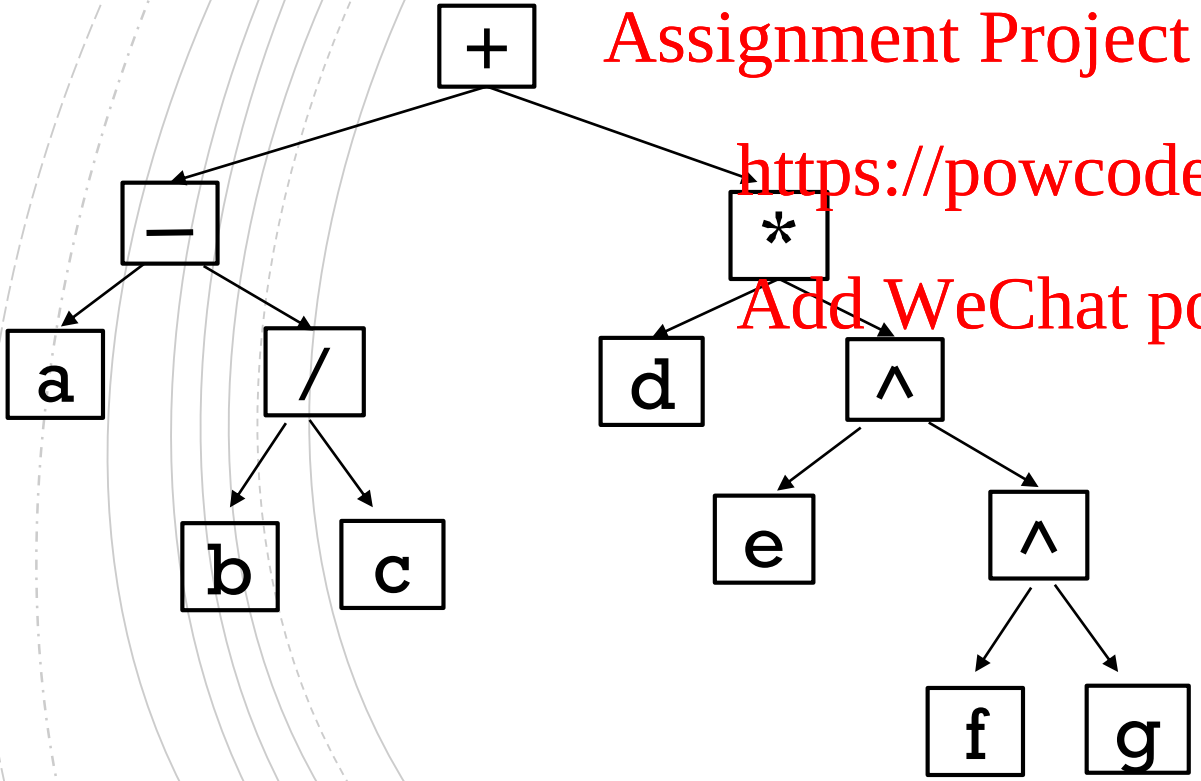
We don't consider unary operators e.g. 3 + -4 = 3 + (-4)

Operator precedence ordering makes brackets unnecessary.

(a – (b / c)) + (d * (e ^ (f ^ g)))

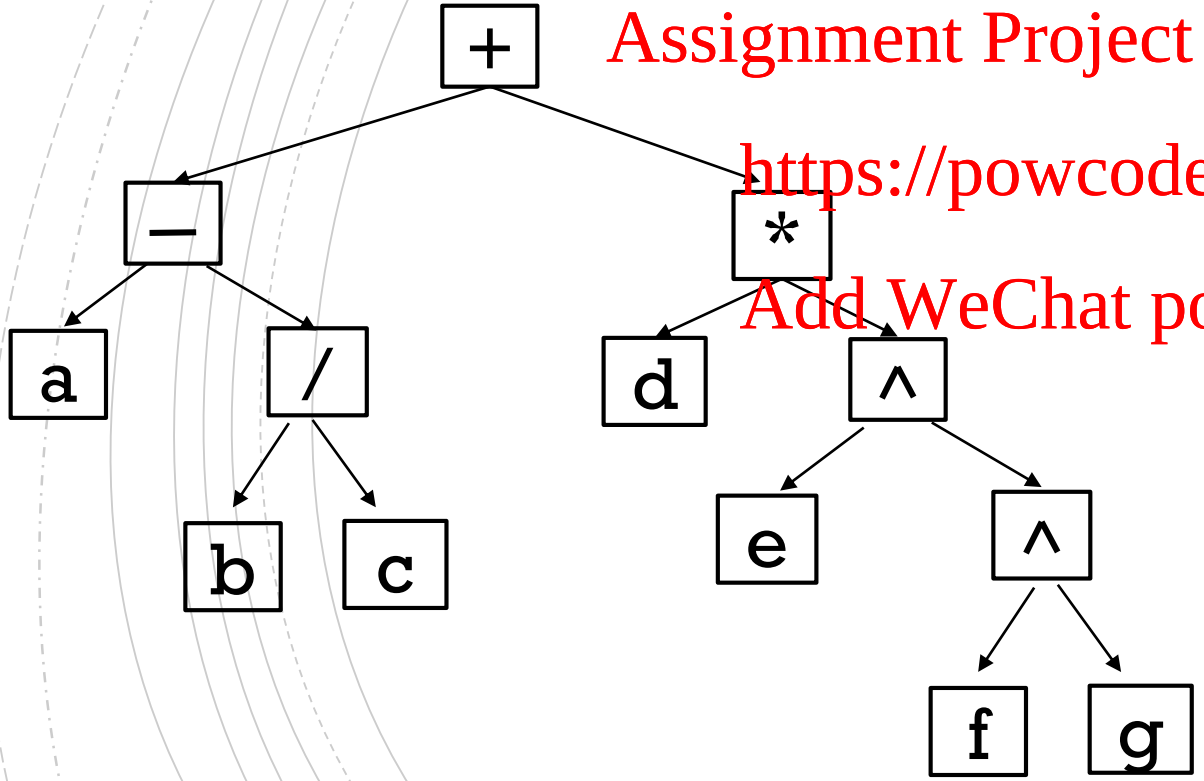
EXPRESSION TREE

a – b / c + d * e ^ f ^ g ≡ (a – (b / c)) + (d * (e ^ (f ^ g)))



Internal nodes are *operators*.
Leaves are *operands*.

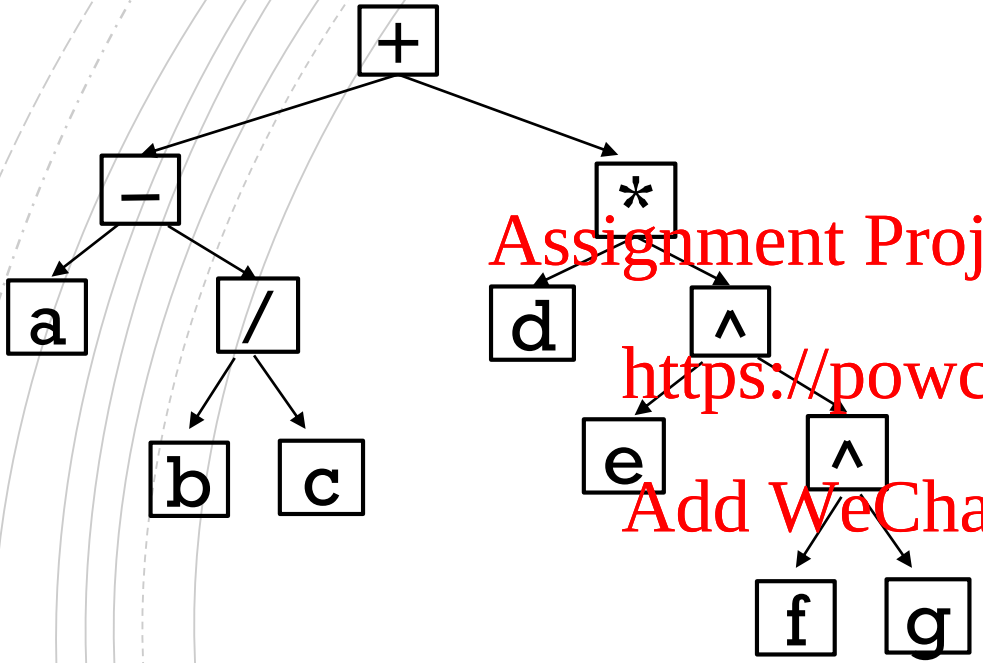
EXPRESSION TREE



An expression tree can be a way of *thinking about* the ordering of operations used when evaluating an expression.

But to be concrete, *let's assume we have a binary tree data structure*.

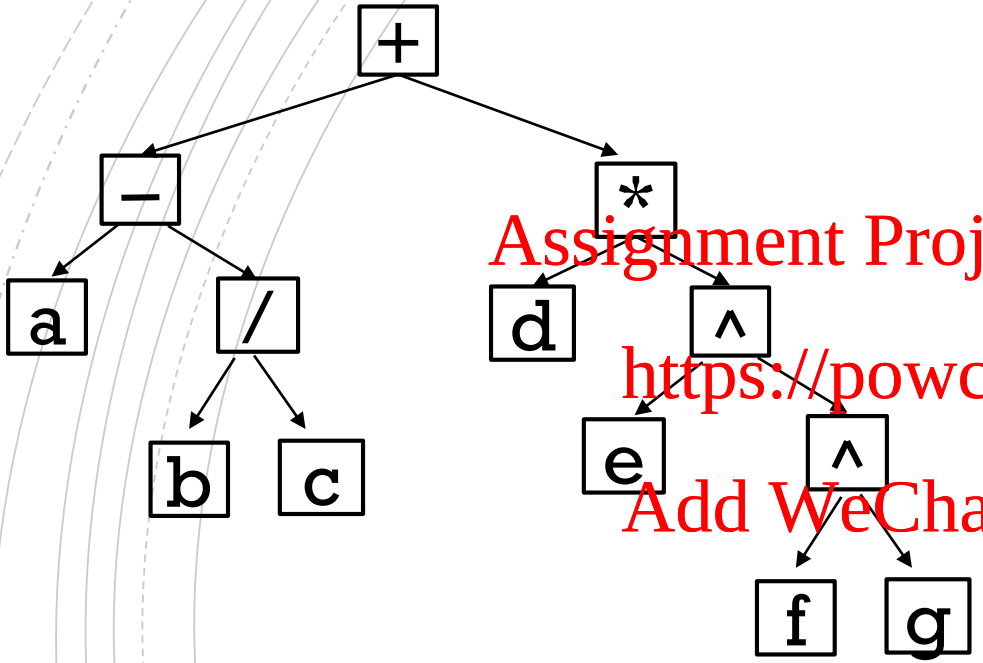
EXPRESSION TREE – TRAVERSAL



If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives:

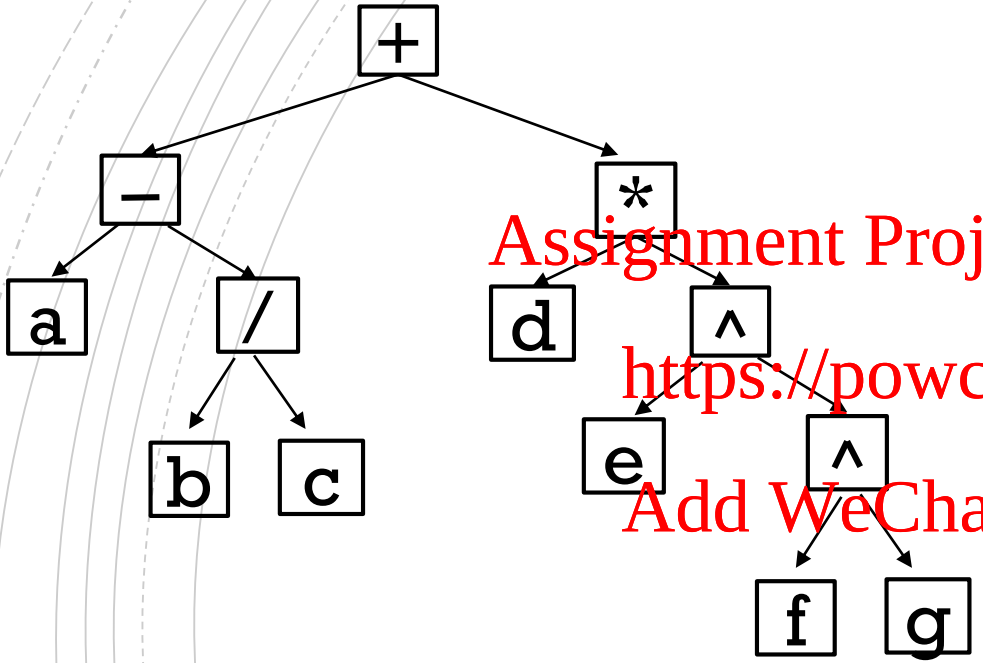
EXPRESSION TREE – TRAVERSAL



If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives: + – a / b c * d ^ e ^ f g

EXPRESSION TREE – TRAVERSAL

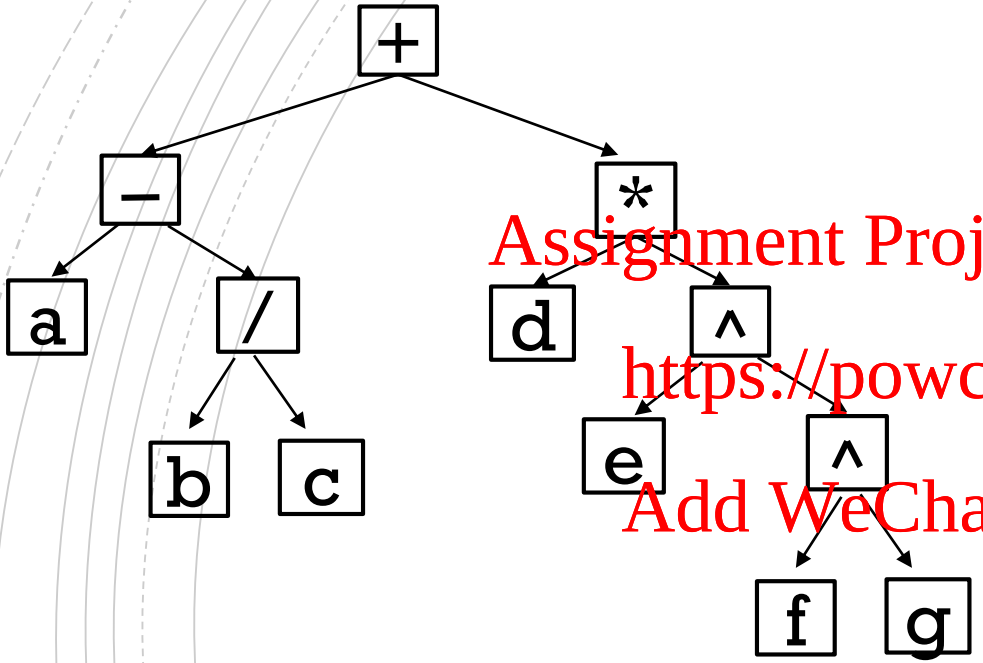


If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives: + – a / b c * d ^ e ^ f g

inorder traversal gives:

EXPRESSION TREE – TRAVERSAL

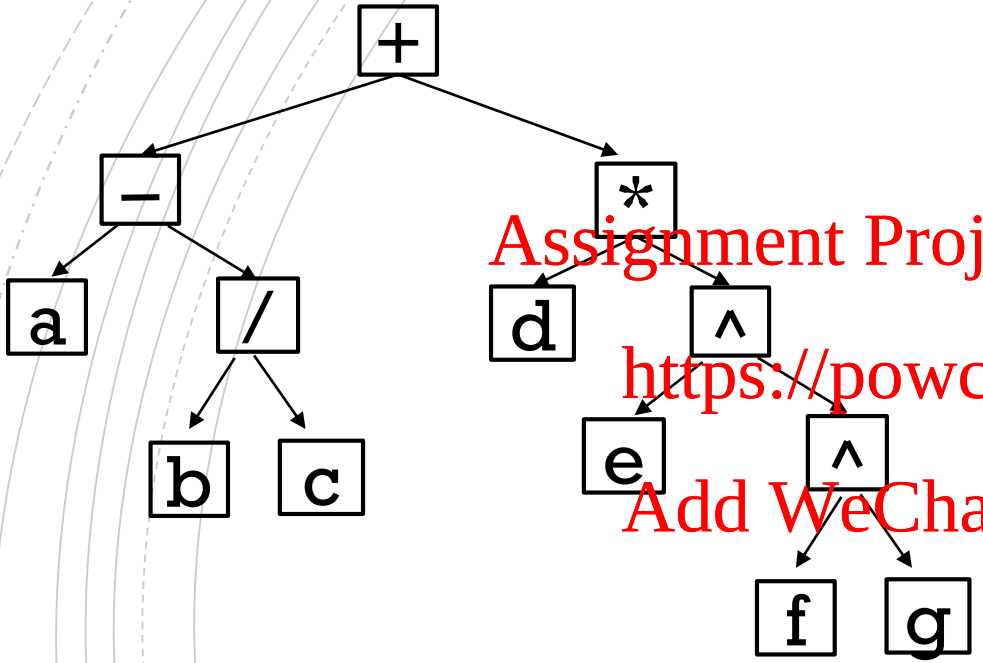


If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives: + – a / b c * d ^ e ^ f g

inorder traversal gives: a – b / c + d * e ^ f ^ g

EXPRESSION TREE – TRAVERSAL



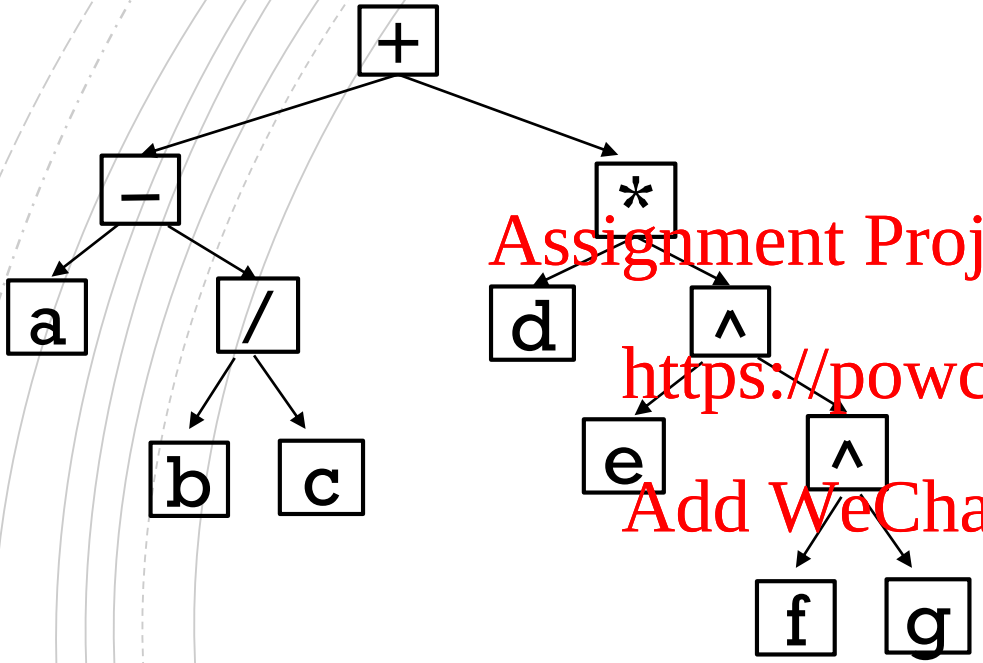
If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives: + – a / b c * d ^ e ^ f g

inorder traversal gives: a – b / c + d * e ^ f ^ g

postorder traversal gives:

EXPRESSION TREE – TRAVERSAL



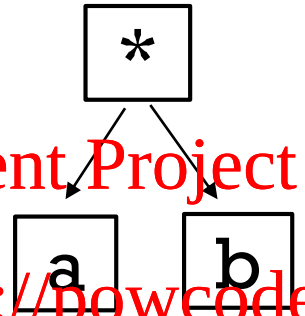
If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives: + – a / b c * d ^ e ^ f g

inorder traversal gives: a – b / c + d * e ^ f ^ g

postorder traversal gives: a b c / – d e f g ^ ^ * +

PREFIX, INFIX, POSTFIX EXPRESSIONS



prefix: * a b

infix: a * b

postfix: a b *

PREFIX, INFIX, POSTFIX EXPRESSIONS

```
baseExp  = variable | integer

op       =  +  |  -  |  *  |  /  |  ^

preExp   =  baseExp  |  op  preExp  preExp
```

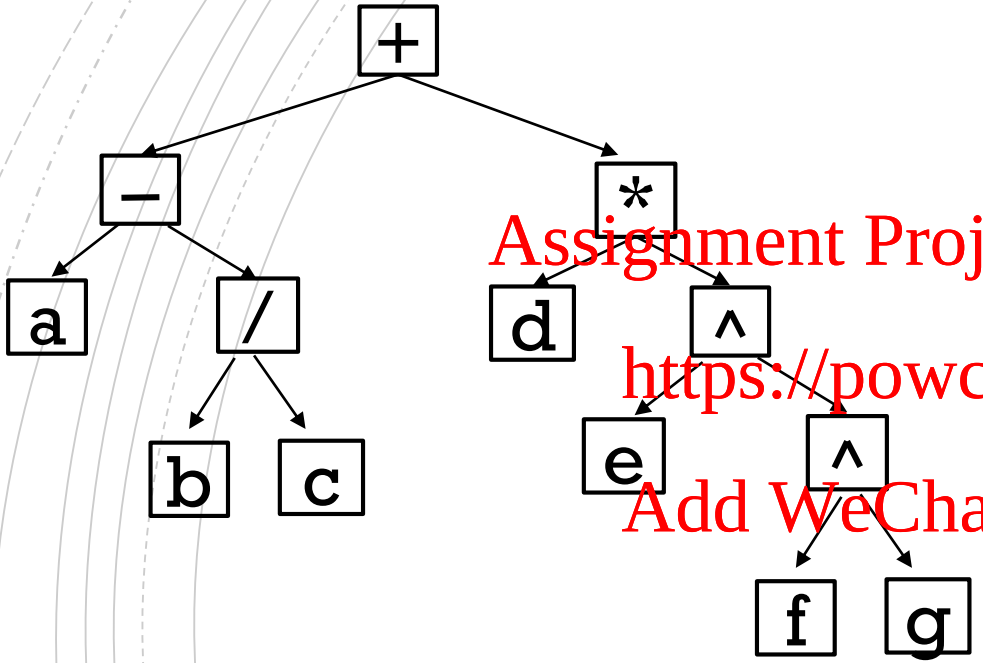

where | means ‘or’

PREFIX, INFIX, POSTFIX EXPRESSIONS


```
baseExp  = variable | integer

op       =  +  |  -  |  *  |  /  |  ^

preExp   =  baseExp  |  op  preExp  preExp

inExp    =  baseExp  |  inExp  op  inExp

postExp  =  baseExp  |  postExp  postExp  op
```

Use only one.

EXPRESSION TREE – TRAVERSAL



If we traverse an expression tree, and *print out* the node label, what is the expression printed out?

preorder traversal gives **prefix expression**: + – a / b c * d ^ e ^ f g

inorder traversal gives **infix expression**: a – b / c + d * e ^ f ^ g

postorder traversal gives **postfix expression**: a b c / – d e f g ^ ^ * +

TERMINOLOGY

- Prefix expressions called "Polish Notation"
  (after Polish logician Jan Lucasewicz 1920's)

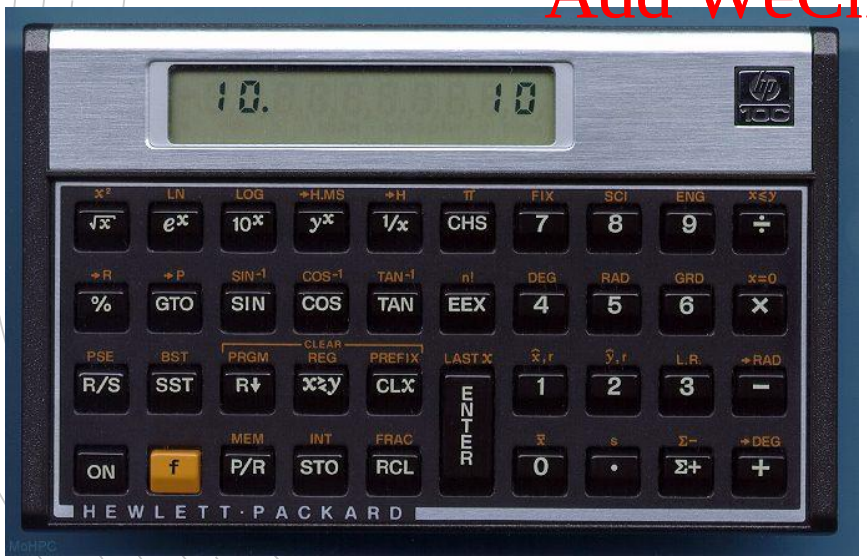
- Postfix expressions are called "Reverse Polish notation" (RPN)

Some calculators (esp. Hewlett Packard) require users to input expressions using RPN.

TERMINOLOGY

- **Prefix** expressions called "Polish Notation"
  (after Polish logician Jan Lucasewicz 1920's)

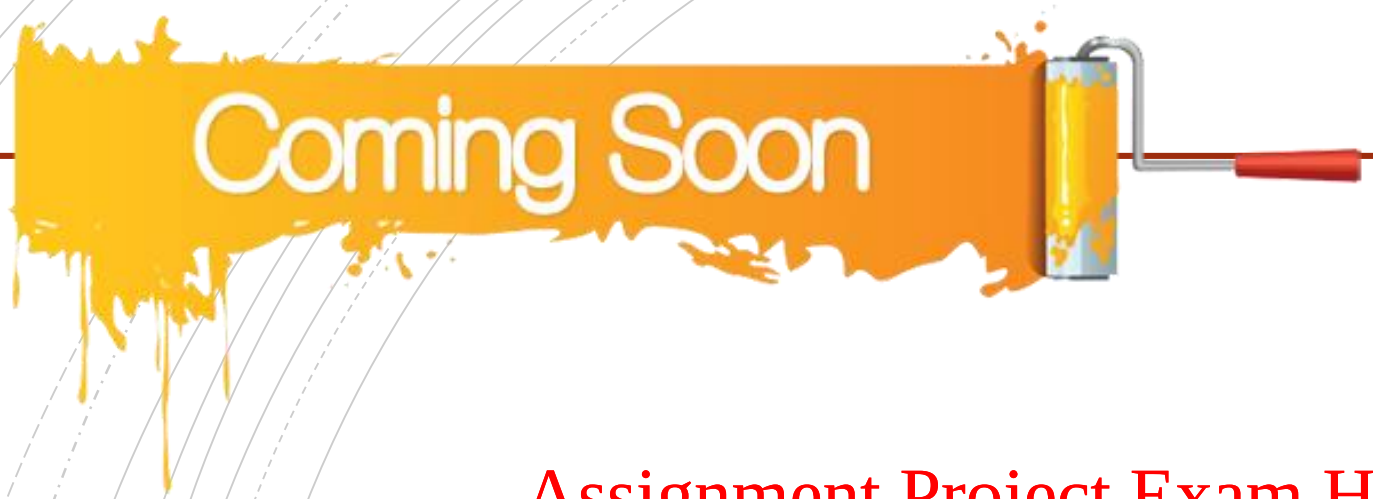
- **Postfix** expressions are called "Reverse Polish notation" (RPN)



Calculate 5 * 4 + 3 :

5 <enter>
4 <enter>
* <enter>
3 <enter>
+ <enter>

No "=" symbol on keyboard.



In the next videos:

- Binary Search Trees
- Heaps