Slide 27:

Pre-Order Traversal

```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder (w)
```

Firstly, we visit the root node and then call preOrder() for each child of the root.

When the left child is called, the node is visited and then preOrder() is call for its child nodes and so on until we arrive at a leaf node. So all of the left children are visited first on the left side of the tree. On the way back up the tree we visit the right children. When we get back to the root we complete the same process on the right hand side of the tree.

Slide 28:

Post-Order Traveral

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

This time we are visiting the root after we visit the subtrees. We traverse all the way down the left again but this time we don't visit a node until we get to the furthest point on the left (leaf node). Then as we travel back up the tree we visit all of the child nodes before the parent nodes.

Slide 29:

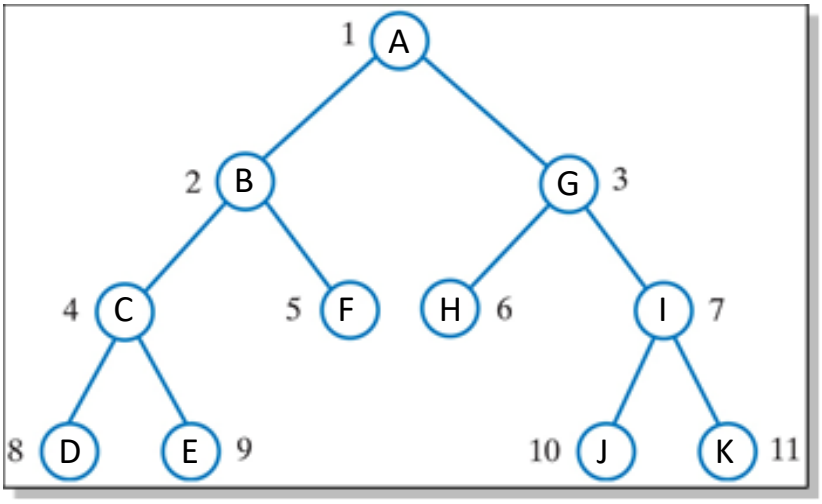
In-Order Traversal

```
Algorithm inOrder(v)
    if hasLeftChild (v)
        inOrder (getLeftChild(v))
    visit(v)
    if hasRightChild(v)
        inOrder (getRightChild (v))
```

This time we are visiting the root node between the visits to the subtrees. Again, we work our way down the left of the tree as we are recursively calling inOrder every time a left child exists. When

there is no further left child, we visit the node we are at (as the condition of the first if statement is not met). We then visit the parent node before visiting the right child.

Slide 31:



This is more complicated than depth-first traversal. The general procedure is to use a queue:

Begin with the root node on the queue

Loop:

Dequeue the front node

Before visiting it, enqueue its children (if any)

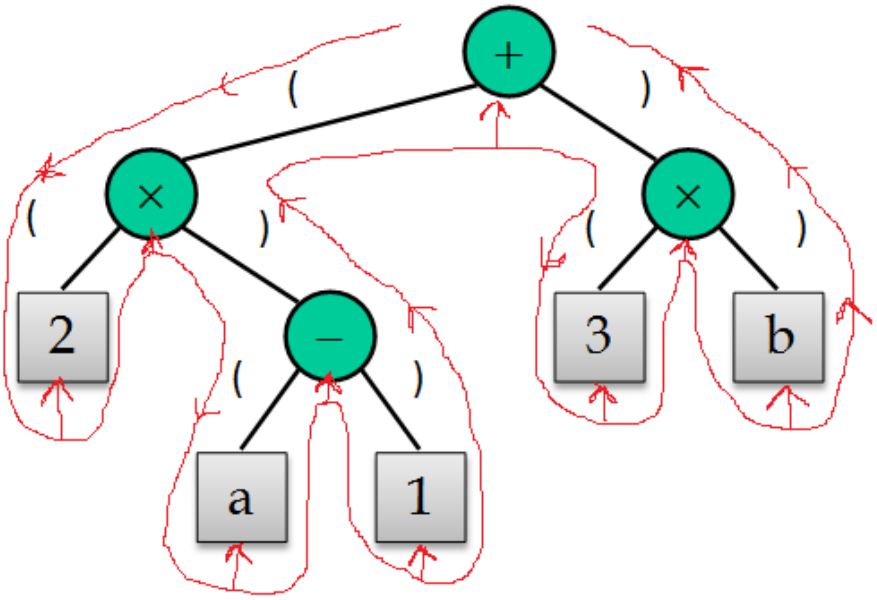
While the queue is not empty

Notes:

The procedure described here is **NOT** recursive, unlike the other traversal procedures we have studied. You only need one queue, not one per level.

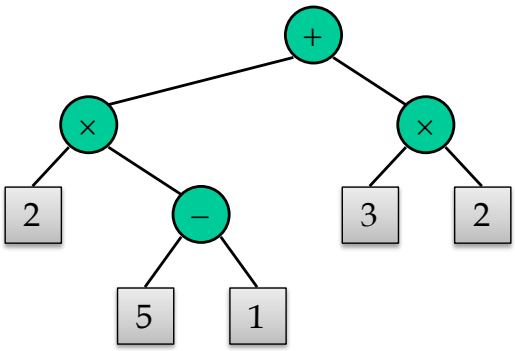
Queue	Dequeue	Enqueue?	Visit Node	
[A]	A←	[B, G]	Visit A	
[B, G]	B←	[G, C, F]	Visit B	
[G, C, F]	G←	[C, F, H, I]	Visit G	
[C, F, H, I]	C←	[F, H, I, D, E]	Visit C	
[F, H, I, D, E]	F←	[H, I, D, E]	Visit F	
[H, I, D, E]	H←	[I, D, E]	Visit H	
[I, D, E]	I←	[D, E, J, K]	Visit I	
[D, E, J, K]	D←	[E, J, K]	Visit D	
[E, J, K] `	E←	[J, K]	Visit E	
[J, K]	J←	[K]	Visit J	(and finally visit K)

Slide 34:



((2 X (a – 1)) + (3 X b))

Slide 35:




```
Algorithm evalExpr(v)
    if isLeaf(v)
        return v.getData()
    else
        x ← evalExpr(getLeftChild(v))
        y ← evalExpr(getRightChild(v))
        ◊ ← operator stored at v
        return x ◊ y
```

+ leaf? No

X ← x leaf? No

X ← 2 leaf? Yes (Return 2)

Y ← - leaf? No

X← 5 leaf? Yes

Y←1 leaf? Yes

♦ ← -

Return 4 (5-1)

♦ X

Return 8 (2x4)

Y ← x leaf? No

X ← 3 leaf? Yes (Return 3)

Y ← 2 leaf? Yes (Return 2)

♦ X

Return 6 (3x2)

● +

Return 14 (8+6)