ECS32B

Introduction to Data Structures

Tree Traversal

Balanced Binary Trees

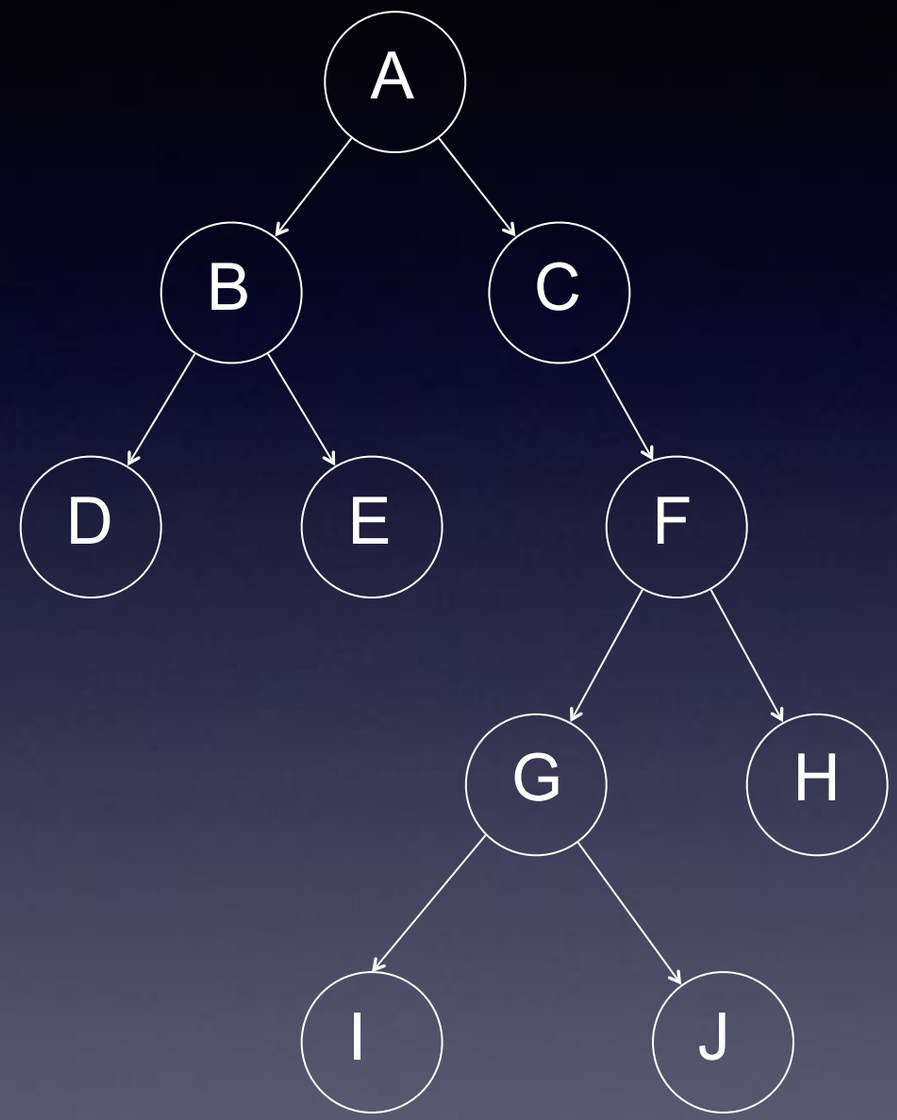
Lecture 23

Announcements

- Homework 5 tonight at 11:59pm
- Extra HW5 lab hours today in 78B with Anshuman from 4:30 to 6:30.
- Makeup assignment will be posted momentarily.
- Homework 6 will be posted next week.
- Sample final will be posted the last week of class.

Tree Traversal

We use **tree traversal** when we want to do something at every node of a tree.



We'll talk about the three main kinds of tree traversal: **preorder**, **postorder**, and **inorder**. The pre-, post-, and in- indicate when the root node is **processed** (also known as **visited**) in relation to its subtrees.

Tree Traversal

preorder

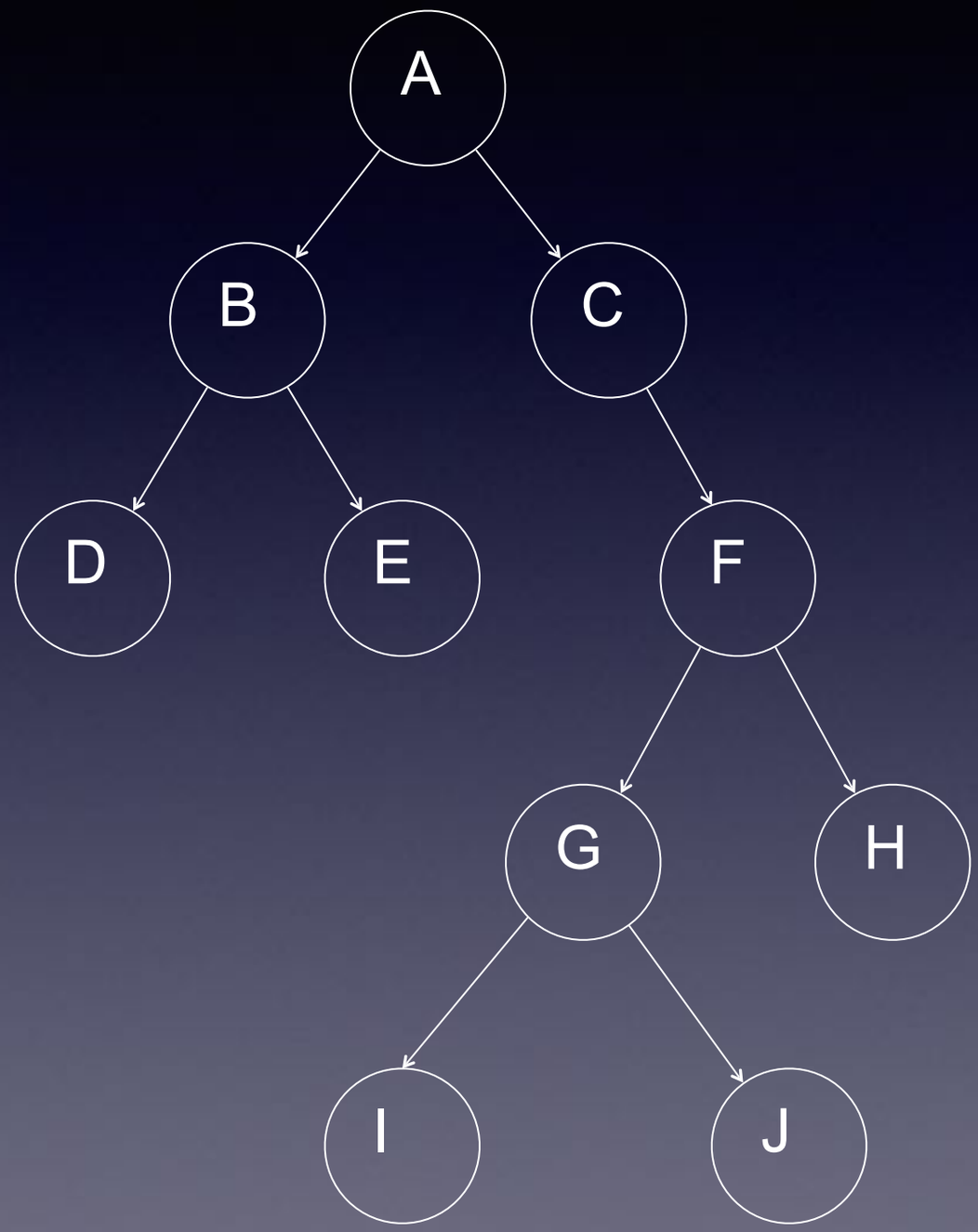
we **process** the root node first, then recursively do a preorder traversal of the **left subtree**, followed by a recursive preorder traversal of the **right subtree**.

inorder

recursively do an inorder traversal on the **left subtree**, **process** the root node, and finally do a recursive inorder traversal of the **right subtree**.

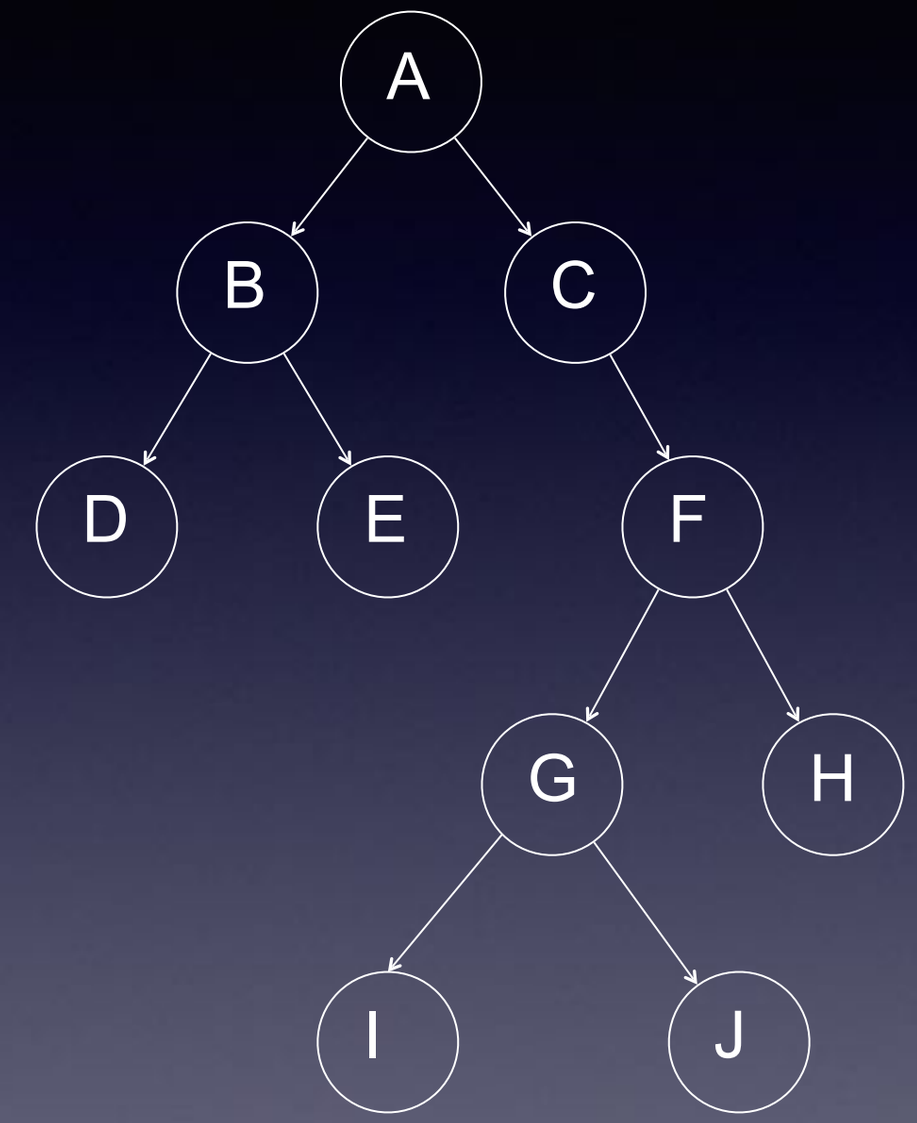
postorder

recursively do a postorder traversal of the **left subtree** and the **right subtree** followed by a **process** to the root node.



Preorder traversal

```
if the tree is empty
  return
else
  visit (process) the root
  apply preorder to the
    left subtree
  apply preorder to the
    right subtree
```

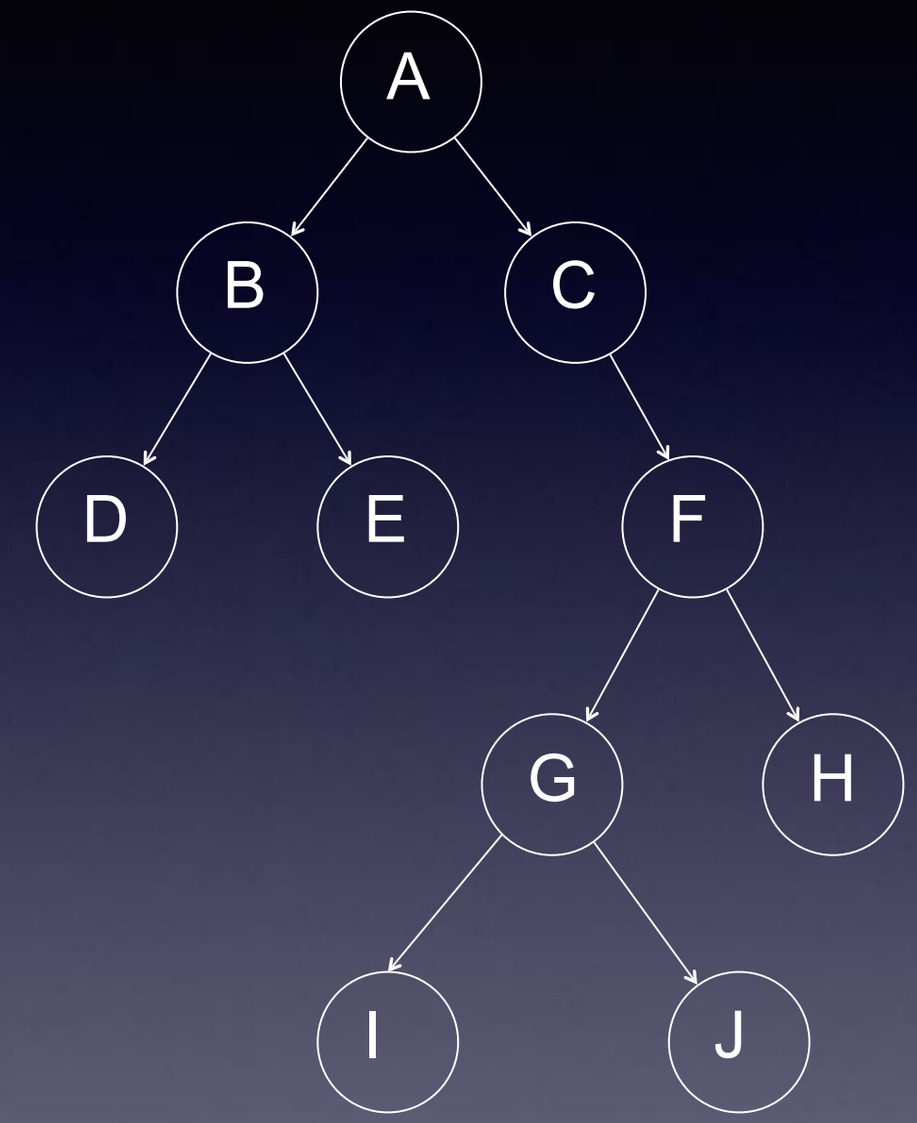


Let's say that visit or process in this case means "print the value". In what order will the values be printed?

A B D E C F G I J H

Postorder traversal

```
if the tree is empty
  return
else
  apply postorder to the
    left subtree
  apply postorder to the
    right subtree
  visit (process) the root
```

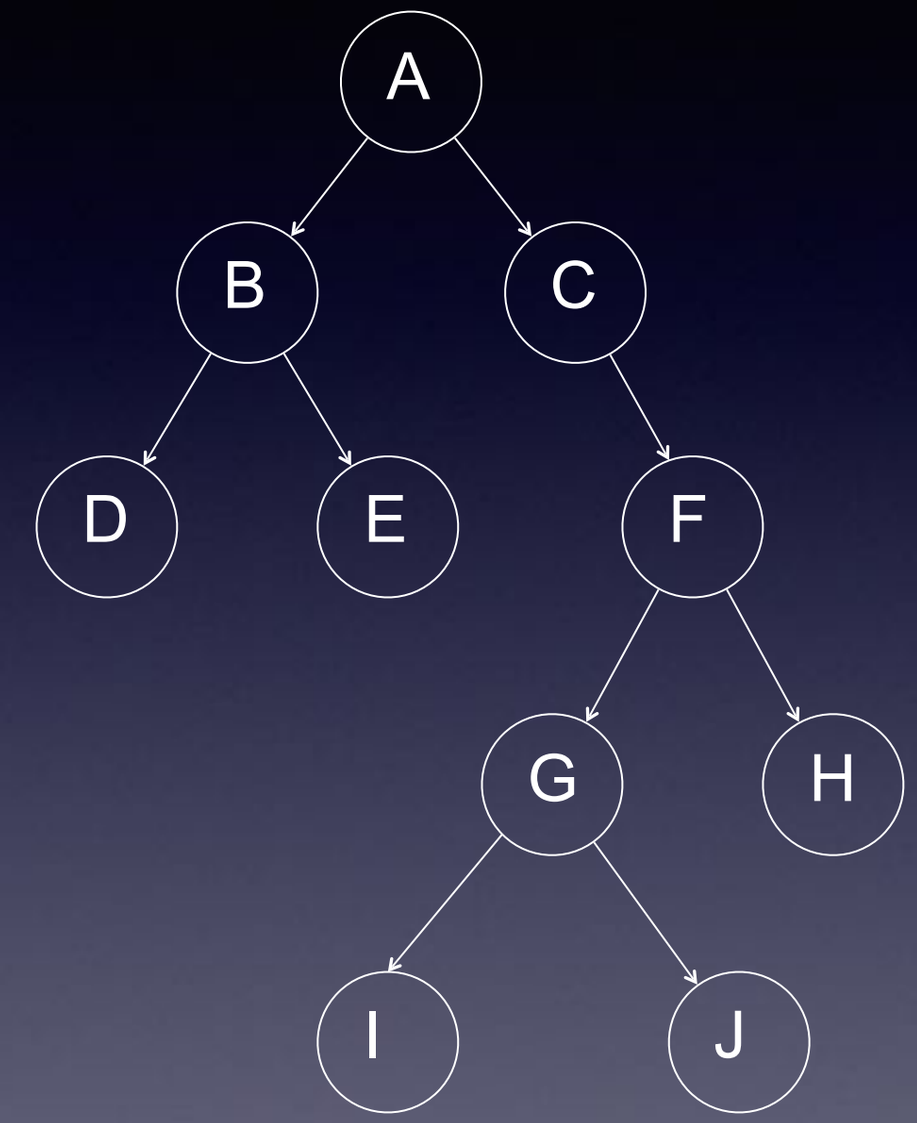


Let's say that visit or process in this case means "print the value". In what order will the values be printed?

D E B I J G H F C A

Inorder traversal

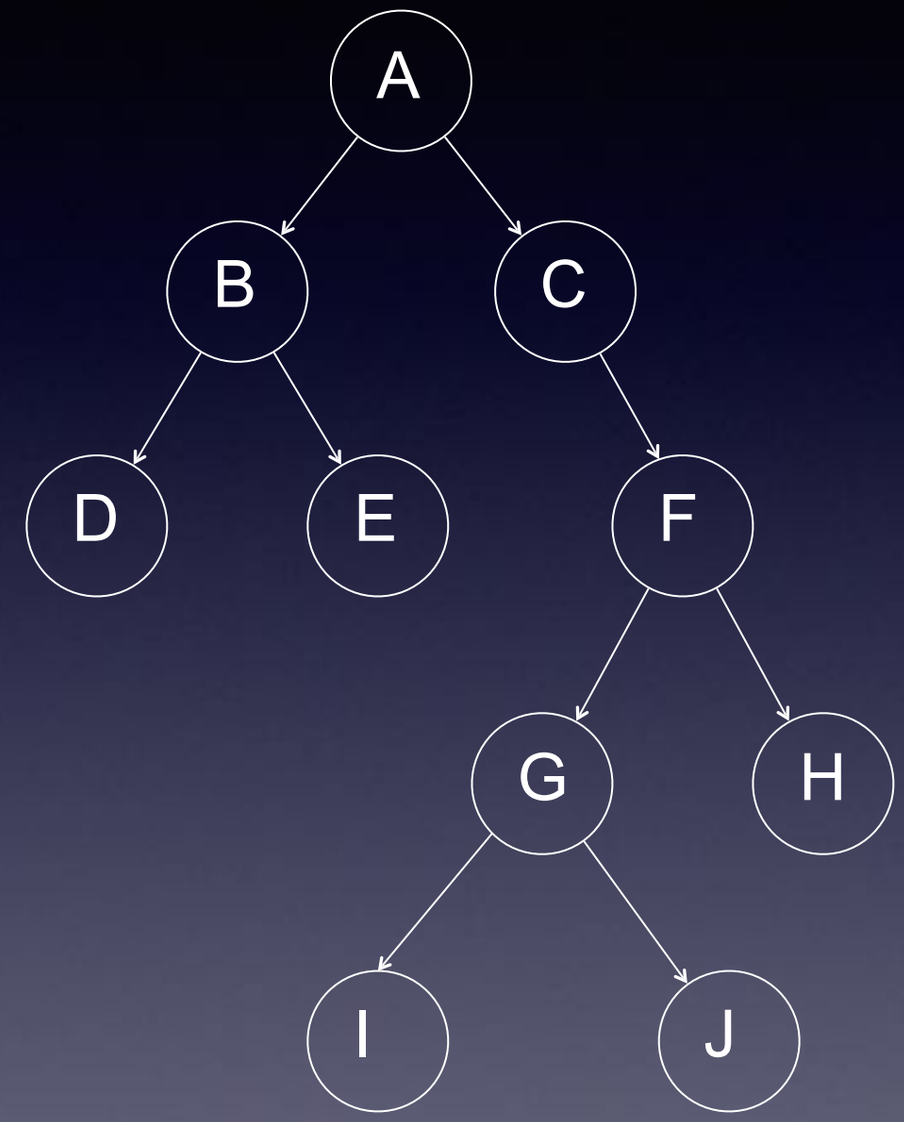
```
if the tree is empty
  return
else
  apply inorder to the
    left subtree
  visit (process) the root
  apply inorder to the
    right subtree
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?

D B E A C I G J F H

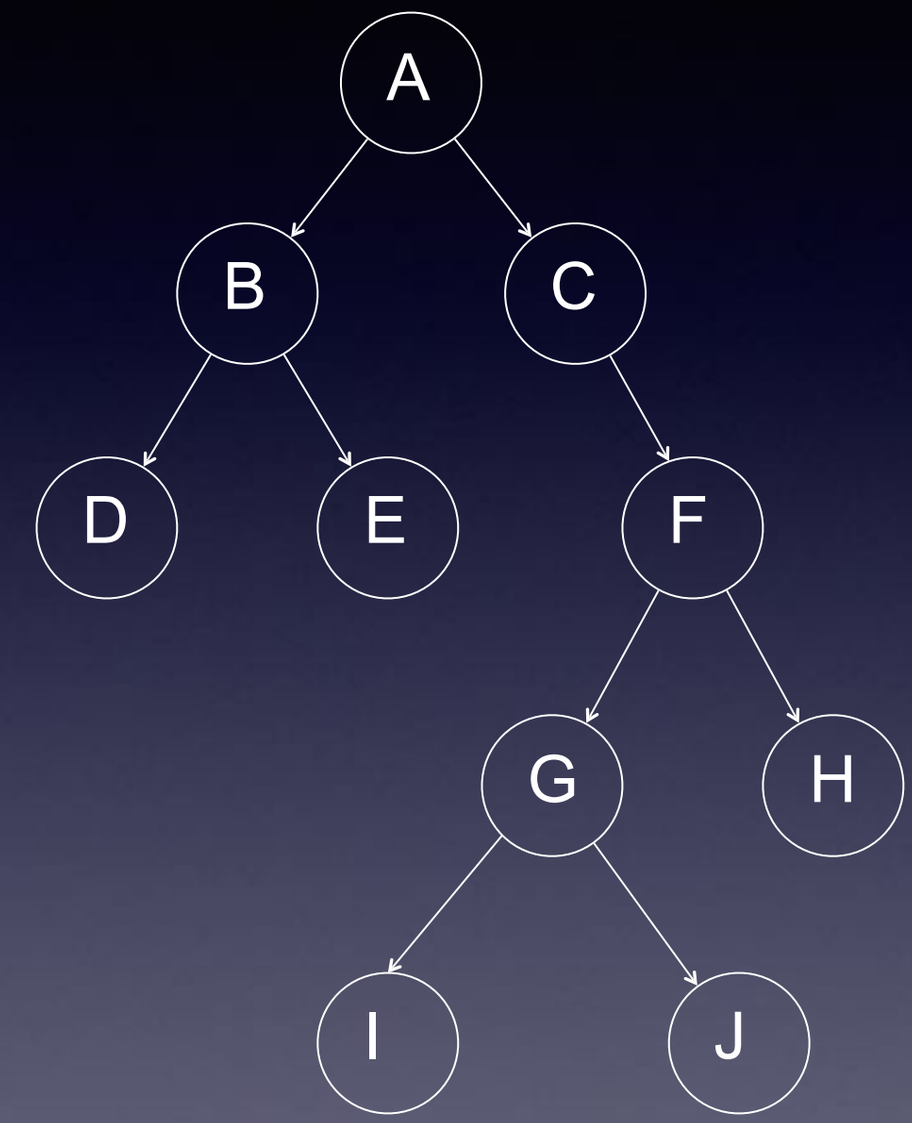
A "bonus" traversal



How could I get the traversal (printing) to happen in this order?
A B C D E F G H I J

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

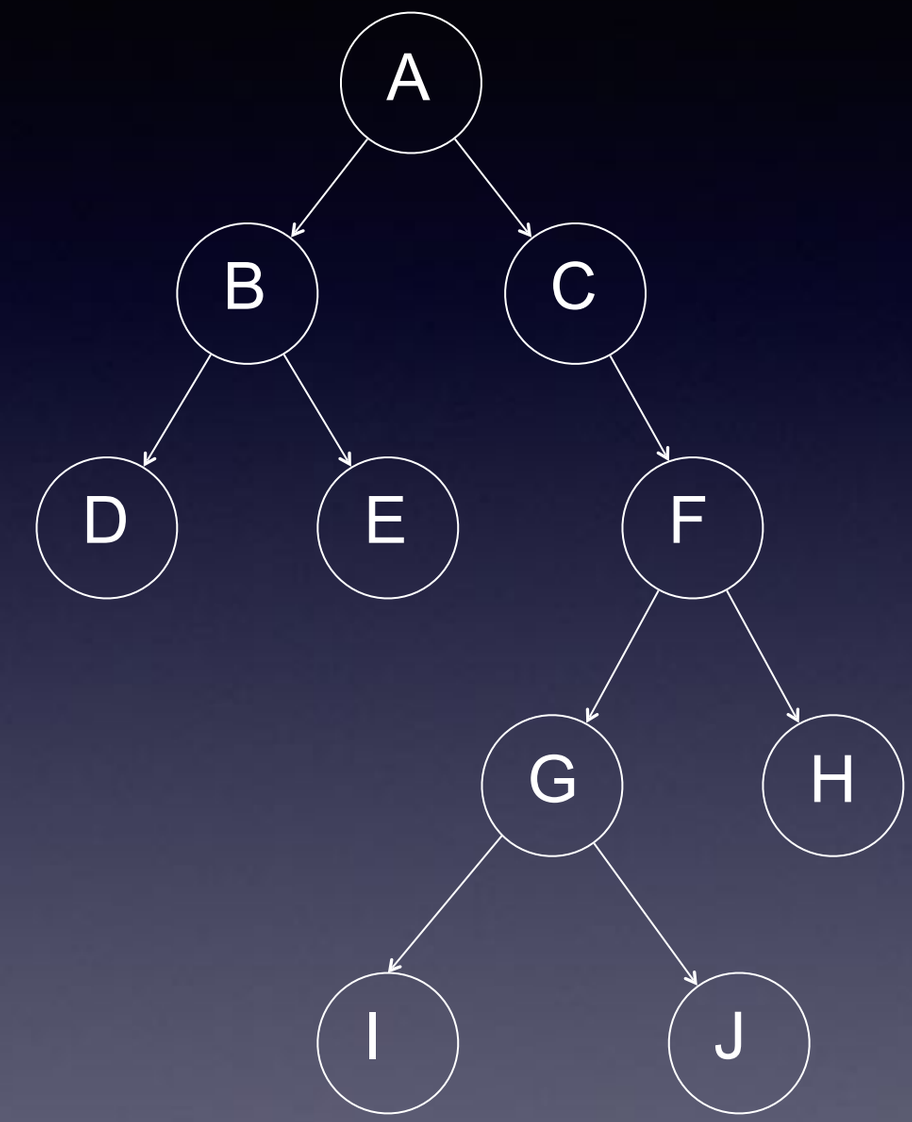


How could I get the traversal (printing) to happen in this order?
A B C D E F G H I J
Level order traversal.

Level order traversal

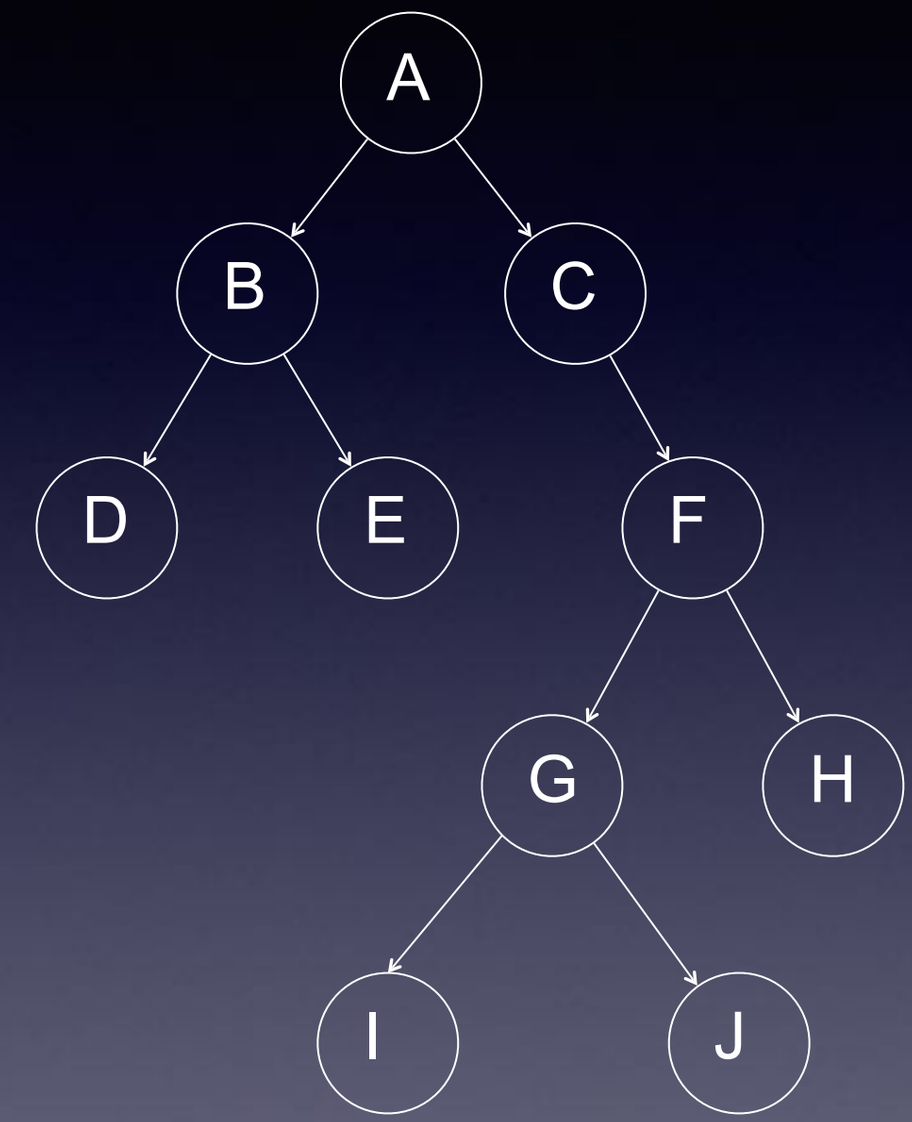
```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

queue:
['A']



Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

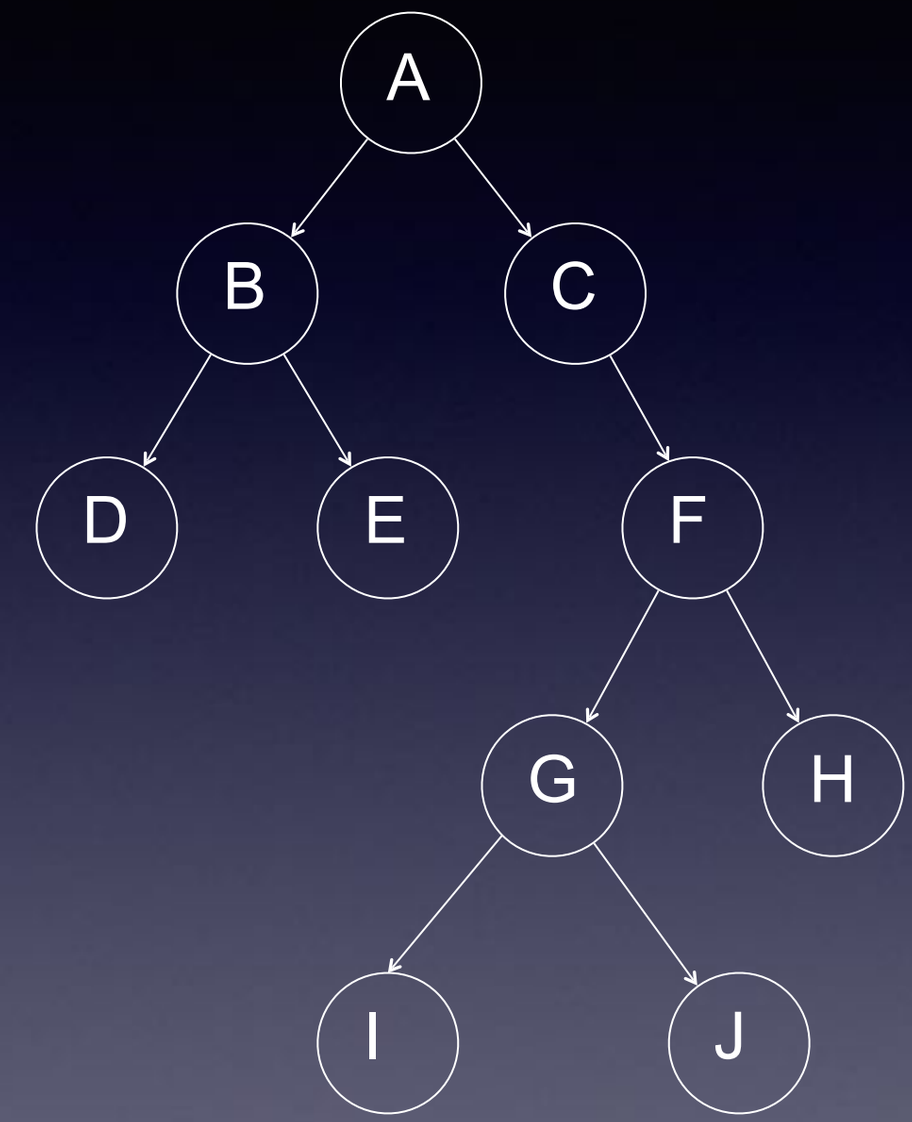


queue:
['C','B']

A

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

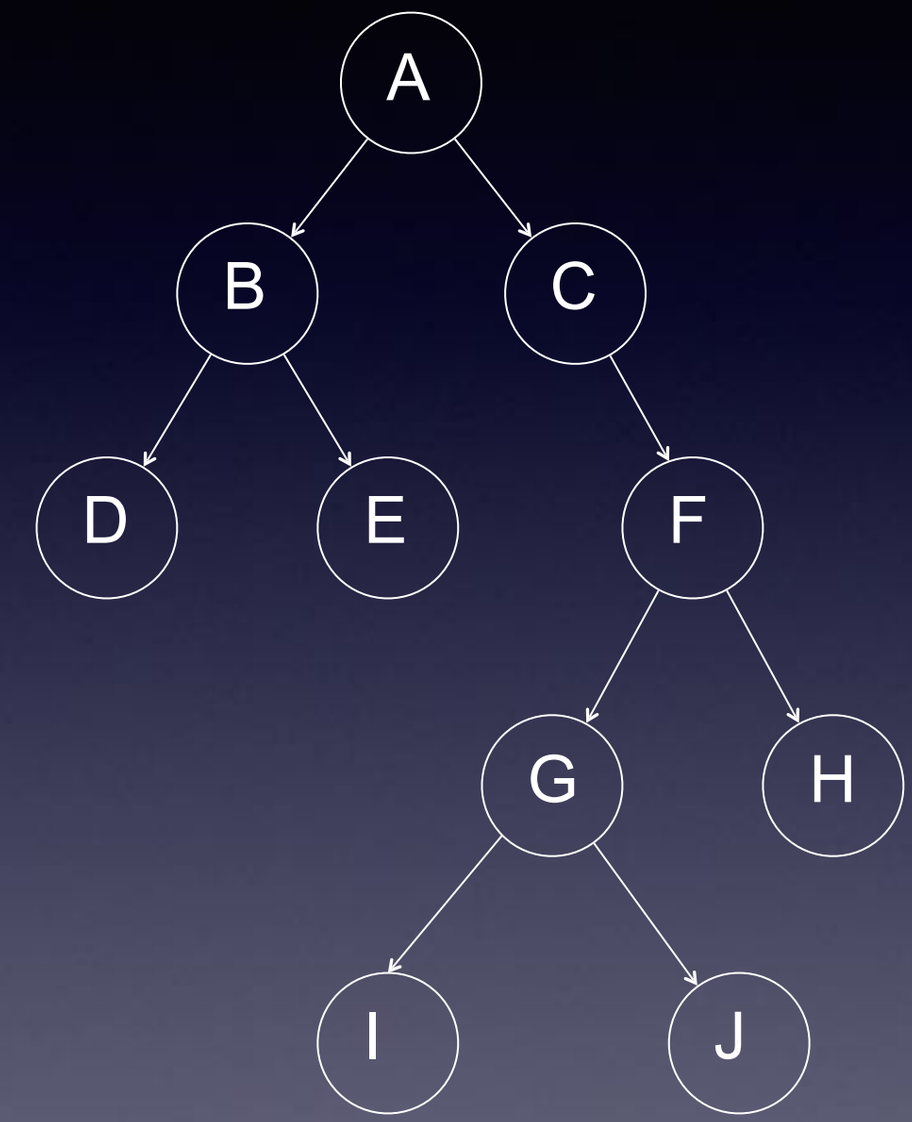


queue:
['E','D','C']

A B

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

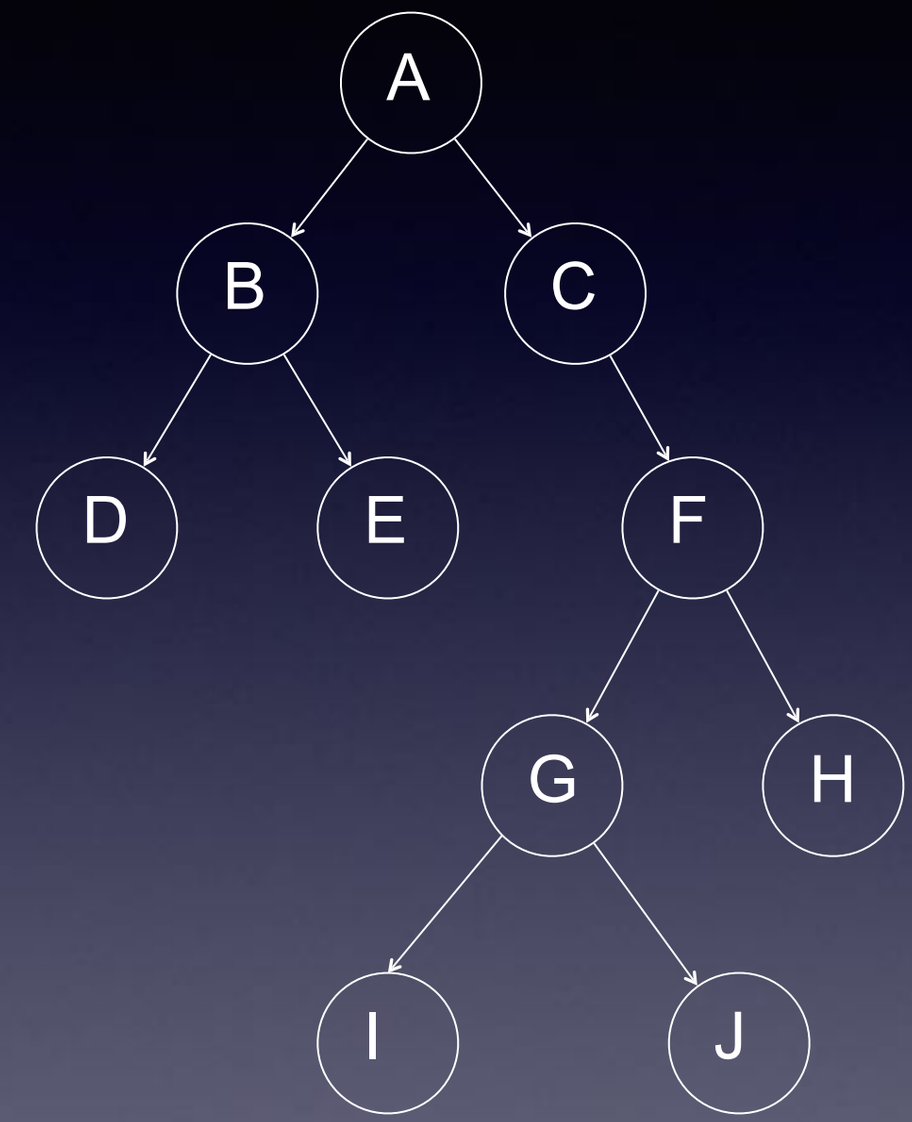


queue:
['F','E','D']

A B C

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

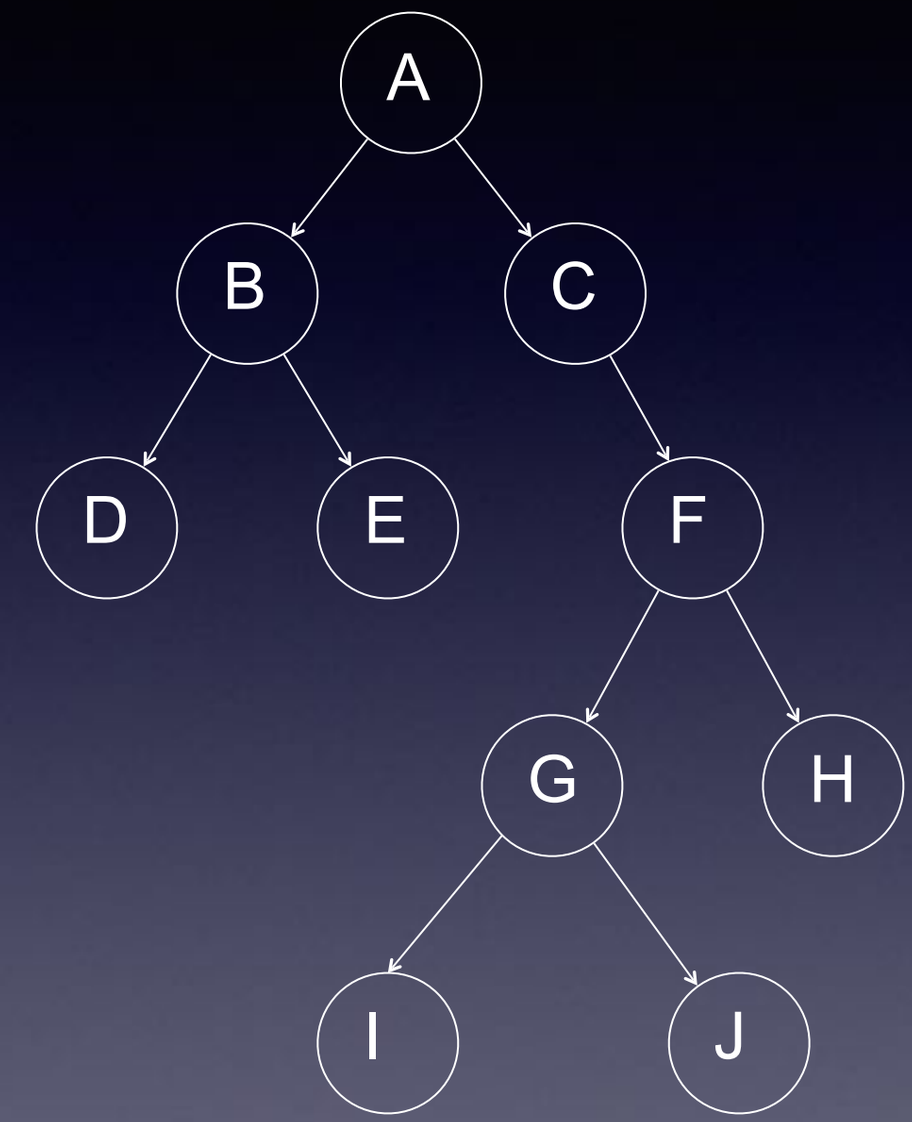


queue:
['H','G']

A B C D E F

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```

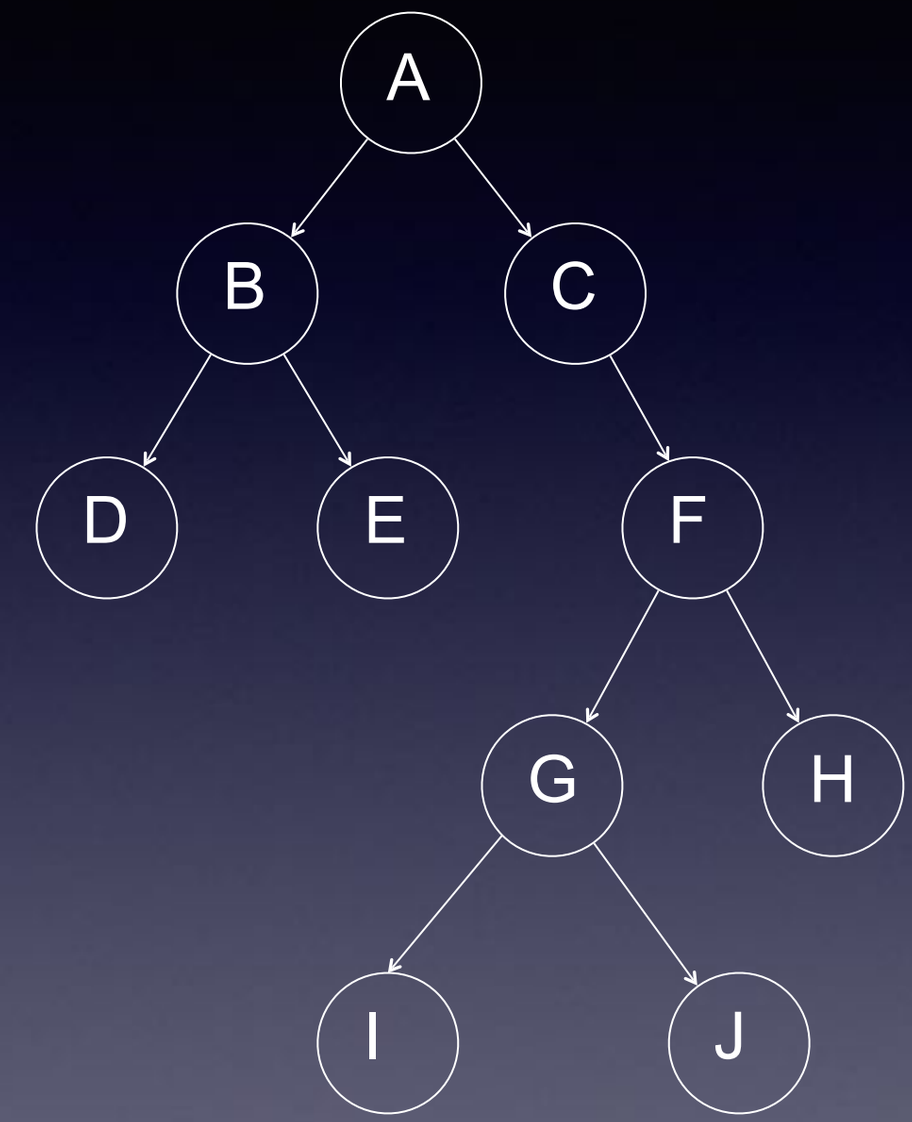


queue:
['J','I','H']

A B C D E F G

Level order traversal

```
add root to queue
while queue not empty:
  take node from queue
  process the node
  if left node ptr isn't null
    then put left node on
    queue
  if right node ptr isn't null
    then put right node on
    queue
```



queue:
[]

A B C D E F G H I J

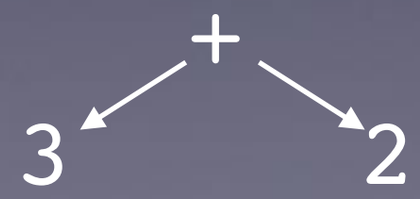
Binary expression trees

Here's one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build an expression tree.

(3 + 2) yields:

The subtrees of this operator are the operands

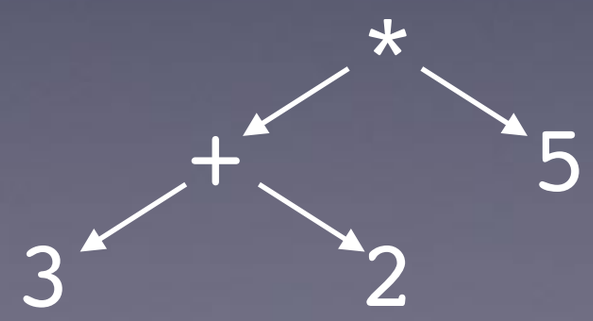


Binary expression trees

Here's one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

* 5 adds this:



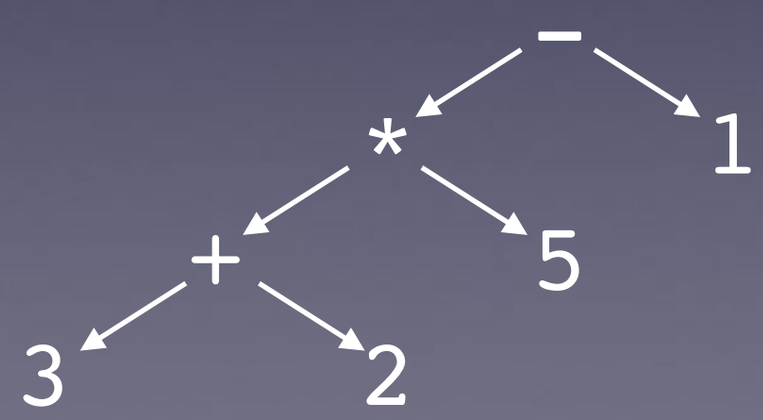
Binary expression trees

Here's one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

If we print this tree using **postorder** traversal, we get:

```
3 2 + 5 * 1 -
```

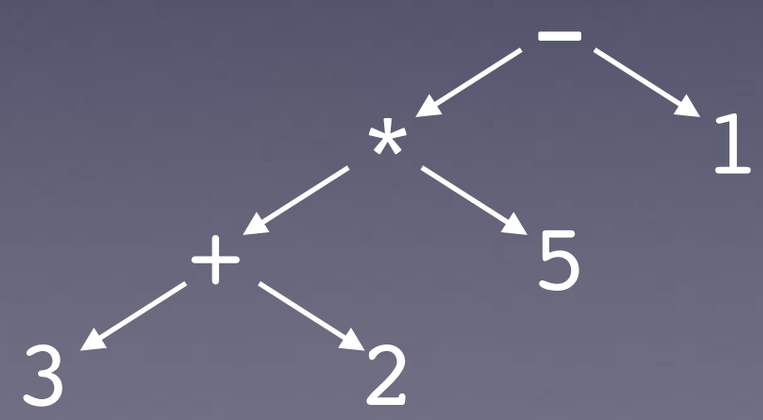


Binary expression trees

Why that's just Reverse Polish Notation! Try it out with your RPN calculator. (What? You don't have one?)

It's much easier to write an algorithm to evaluate an expression in RPN than in traditional infix notation. No parentheses are needed, calculations can be done immediately, and you can use a stack to do the calculations (it's just conceptually simpler).

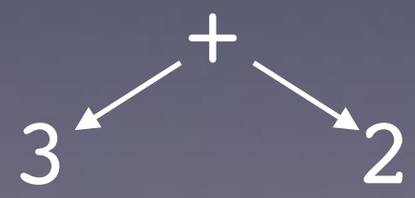
```
3 2 + 5 * 1 -
```



Binary expression trees

Now consider the expression 3 + 2 * 5 – 1.

Things begin the same way. 3 + 2 gives:

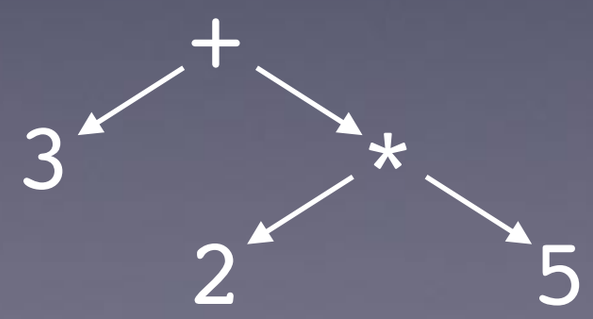


Binary expression trees

Now consider the expression 3 + 2 * 5 – 1.

But the * 5 changes things.

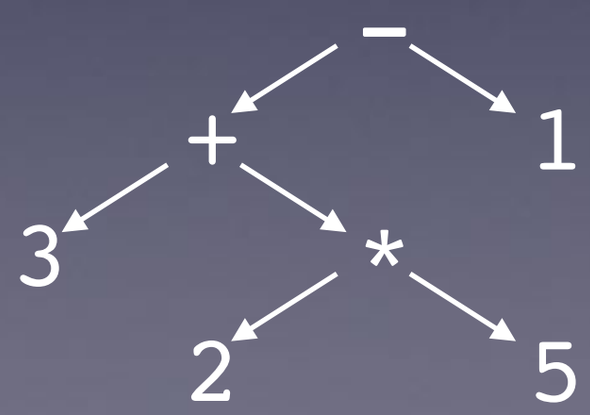
The * has higher precedence than the +, so 2 * 5 has to happen first. It has to go lower in the expression tree:



Binary expression trees

Now consider the expression 3 + 2 * 5 – 1.

The – 1 has lower precedence , so it goes higher in the expression tree:



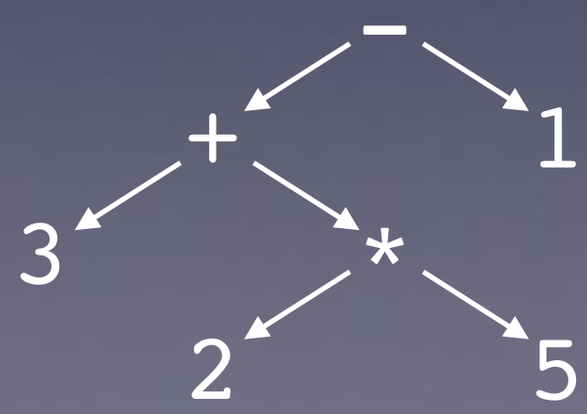
Binary expression trees

Now consider the expression 3 + 2 * 5 – 1.

A postorder tree traversal gives this RPN expression:

Again, you can try this on your RPN calculator.

```
3 2 5 * + 1 –
```



Binary expression trees

Let's formalize this using the following procedure:

First we fully parenthesize the expression 3 + 2 * 5 – 1. Higher precedence operators are evaluated first, otherwise it's left to right:

((3 + (2 * 5)) – 1)

Next split the expression on whitespace into a series of **tokens**. Each token is either a **number**, an **operator**, an **opening parenthesis**, or a **closing parenthesis**.

The actual tree building procedure begins with an empty root node and scans the list of tokens from left to right.

Binary expression trees

The tree building procedure begins with an empty root node set as the current node

Then it scans the list of tokens from left to right processing each one in turn.

((3 + (2 * 5)) – 1)

Binary expression trees

Scanning **((3+(2*5))-1)**

If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

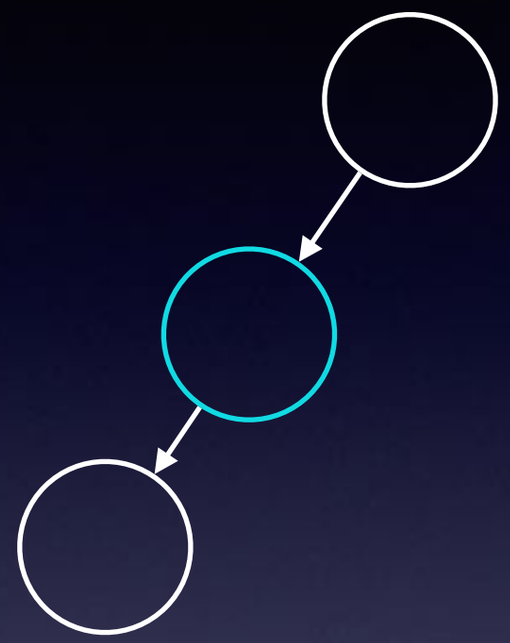
Scanning **((3 + (2 * 5)) – 1)**

If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

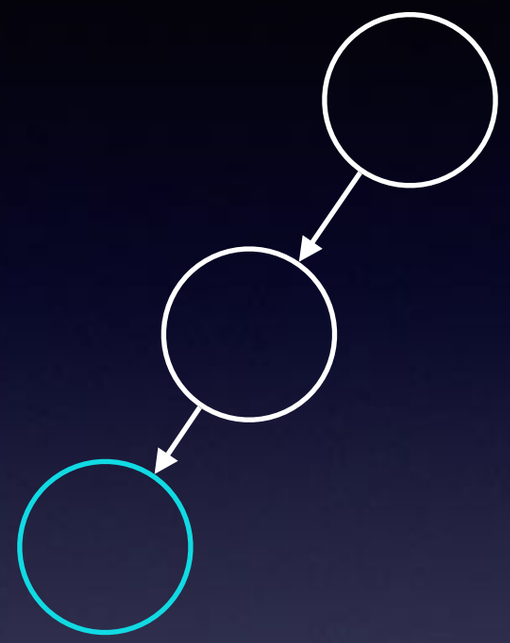


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

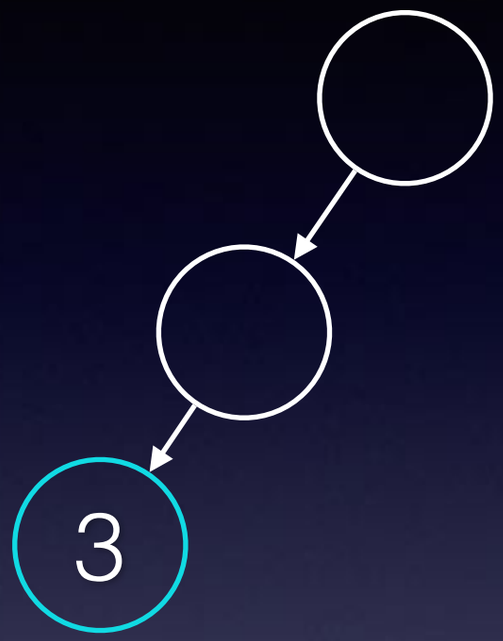


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

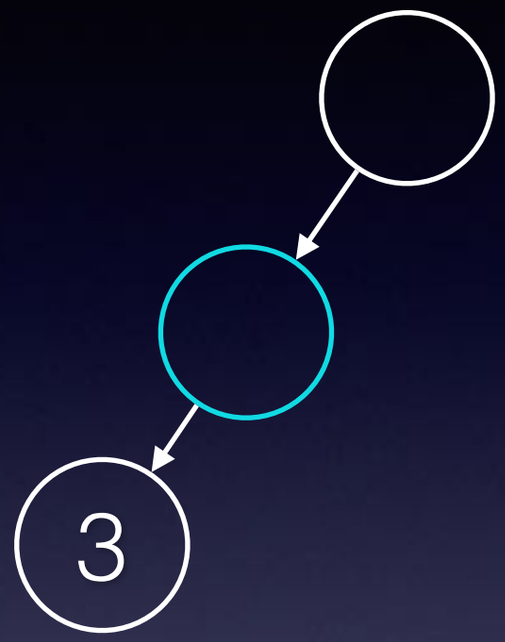


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

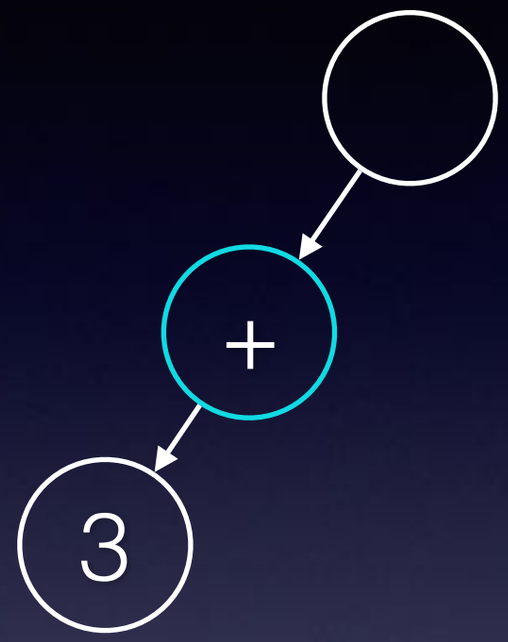


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

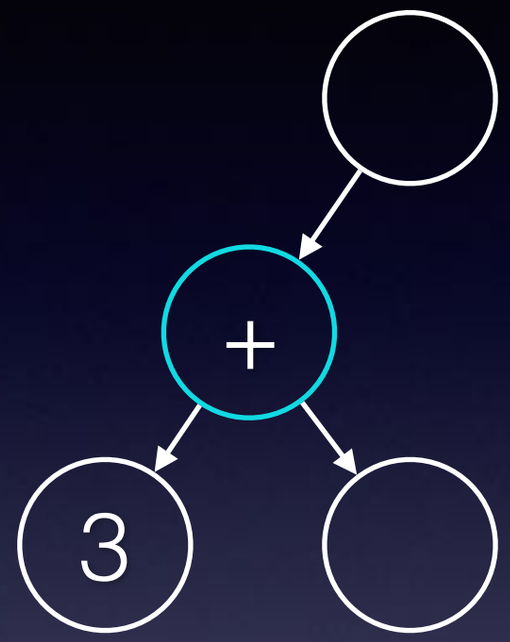


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

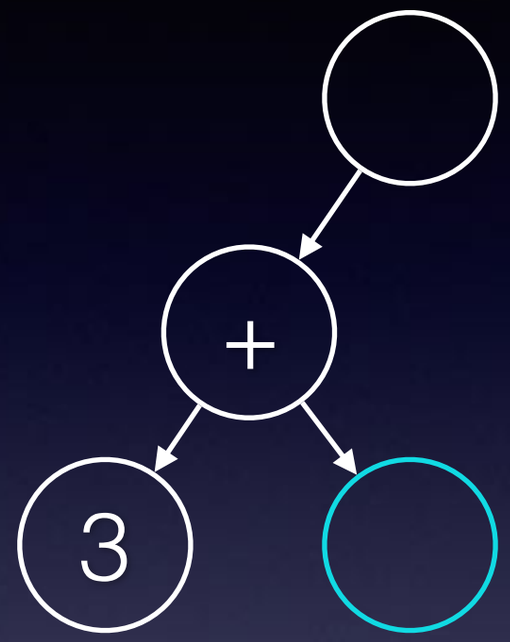


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

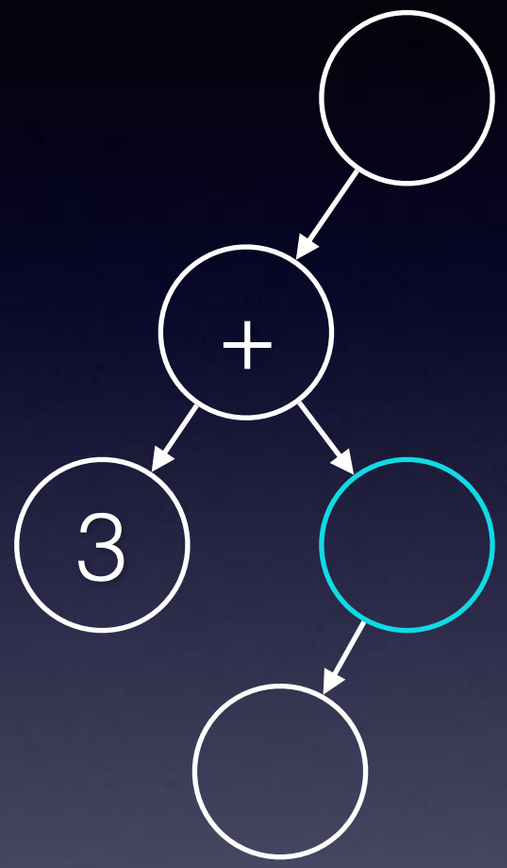


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

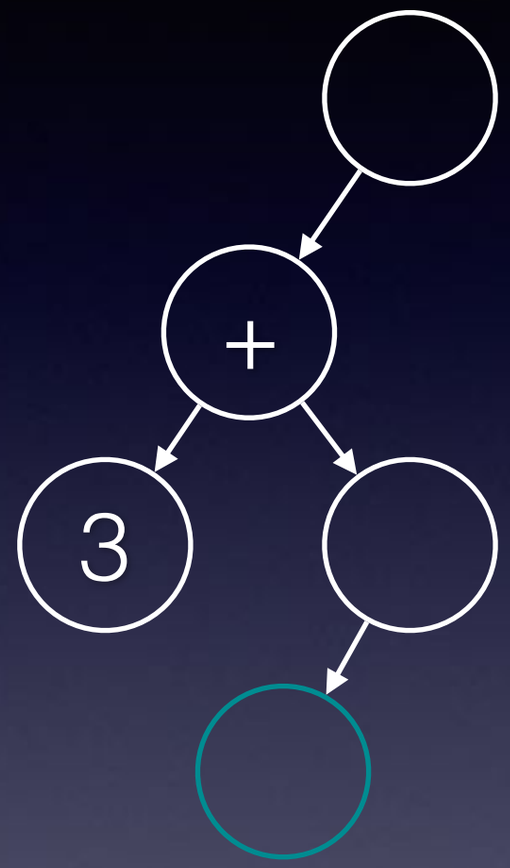


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

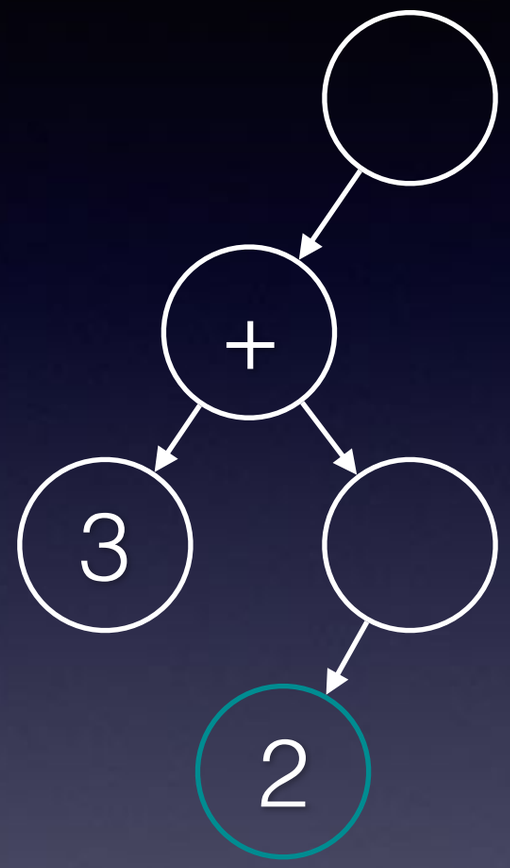


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

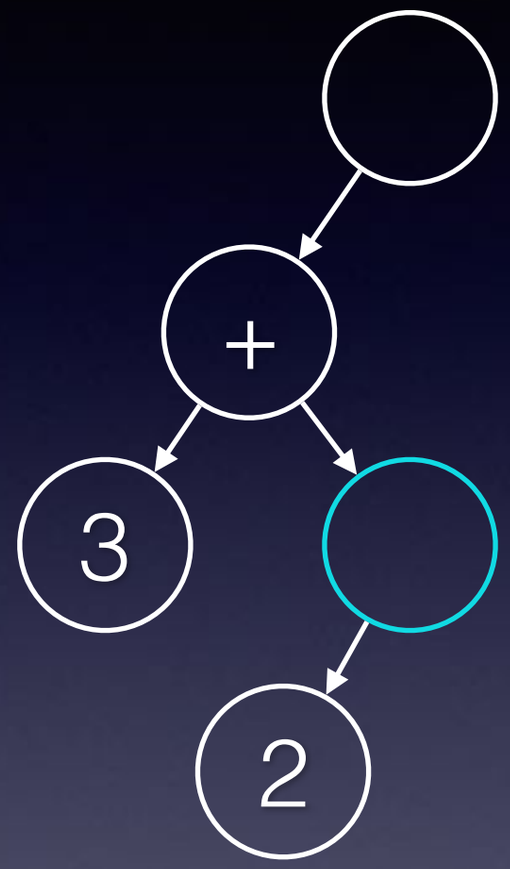


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**



If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

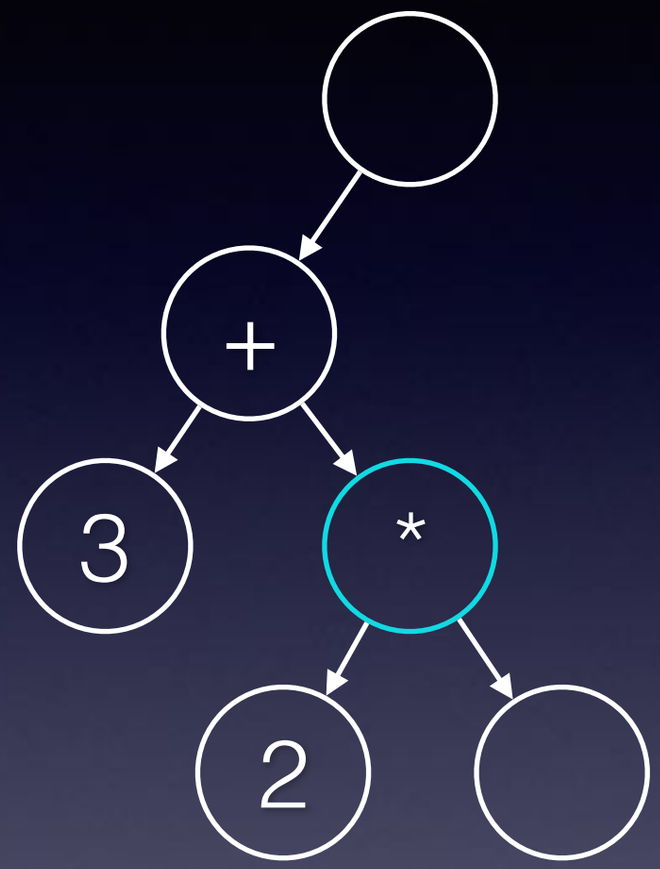
Scanning **((3+(2*5))-1)**

If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))−1)**

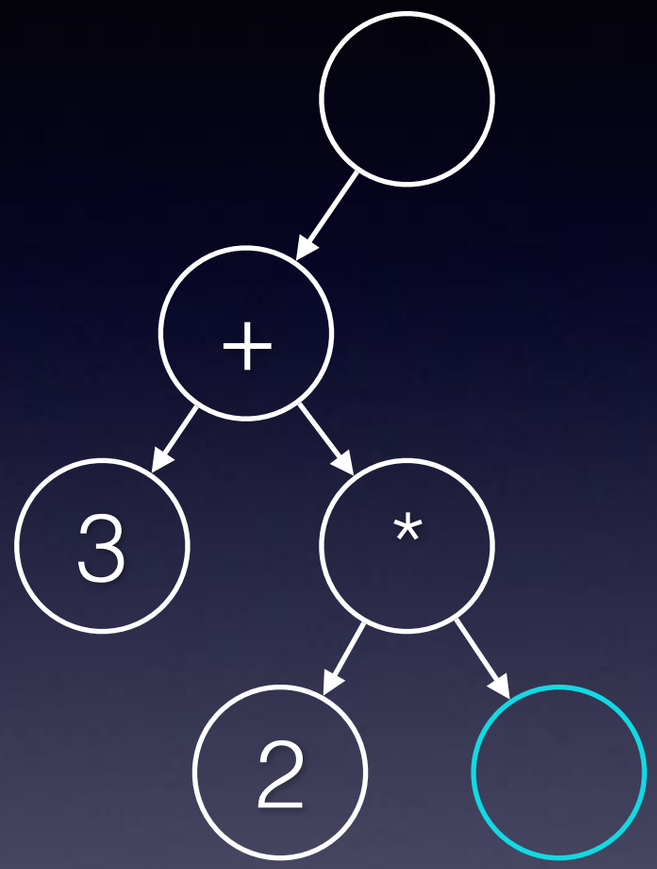


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

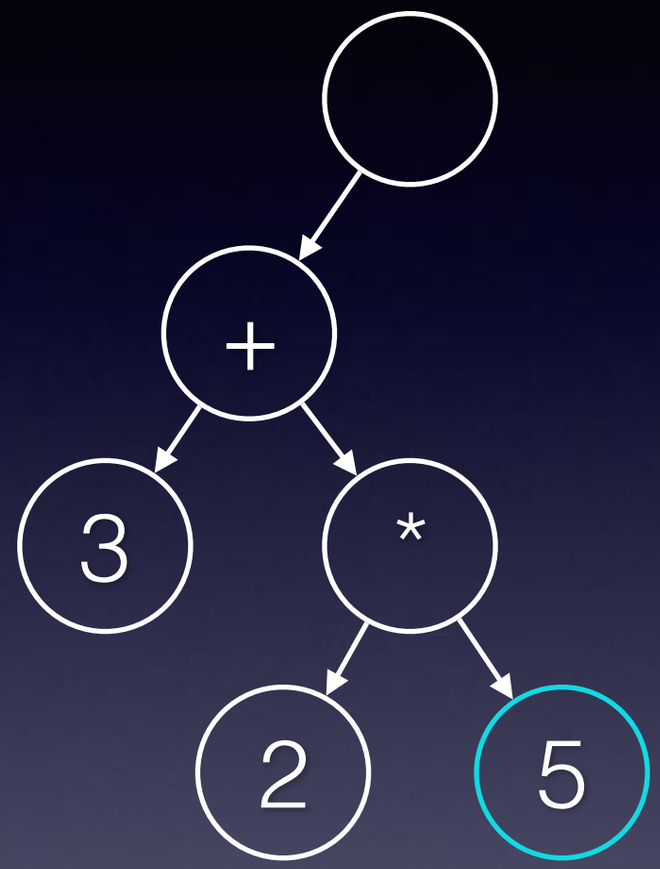


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

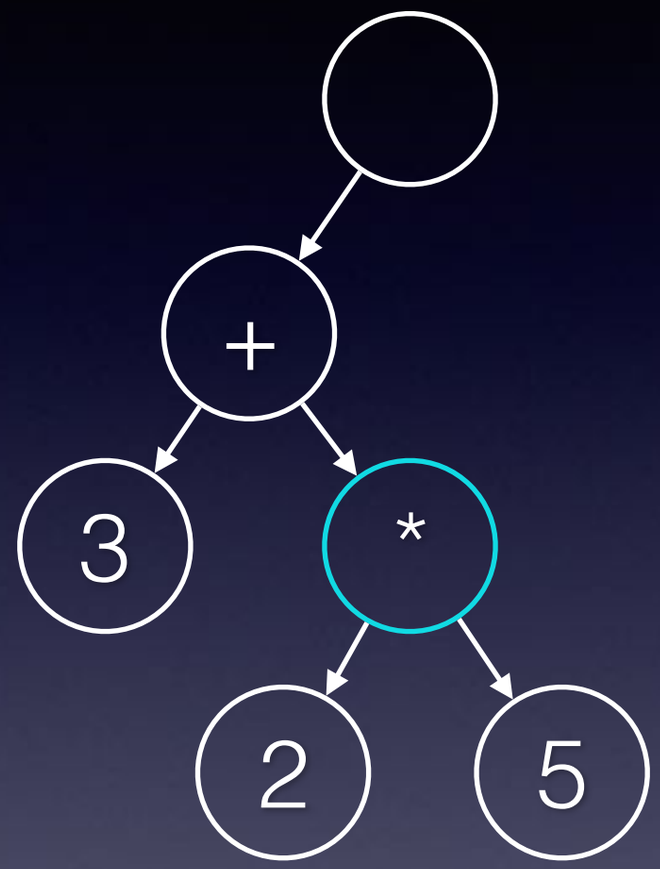


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

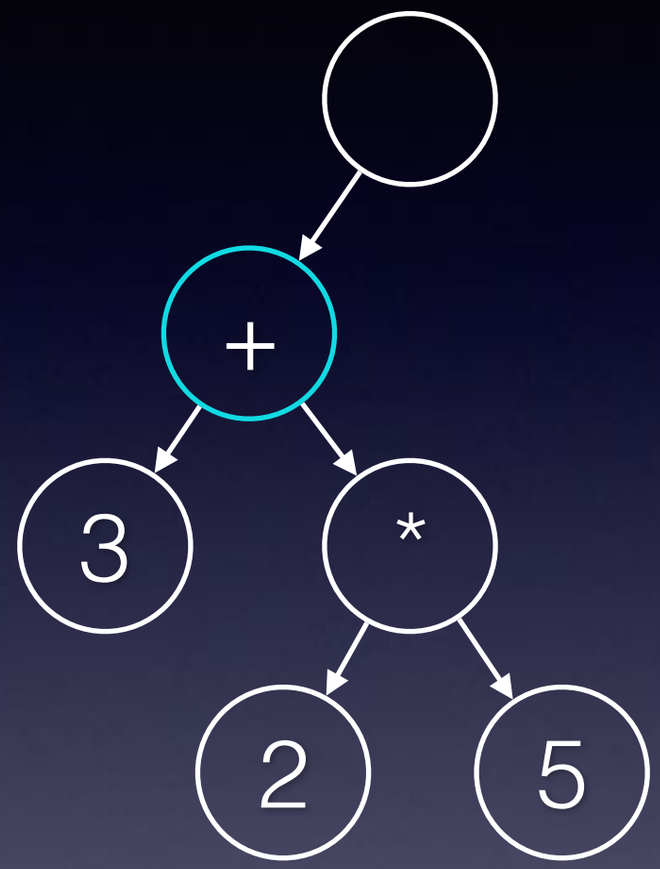


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

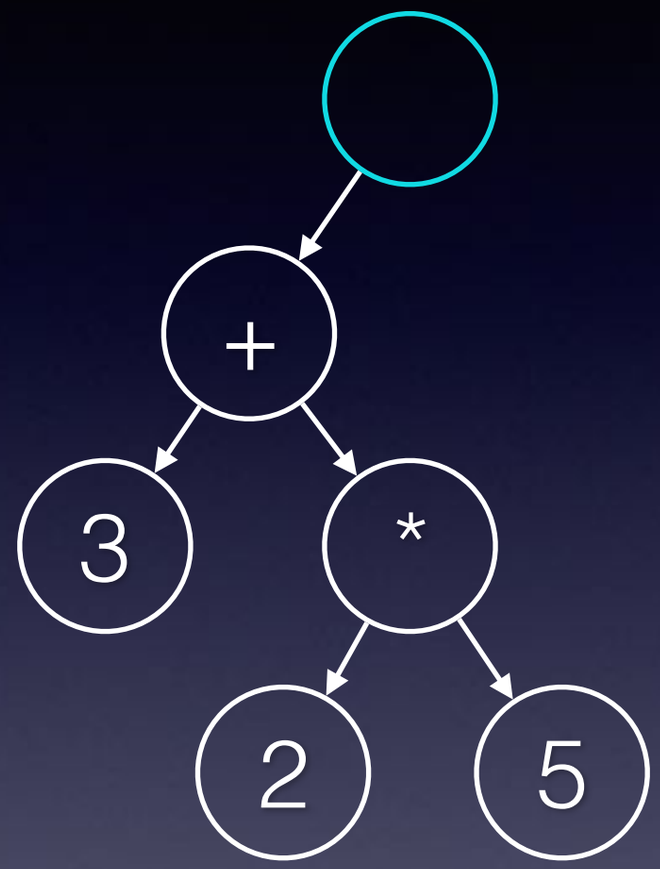


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

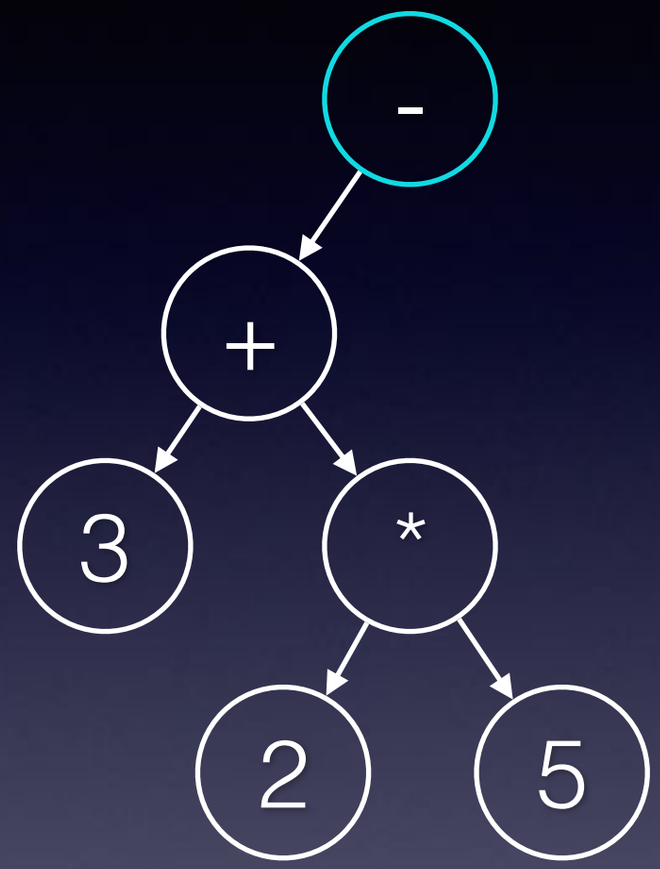


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

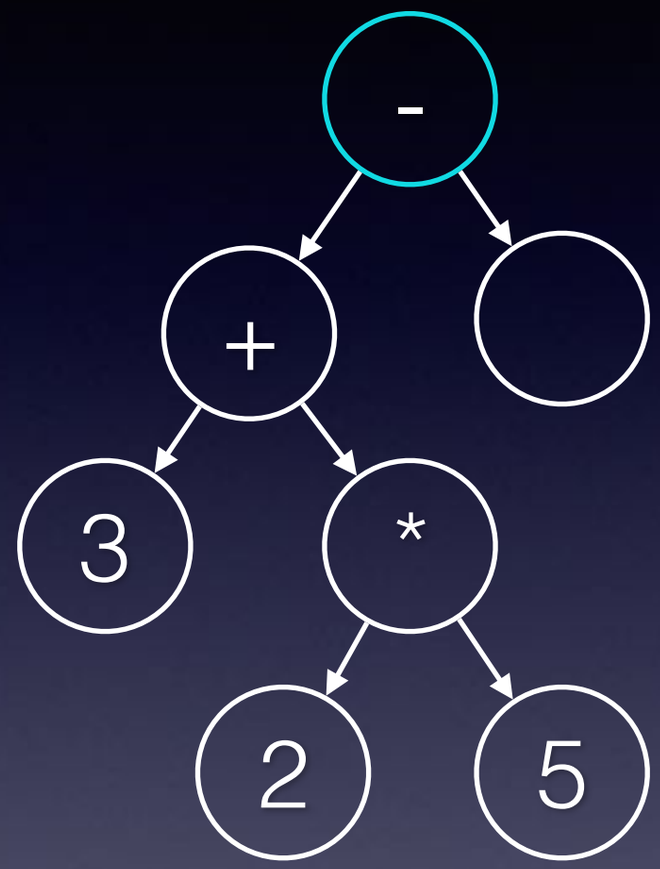


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

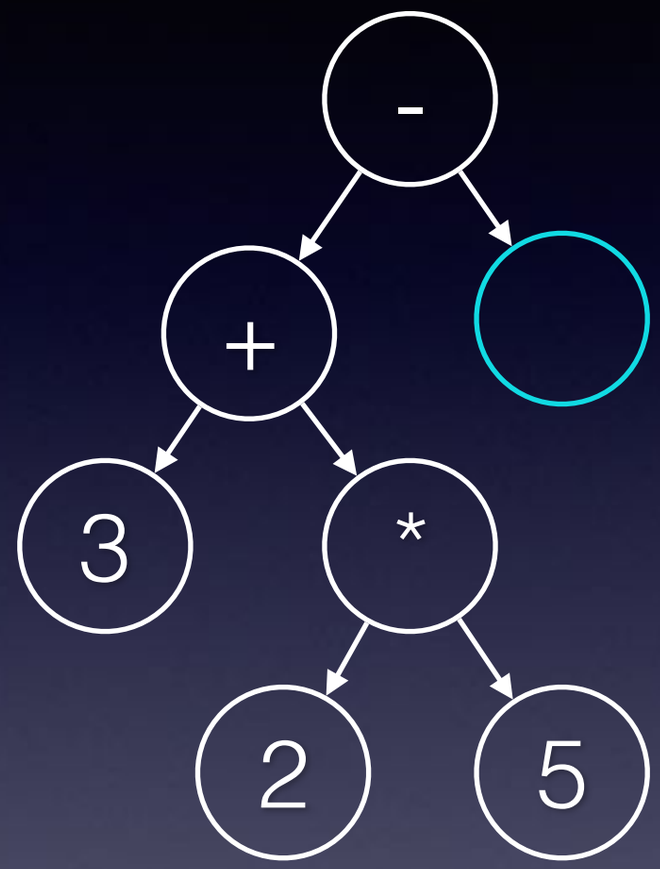


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))−1)**

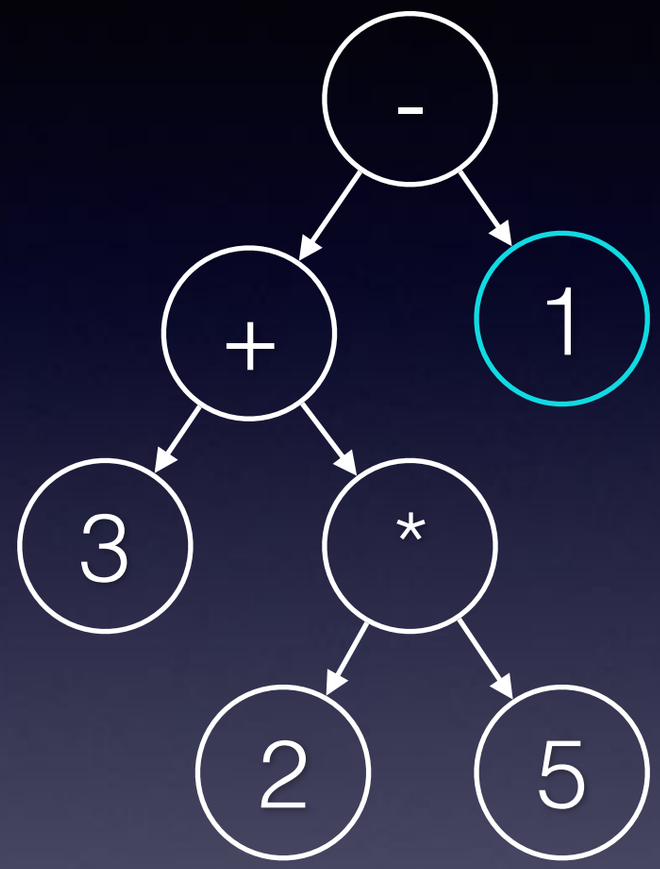


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

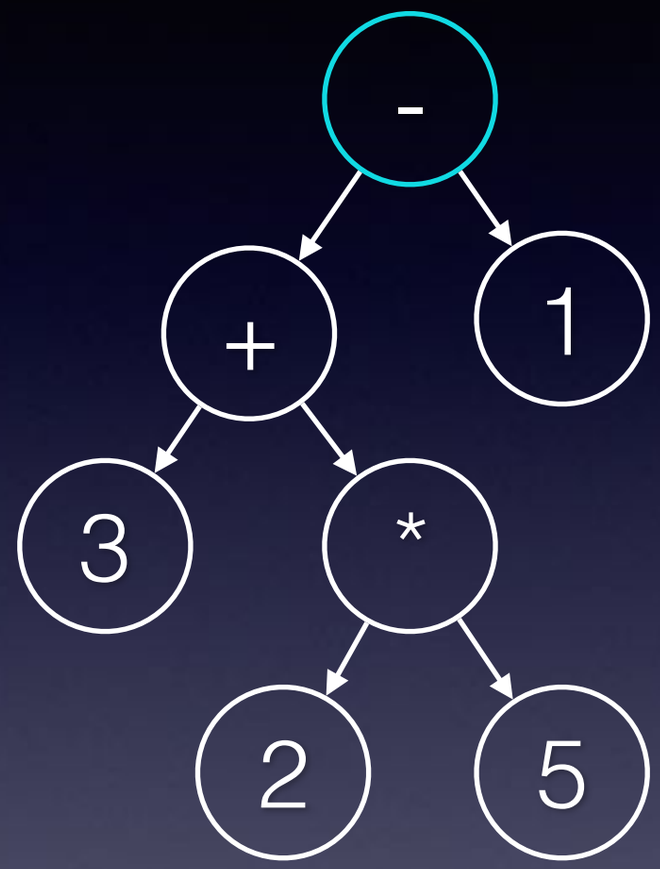


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3 + (2 * 5)) – 1)**

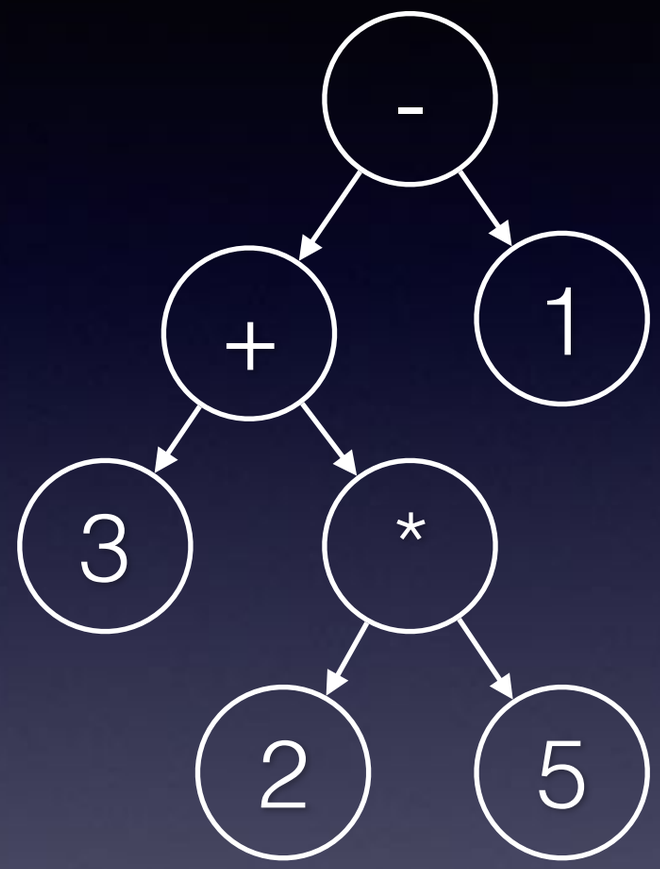


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

Scanning **((3+(2*5))-1)**

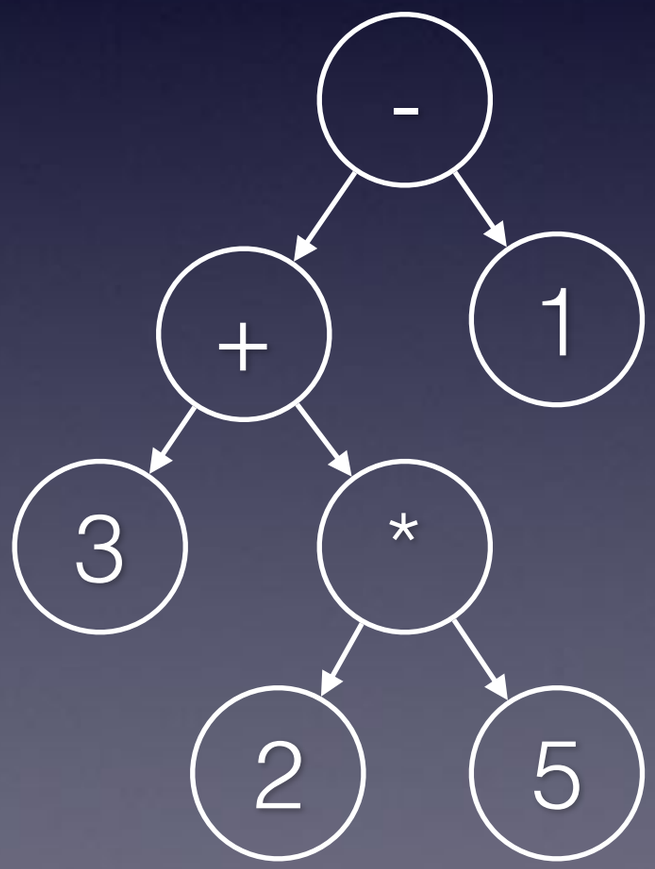


If the current token is a:

1. **opening parenthesis** add a new node as the left child of the current node, and descend to the left child.
2. **operator** set the root value of the current node to the operator. Add a new node as the right child of the current node and descend to the right child.
3. **number**, set the root value of the current node to the number and return to the parent.
4. **closing parenthesis**, go to the parent of the current node.

Binary expression trees

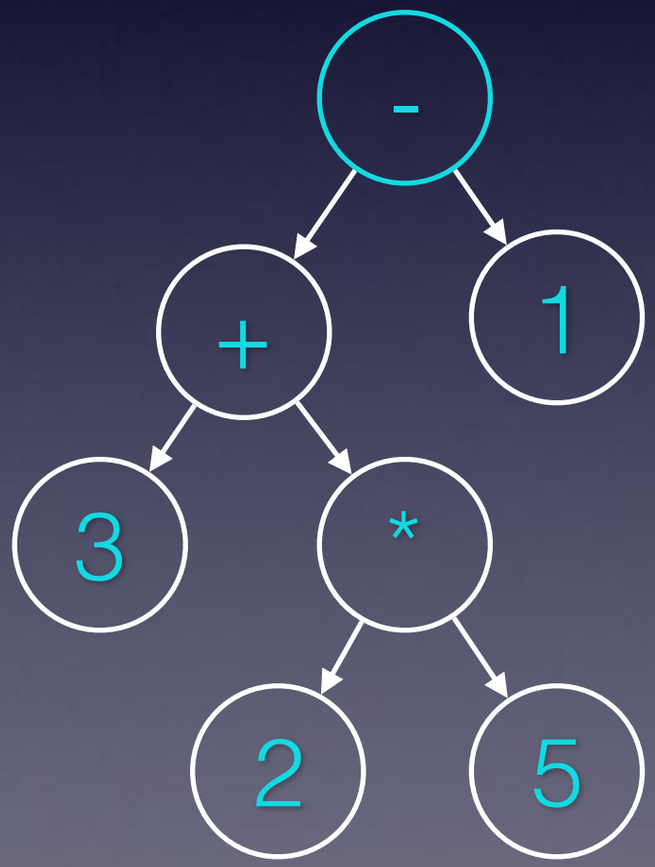
Postorder traversal




```
if the tree is empty
  return
else
  apply postorder to the
    left subtree
  apply postorder to the
    right subtree
  visit (process) the root
```

Binary expression trees

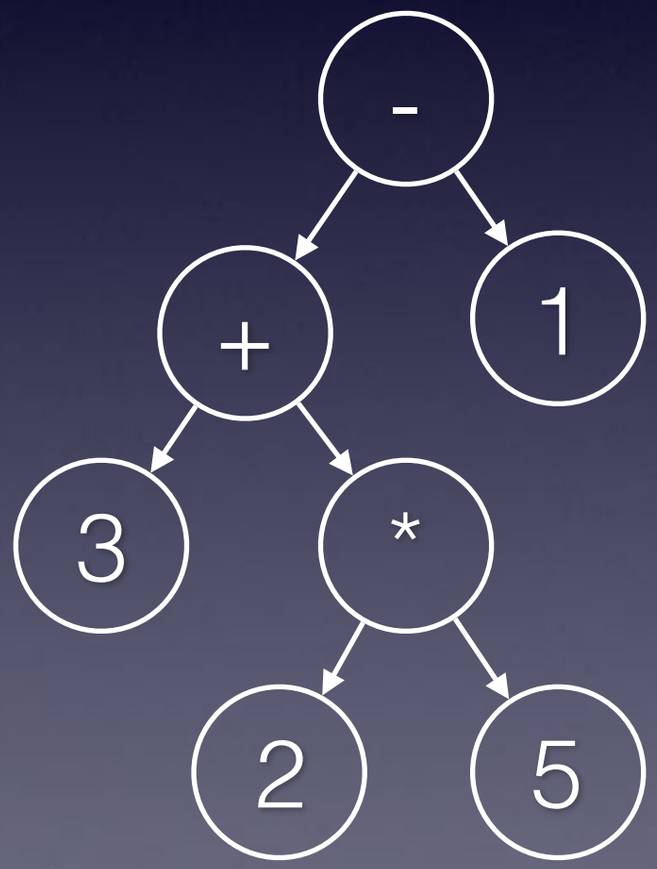
Postorder traversal 3 2 5 * + 1 -




```
if the tree is empty
  return
else
  apply postorder to the
    left subtree
  apply postorder to the
    right subtree
  visit (process) the root
```

Binary expression trees

Inorder expression printing (traversal)

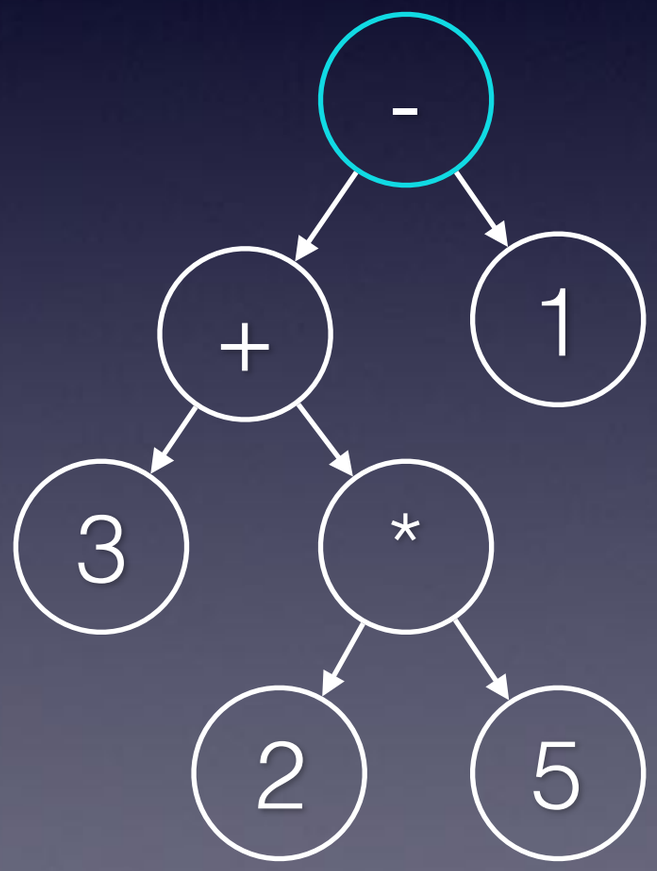



```
if the tree is empty
  return
else
  if non empty left subtree
     print '('
  apply inorderexp to the
    left subtree
  print the root
  apply inorderexp to the
    right subtree
  if non empty right subtree
     print ')'
```

Binary expression trees

Inorder expression printing (traversal)

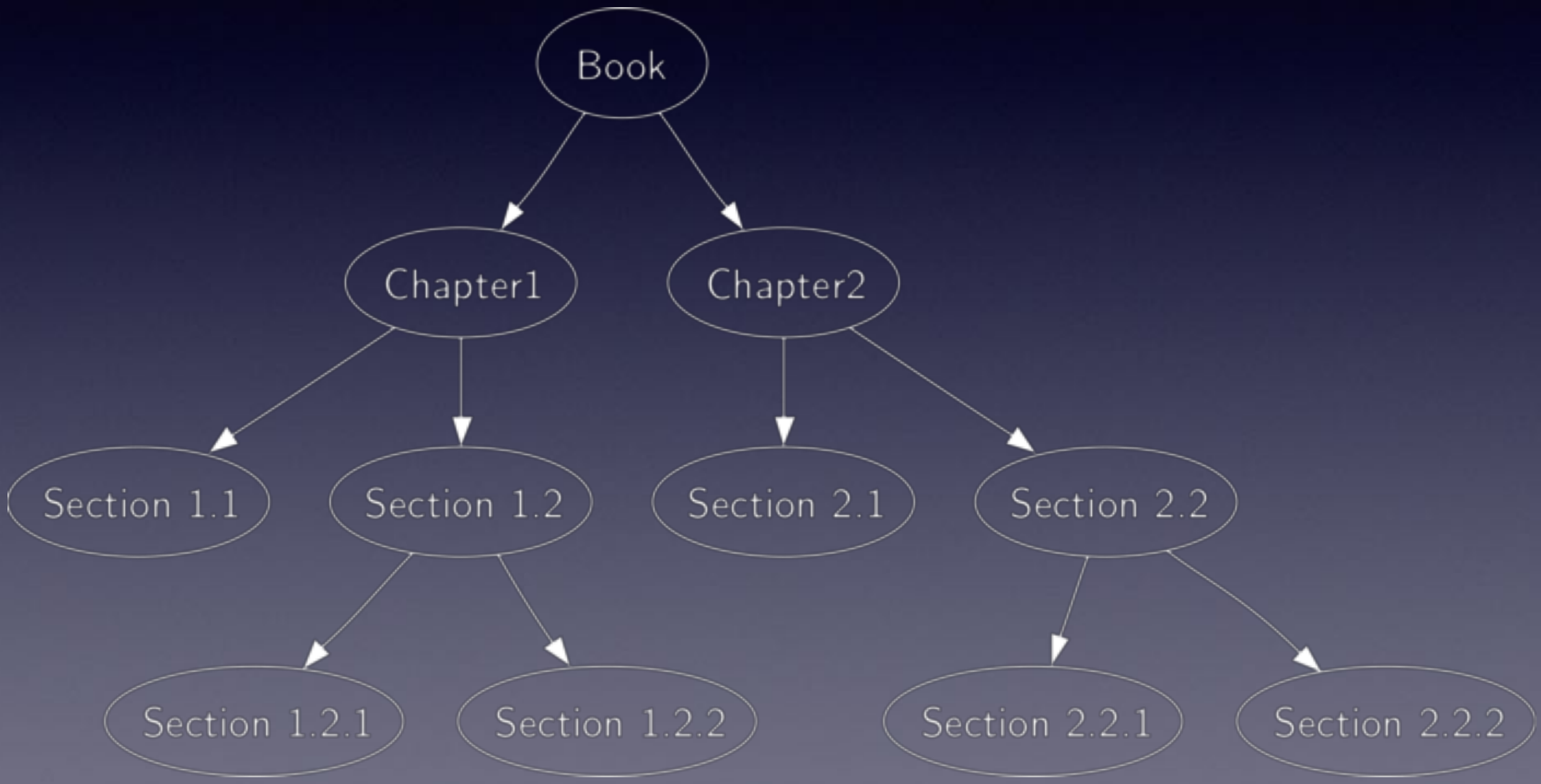
((3+(2*5))-1)




```
if the tree is empty
  return
else
  if non empty left subtree
     print '('
  apply inorderexp to the
    left subtree
  print the root
  apply inorderexp to the
    right subtree
  if non empty right subtree
     print ')'
```

What Traversal Here?

Which traversal reconstructs a Table of Contents?

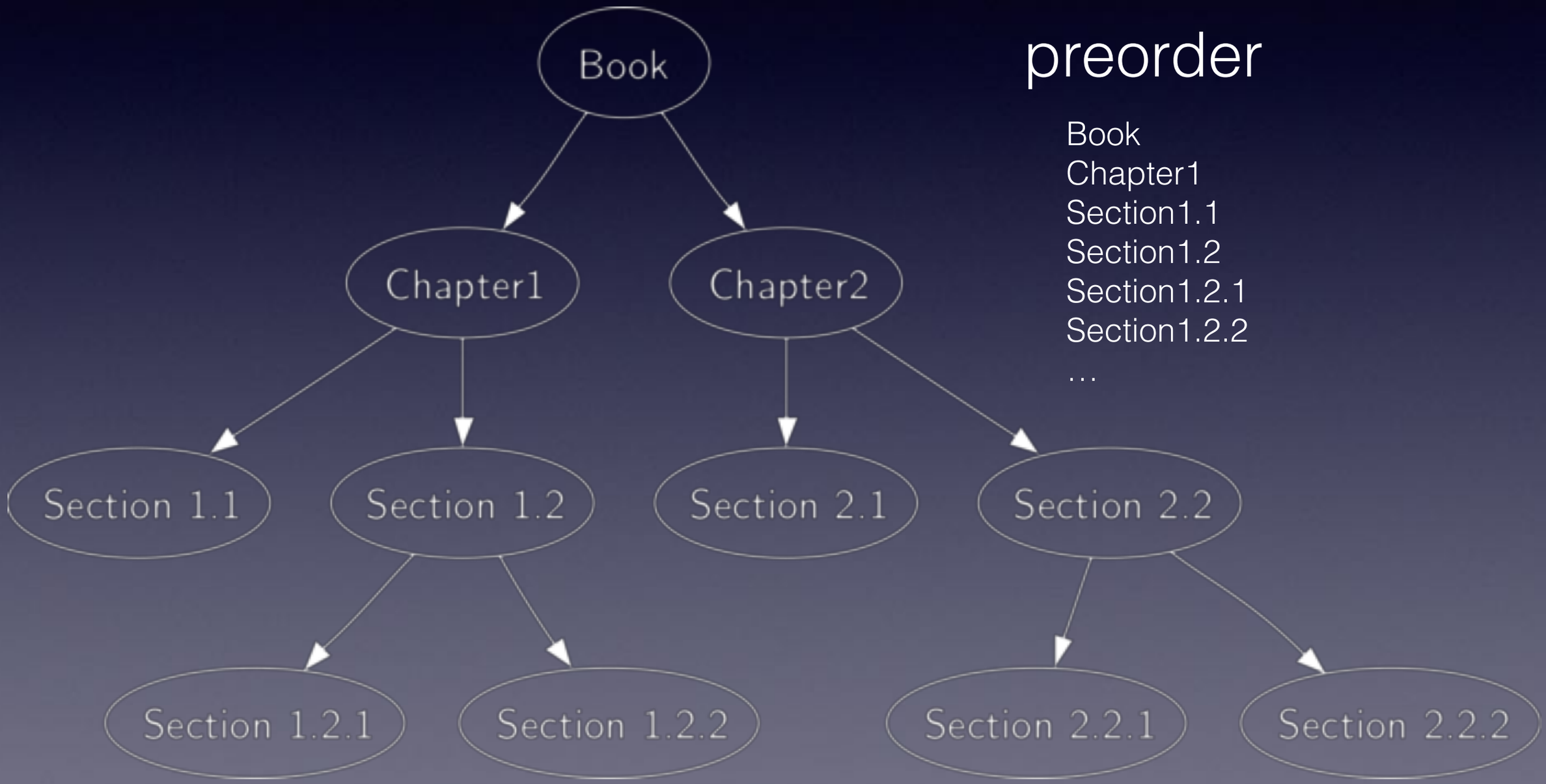


What Traversal Here?

Which traversal reconstructs a Table of Contents?

preorder

Book
Chapter1
Section1.1
Section1.2
Section1.2.1
Section1.2.2
…



Binary Search Trees

Binary search trees (BSTs) make it possible to manage lots of data while providing fast search, fast insertion, and fast deletion.

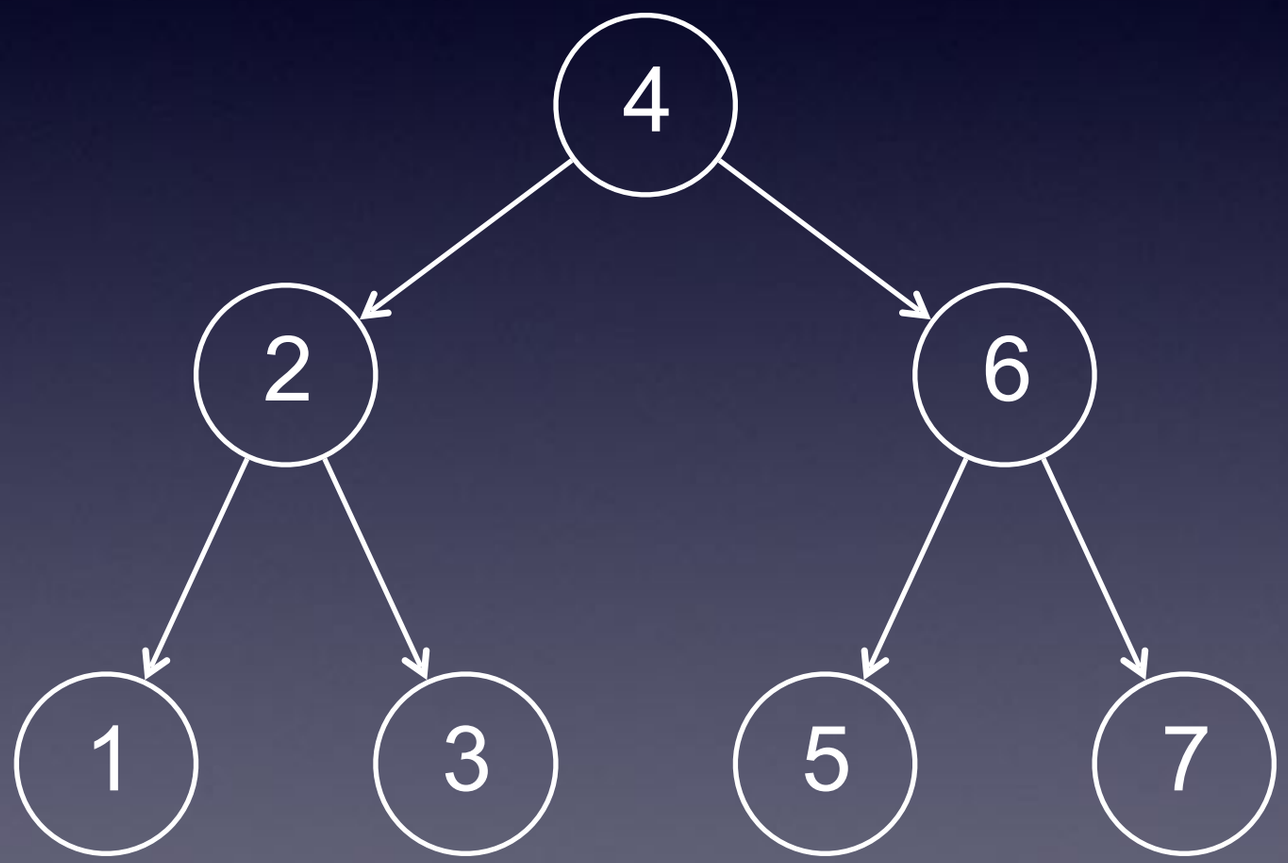
If the BST has a “good” shape, these operations have O(log n) time complexity.

But the height of the tree has to be small relative to the number of nodes in the tree.

Operations are speedy as long as the tree is “shallow”. (e.g., complete binary search tree, or one which isn’t necessarily complete but the “fringe” is only at the bottom level)

Attributes of a “good” BST

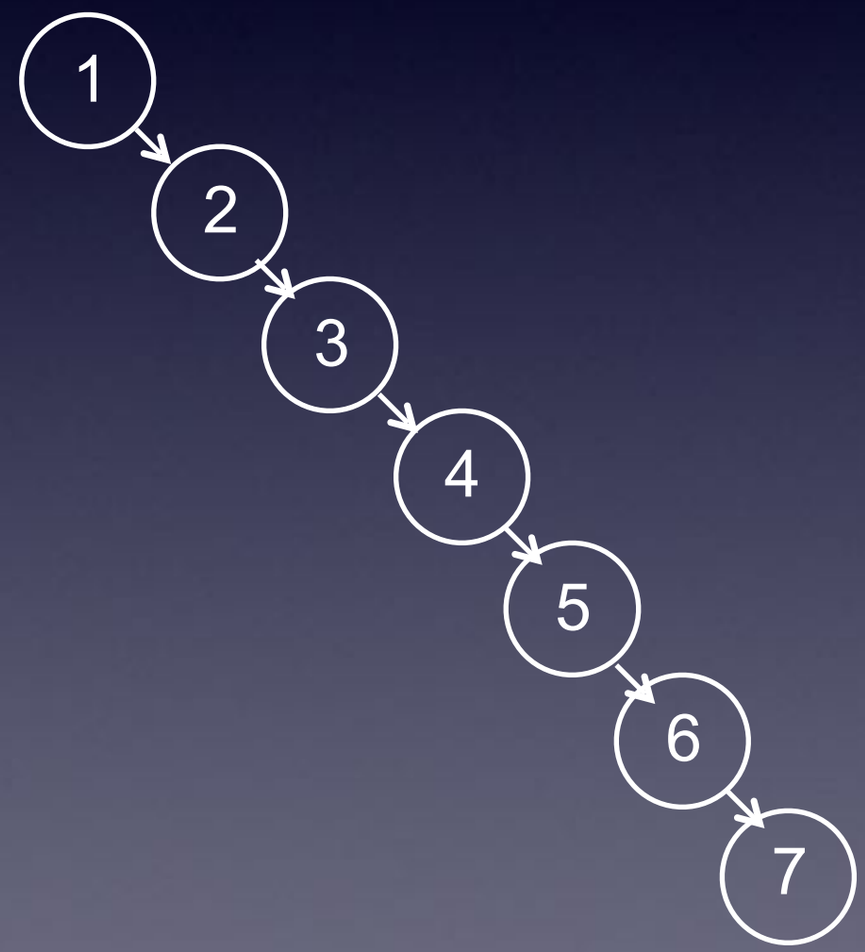
This is **really good**



O(logn) search

Attributes of a “good” BST

This is **really bad**



O(n) search

Attributes of a “good” BST

There is a simple algorithm for making a "good" BST from a sorted list. It has a similar structure to binary search of a sorted list. The value of the root node would be the midpoint of the sorted interval.

However, insertions and deletions over the course of using it would still make some branches long and others short.

The tree would no longer be “shallow” and operations require more time to finish.

This is a problem.

Is there a solution?

Of course there is.

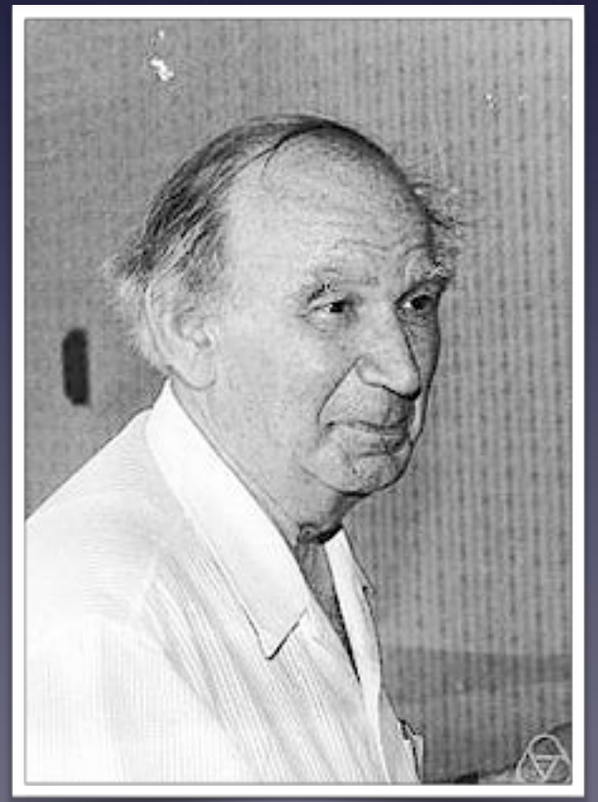
What's needed is a way to manage the height of the tree as you add things to the tree or take things away from the tree.

What's needed is **balance**.

Balanced trees are "shallow".

AVL trees

In 1962, Russian mathematicians Georgy Adelson-Velsky and Evegeny Landis published an algorithm for automatically maintaining the overall balance in a binary search tree. BSTs using this approach are called AVL trees.



AVL trees

Their algorithm maintains a tree's balance by tracking the difference in **height** of each subtree. That **balance** is the height difference between the two subtrees.

As items are inserted in (or deleted from) the tree, the balance of each subtree is updated from the insertion (or deletion) point up to the root. If the balance ever goes outside the nominal range of -1 to 1, **rotation** is applied to bring the balance back to the nominal range.

That gives us three ideas we need to understand if we're going to work with AVL trees:

height **balance** **rotation**

Balance

Balance for a tree at its root is

height(left subtree) – **height**(right subtree)*

If that difference is zero everywhere, the tree is **perfectly balanced**.

An AVL tree has a balance is between -1 and 1 everywhere in the tree. It is **balanced enough**, even though it may not look like it…

* Balance could also be height(right subtree) – height(left subtree). It works either way. Change the arithmetic accordingly.

Balanced enough

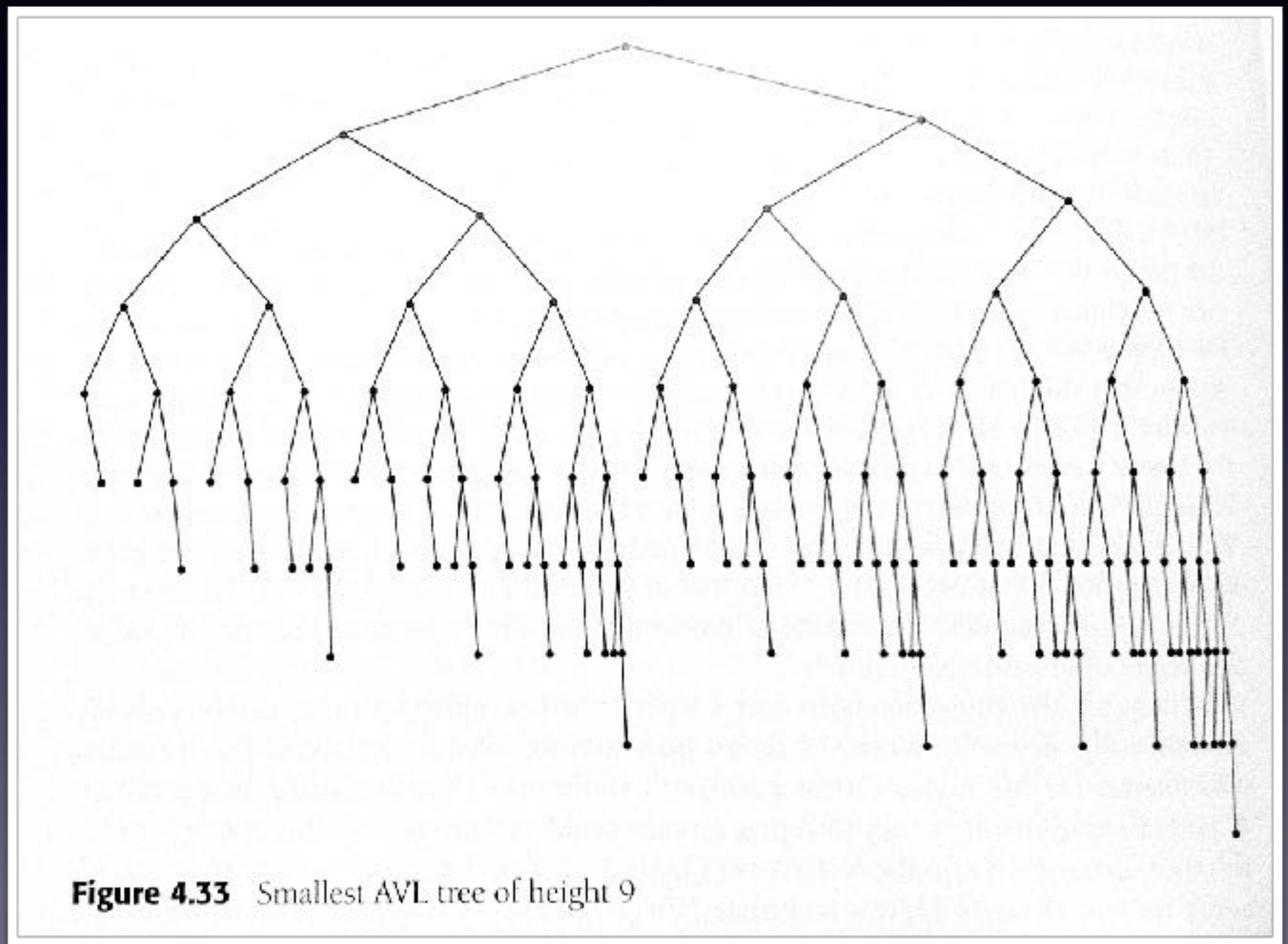


Figure 4.33 Smallest AVL tree of height 9

This is what balanced enough looks like

Balanced enough

If the balance is between -1 and 1 everywhere in the tree, then the **maximum height** of the tree has been shown* to be approximately **1.44 log n**.

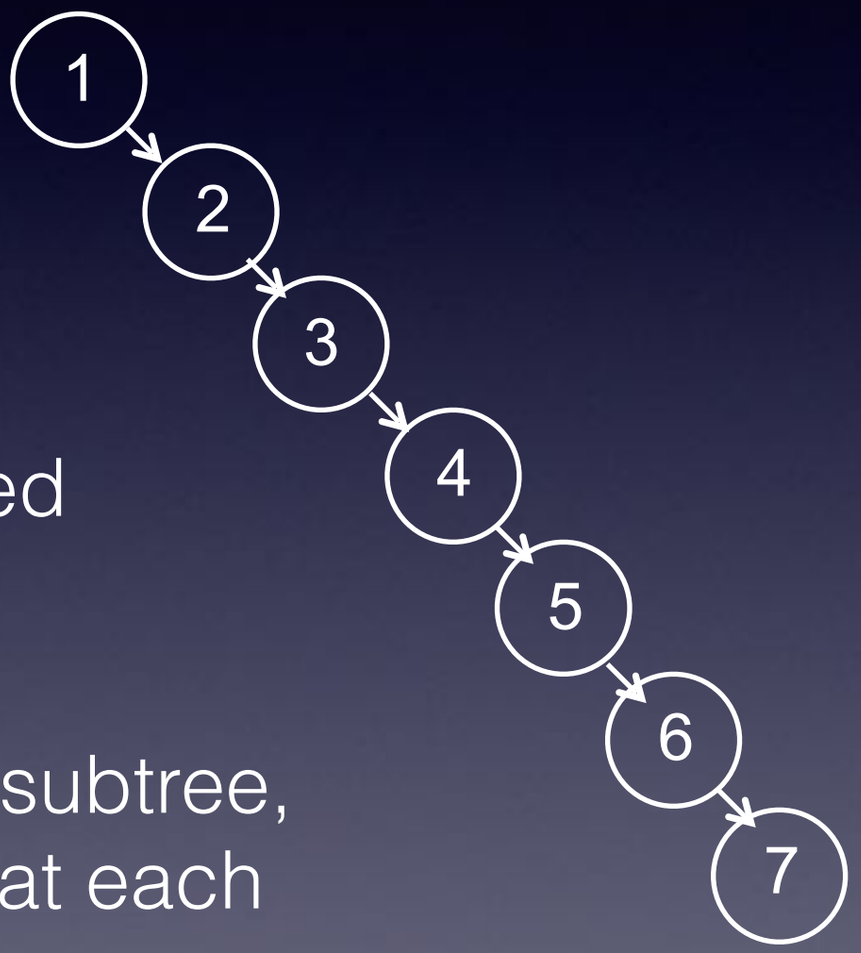
That's great. The maximum height is the worst case complexity of search. The worst case is a function of log n now, instead of a function of n in the general case.

So we don't need a perfectly balanced BST to maintain O(log n) asymptotic time complexity for search.

* We'll just accept that a proof exists. The textbook gives some additional details.

Self-balancing search trees

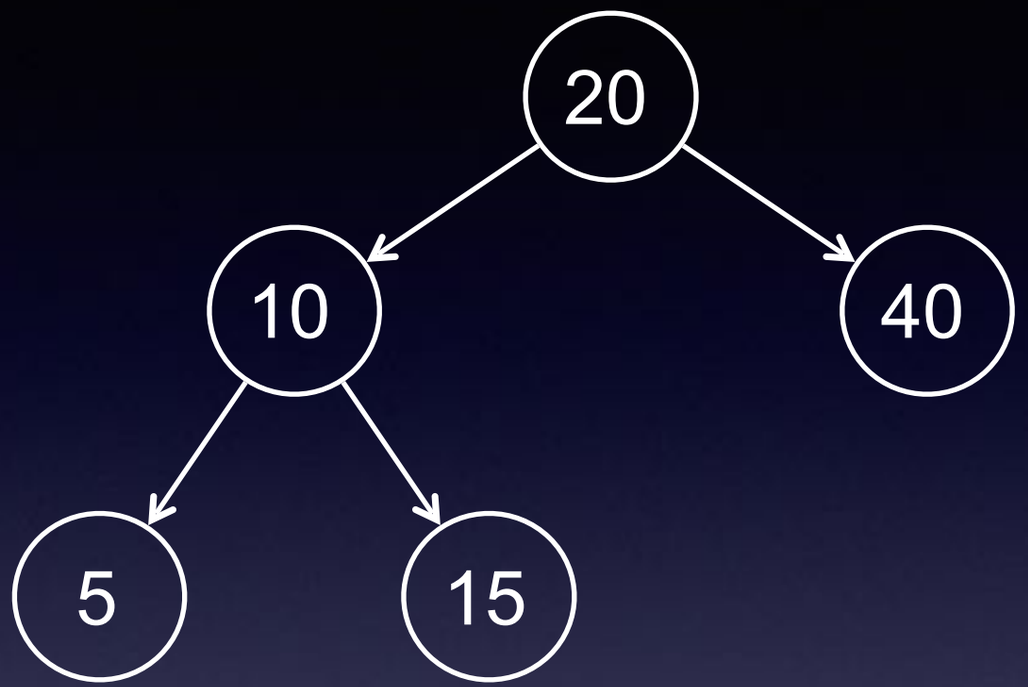
As we've seen, we can come up with a severely unbalanced BST when building the tree.



The easiest way to keep a tree balanced is to never let it become unbalanced.

So when a new node is inserted into a subtree, the AVL algorithm checks the balance at each parent node up the insertion path.

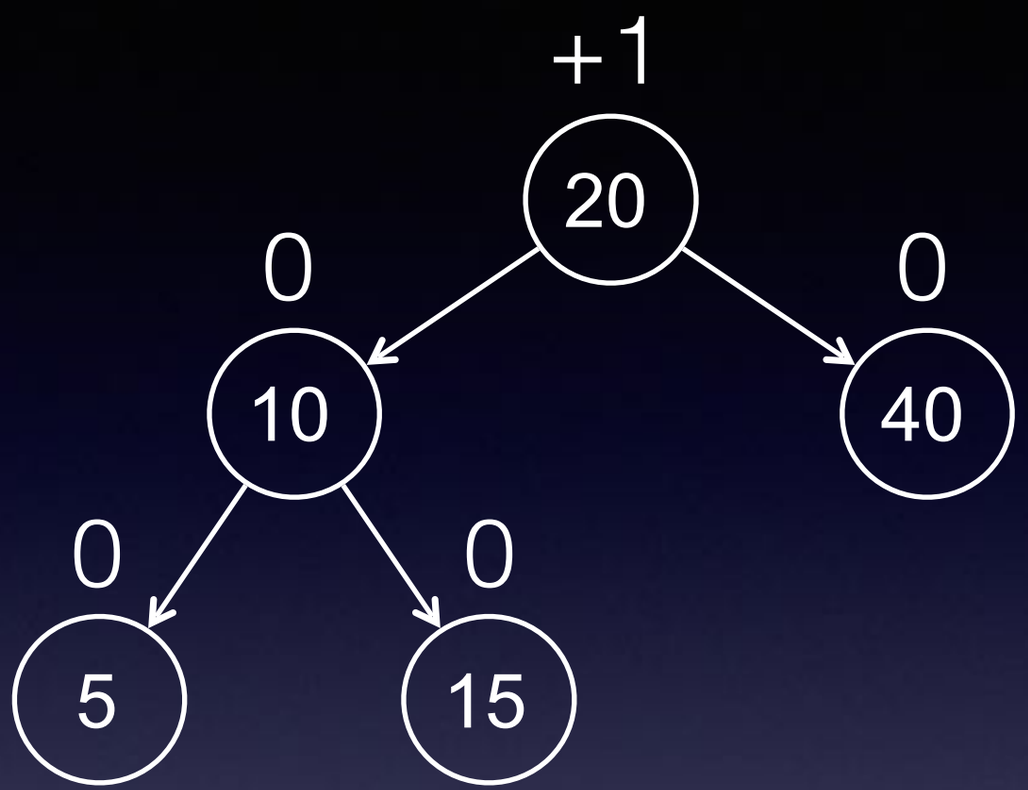
Balance



As we work with our binary search tree, we want to maintain balance to preserve our fast search times. As the balance changes from good to not so good, we need to change the relative height of the left and right subtrees that are imbalanced while preserving BST properties.

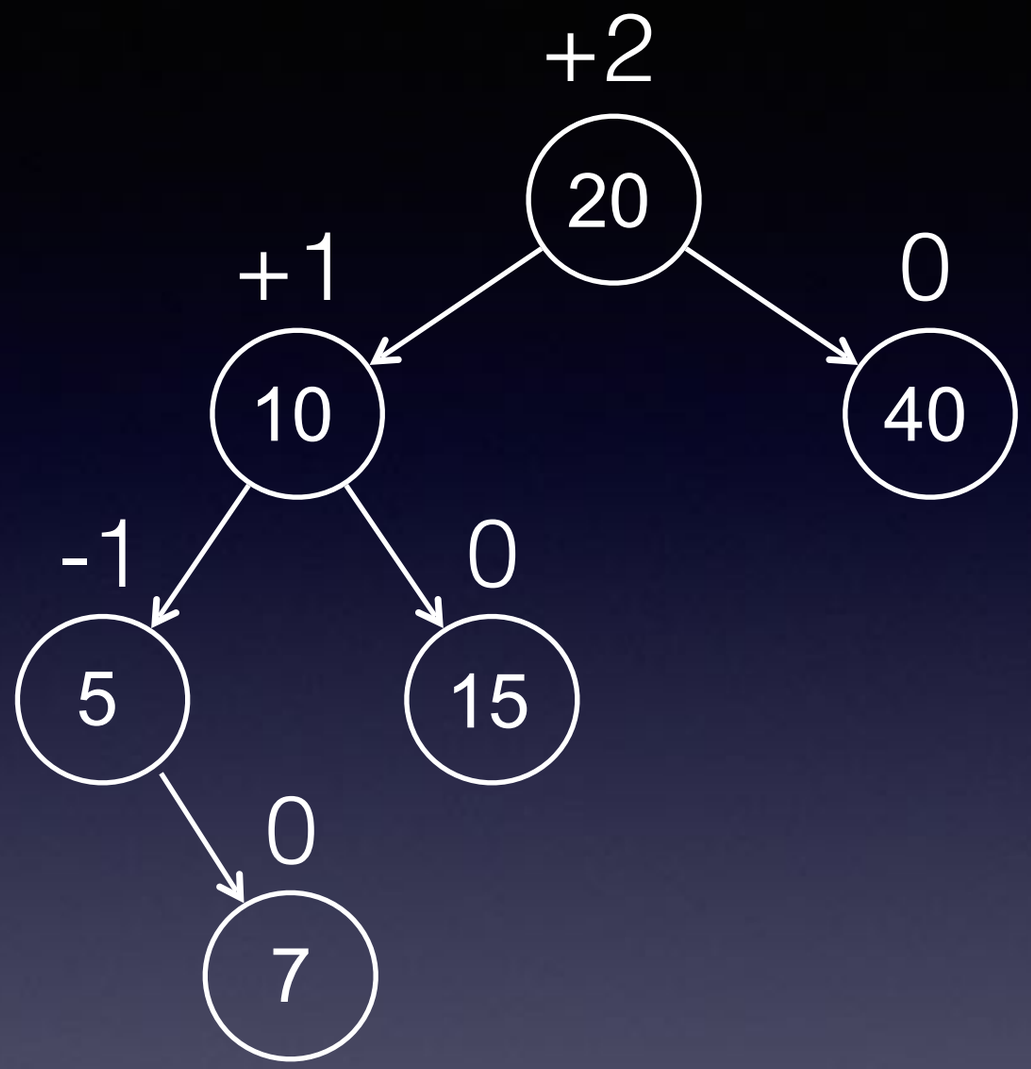
How’s the balance for this tree?

Balance



We’ve computed the balance at the root of each subtree. These numbers are good. What happens when we add the value 7 to the tree?

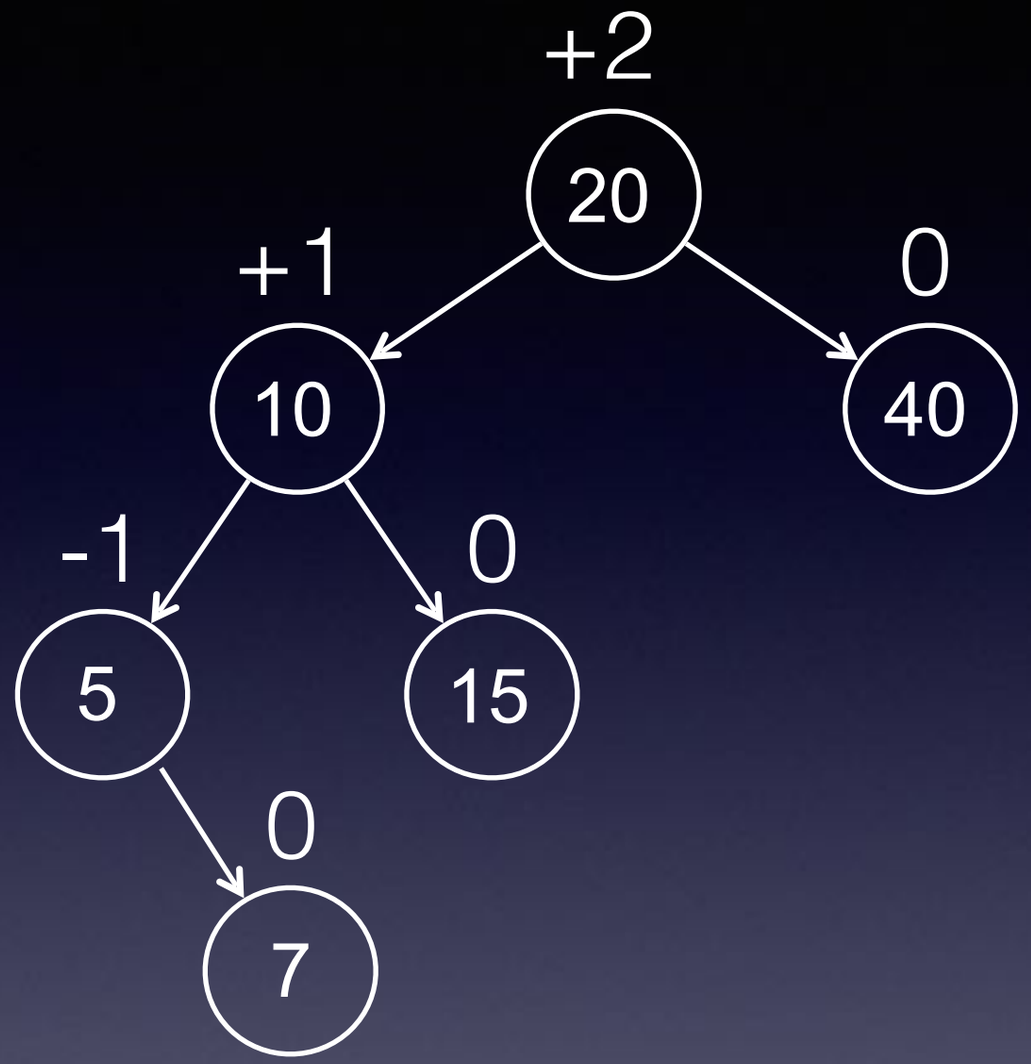
Balance



We've computed the balance at the root of each subtree. These numbers are good. What happens when we add the value 7 to the tree?

Now the balance at the root is unacceptable.

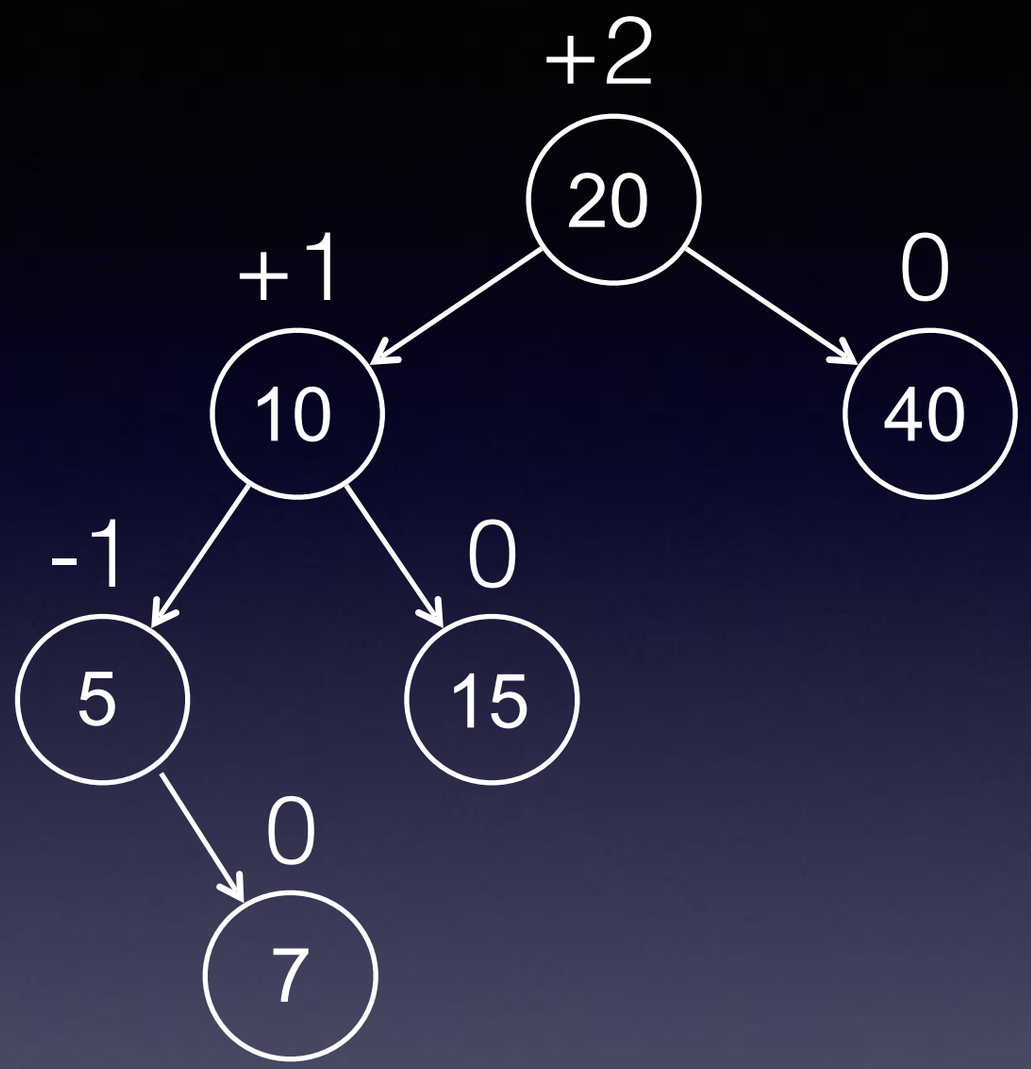
Balance



Now the balance at the root is unacceptable.

The fact that the balance is positive tells us that the tree is heavy to the left, and that we should rotate the tree to the right to correct the imbalance.

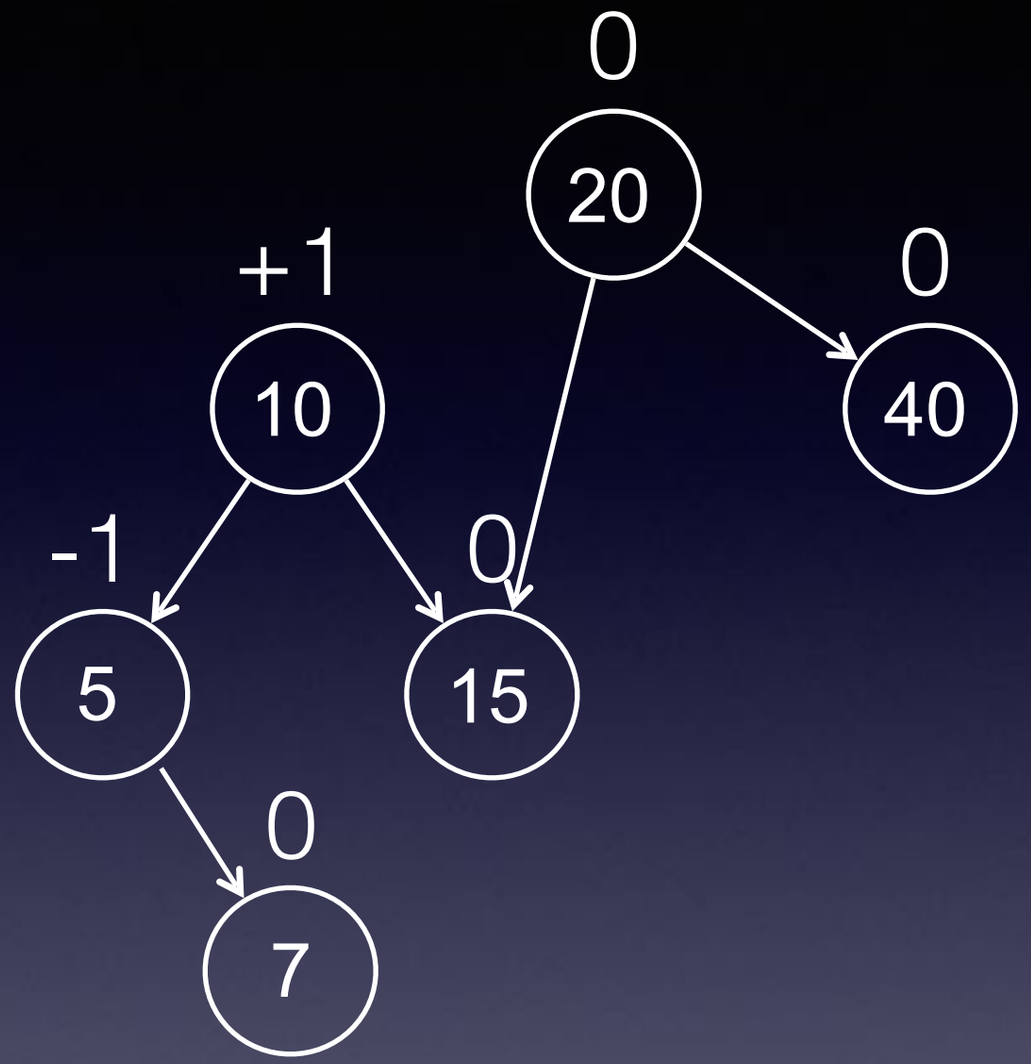
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

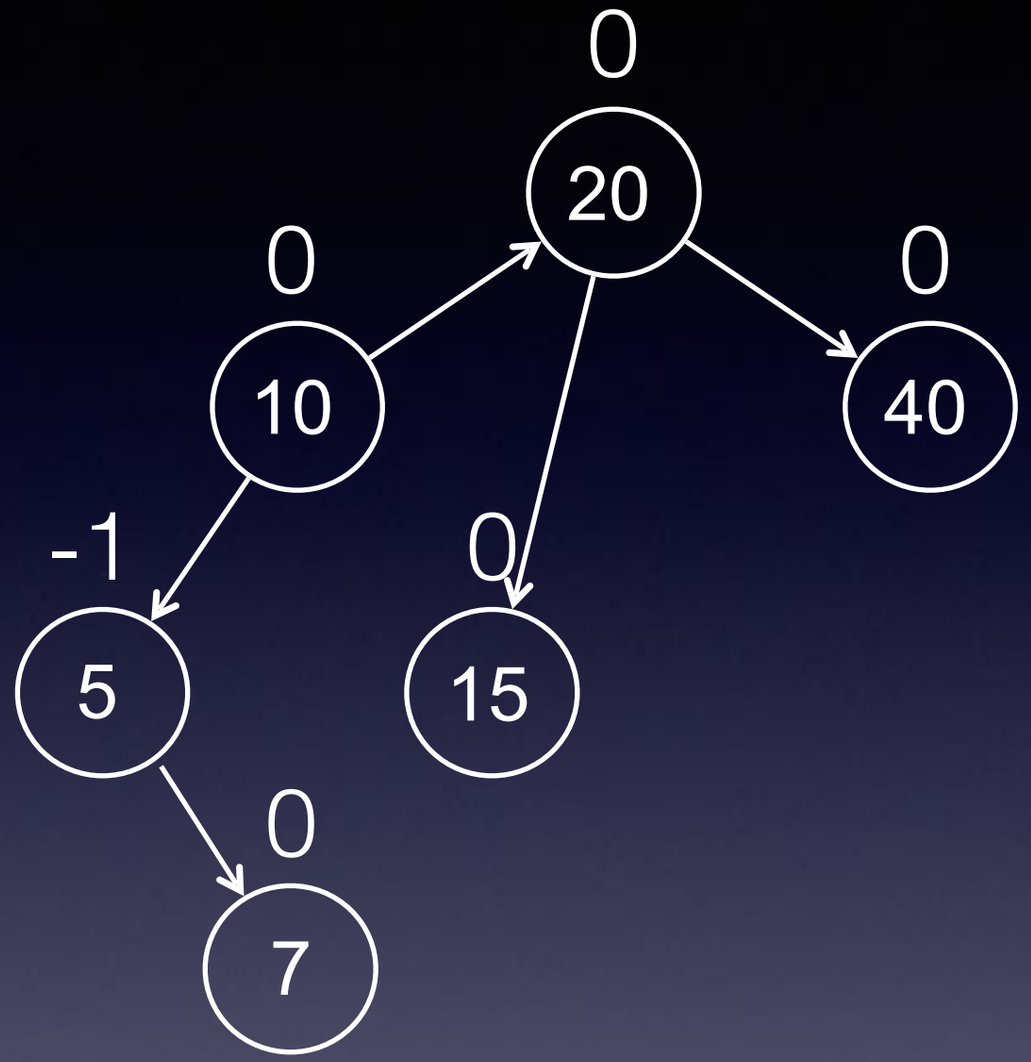
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

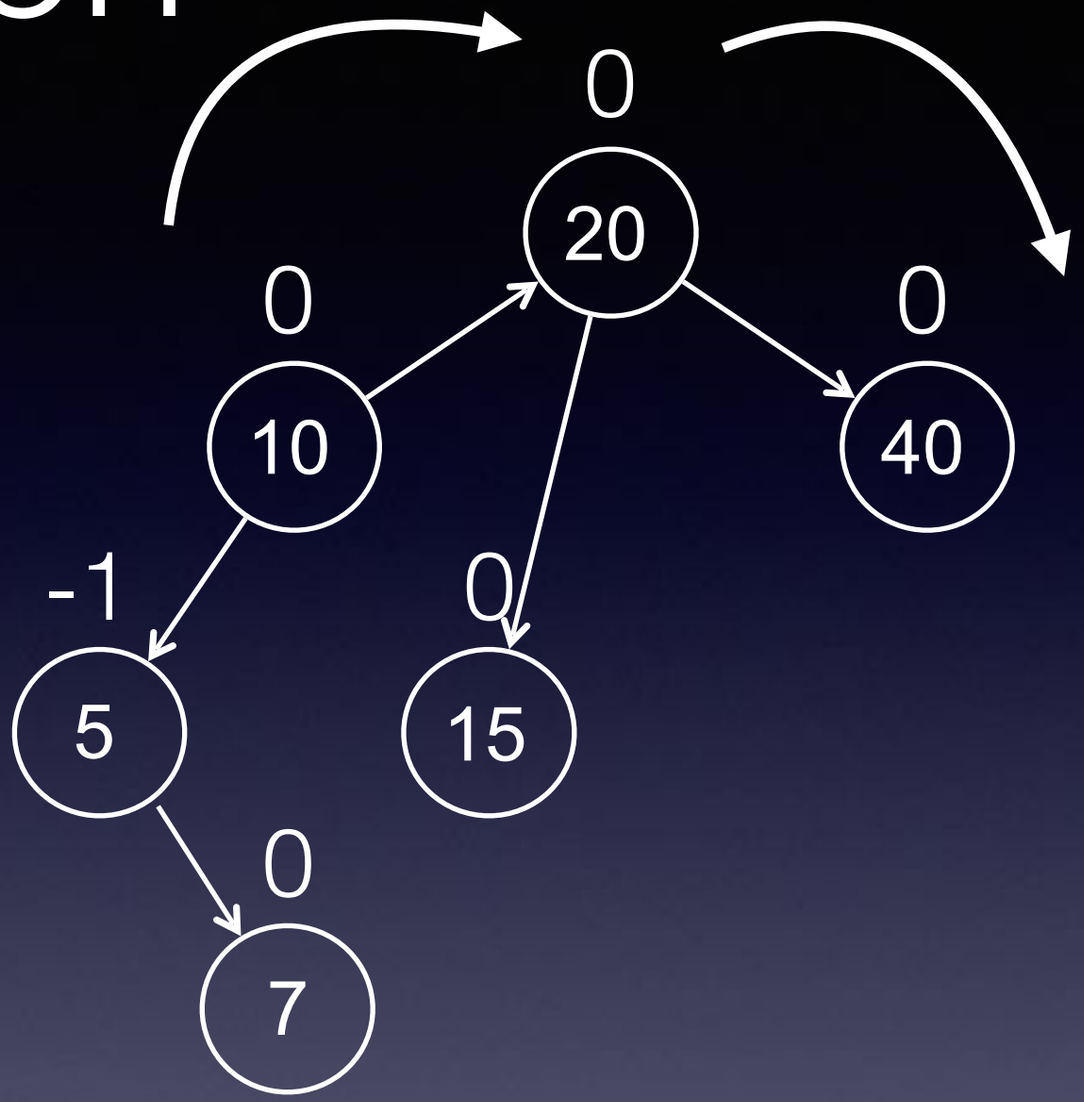
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

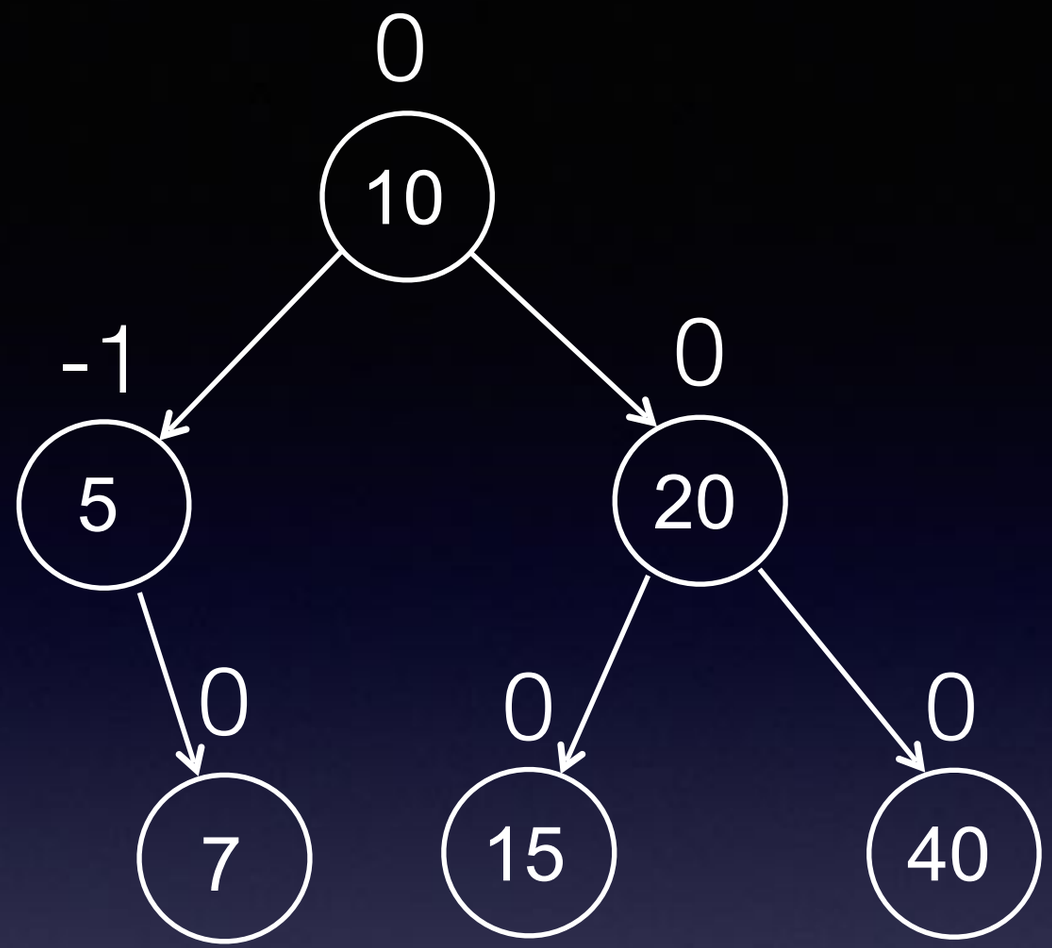
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

Rotation



And now the tree is back in balance.

How’s it done?