CSE 373

OCTOBER 11TH – TRAVERSALS AND AVL

MINUTIAE

- **Feedback for P1p1 should have gone out before class**
  - Grades on canvas tonight
  - Emails went to the student who submitted the assignment
  - If you did not receive an email, it is because your code didn’t compile. Verify that you submitted the correct files on canvas and contact me

MINUTIAE

- **HW2 out tonight**
  - Written assignment
  - Analysis and bigO
  - Should be simple, opportunity to get feedback on written problems before the midterm

MINUTIAE

- **P1 student feedback**
  - New project this quarter
  - Opening anonymous survey tonight
  - How long did you spend?
  - Which parts of the project were poorly explained?
  - What did you get out of the project?

MINUTIAE

- **Because of the number of problems with the project, I have decided to increase the number of late days to 4.**
  - If you’ve already completed this project, you can use it on a later date, but this gives you a little more leeway to complete this assignment.
  - Late days will be accurate tonight when your canvas grade is posted.

TODAY'S LECTURE

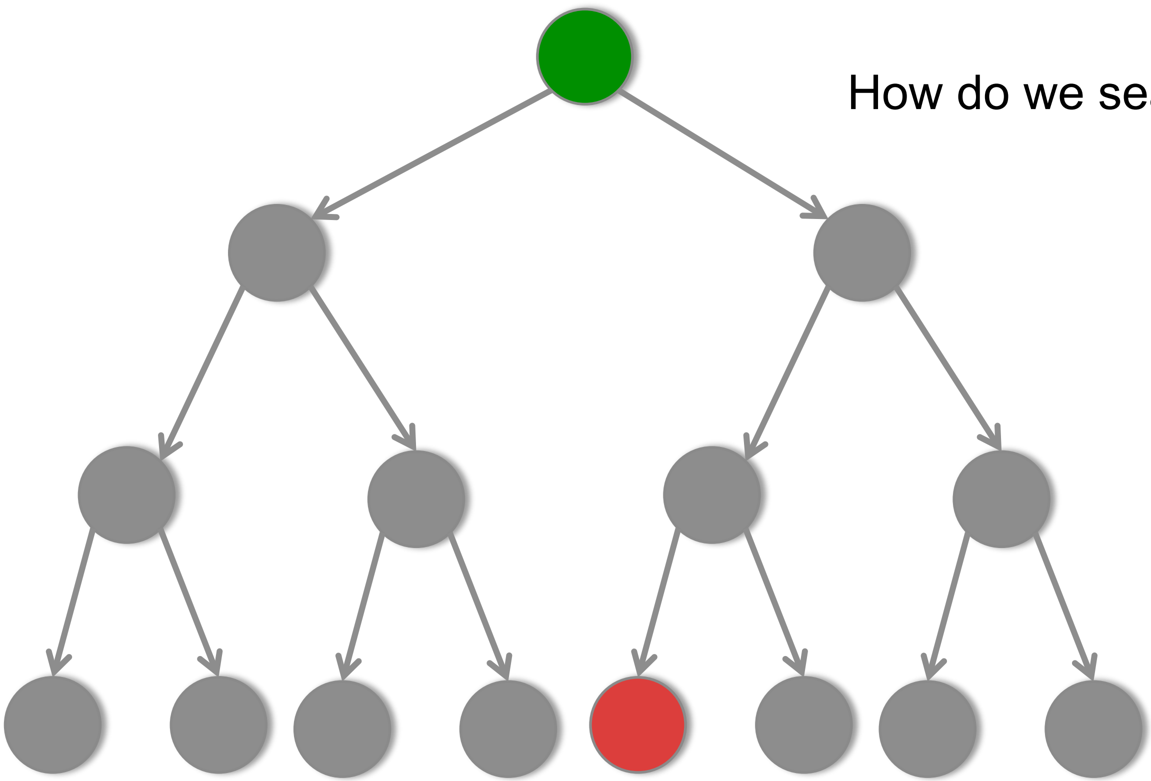
- **Traversal review**
  - DFS/BFS/Pre/In/Post order
- **Memory Analysis**
- **AVL Trees and how to balance**

DEPTH FIRST SEARCH

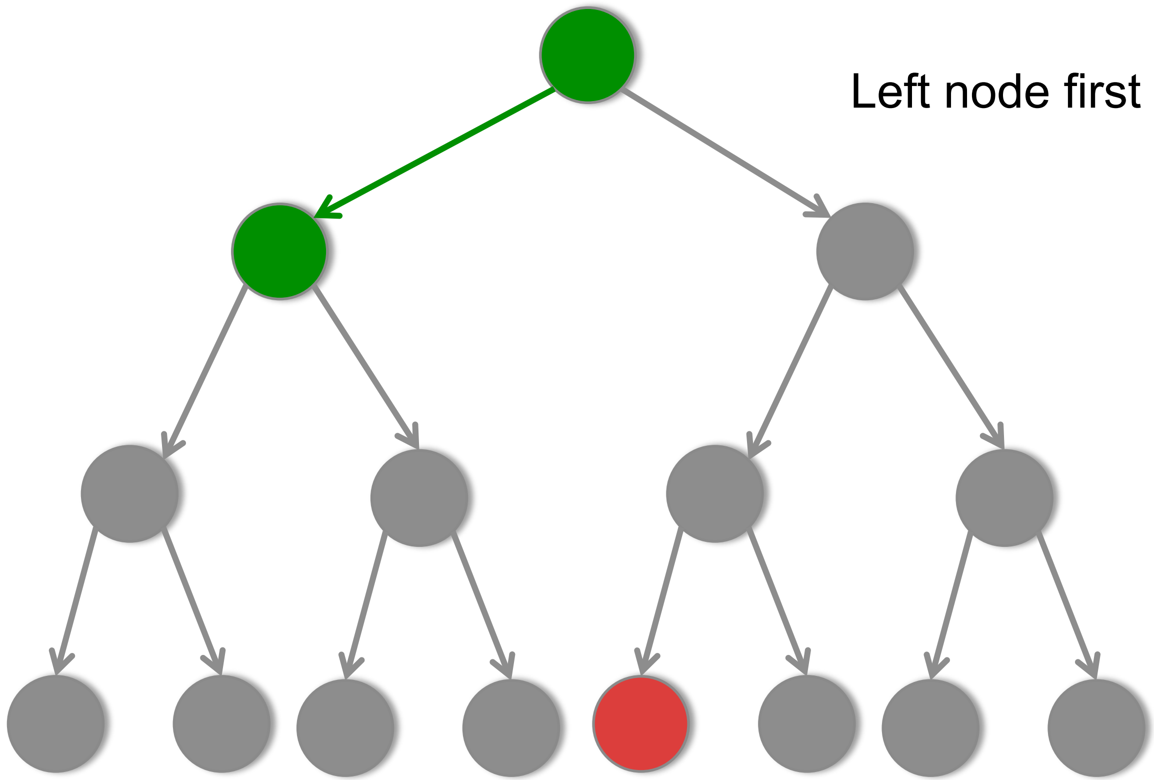
- All tree traversals start at the root
- As the name implies, traverse down the tree first.
- Left or right does not explicitly matter, but left usually comes first.

DEPTH FIRST SEARCH

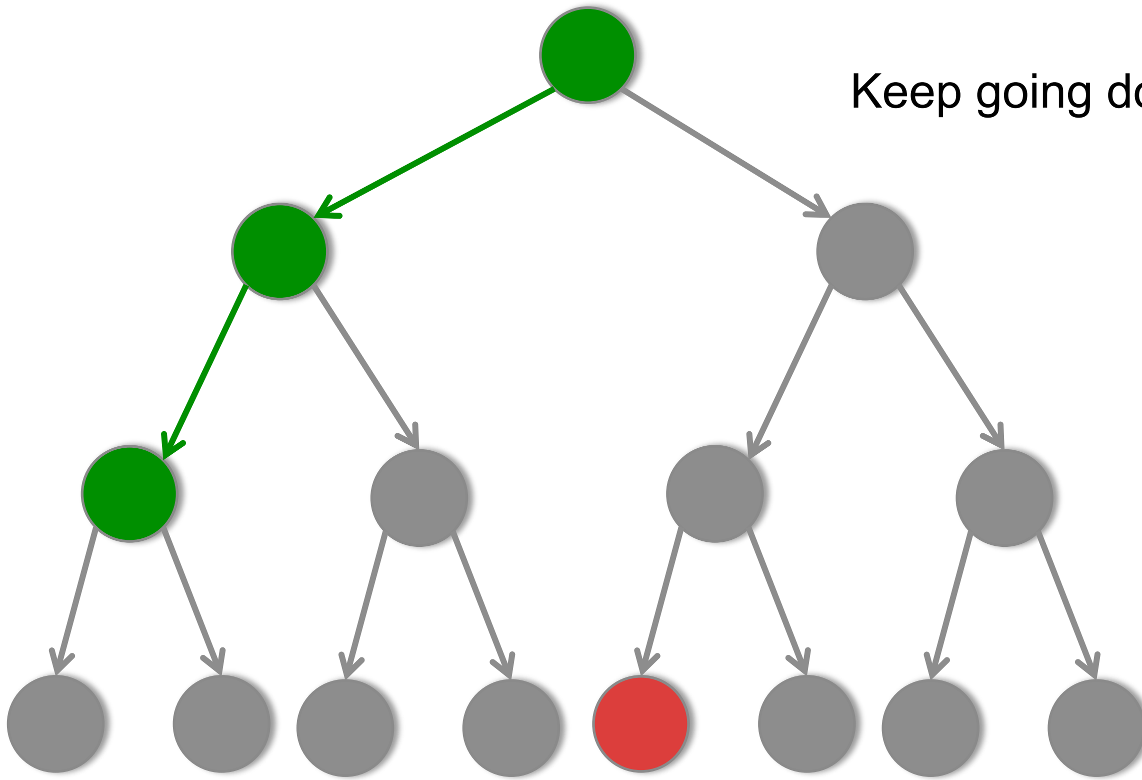
How do we search this tree?



DEPTH FIRST SEARCH



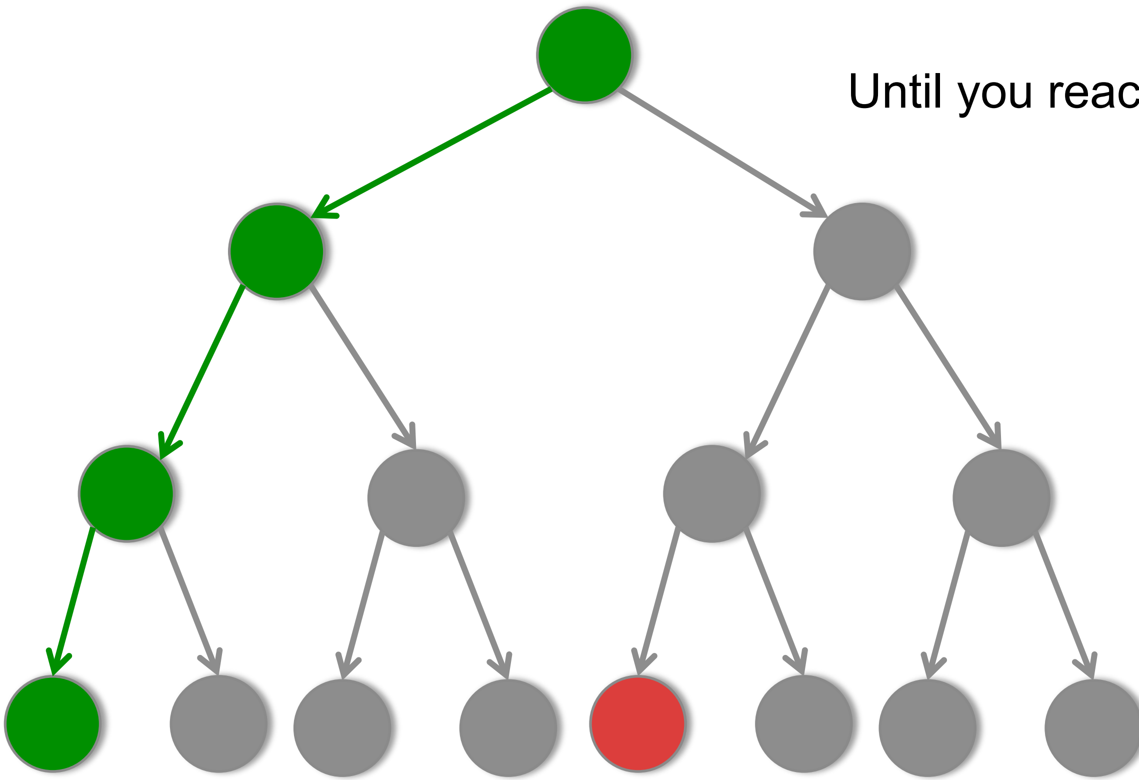
DEPTH FIRST SEARCH



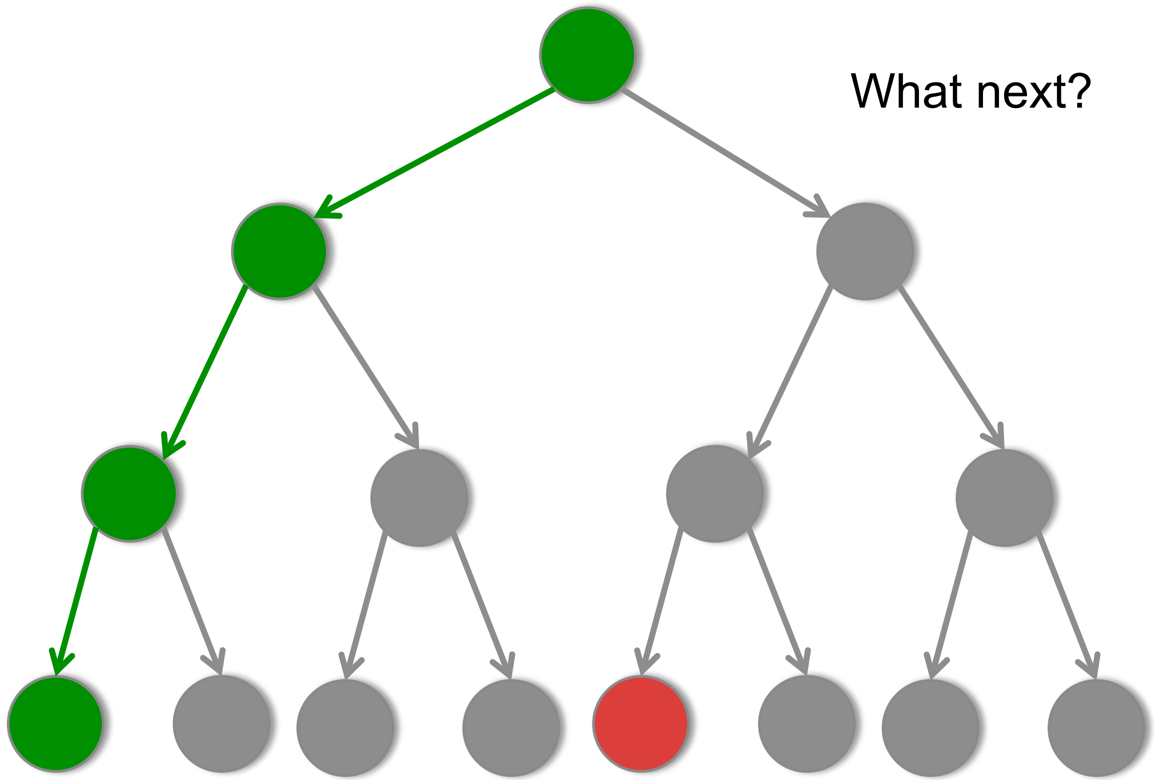
Keep going down the left nodes

DEPTH FIRST SEARCH

Until you reach the bottom

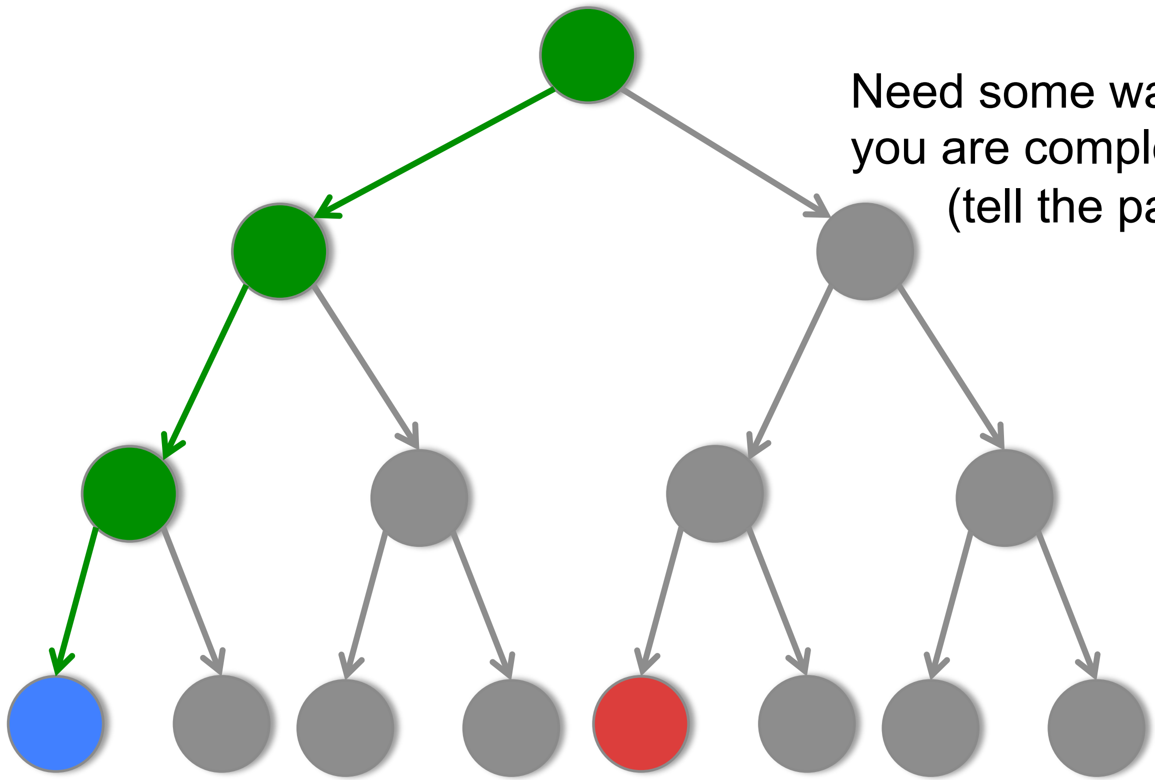


DEPTH FIRST SEARCH



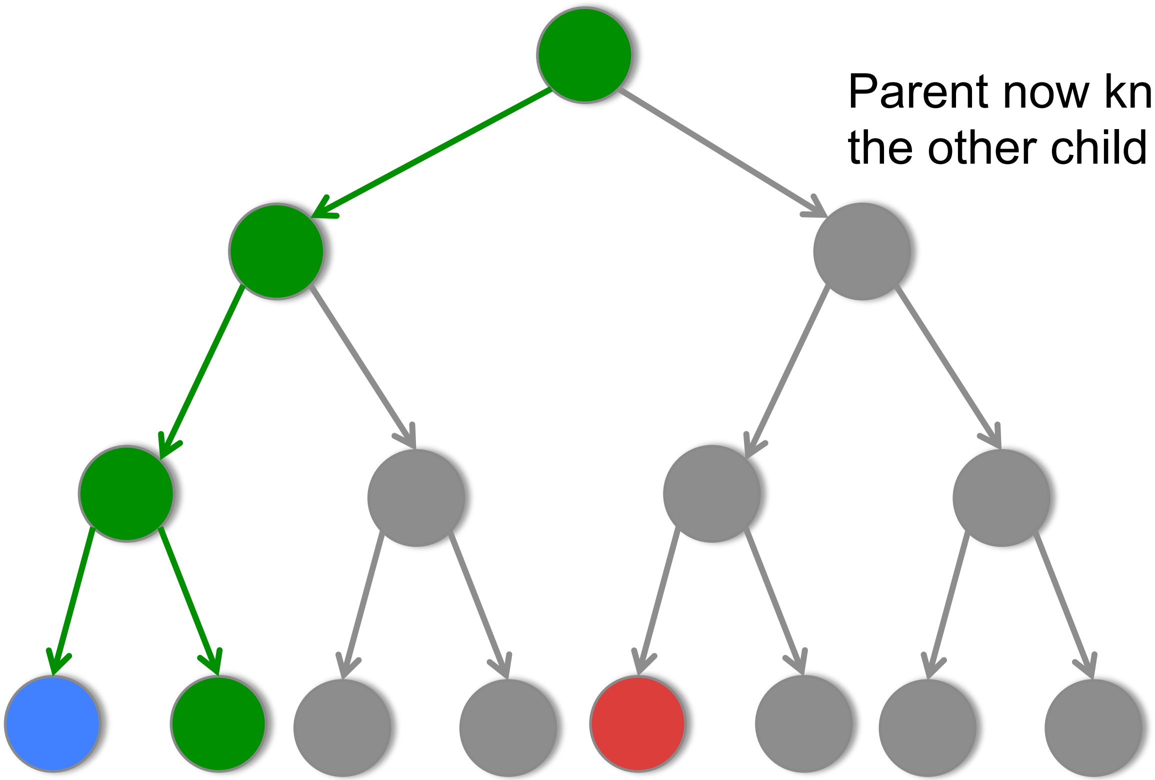
What next?

DEPTH FIRST SEARCH



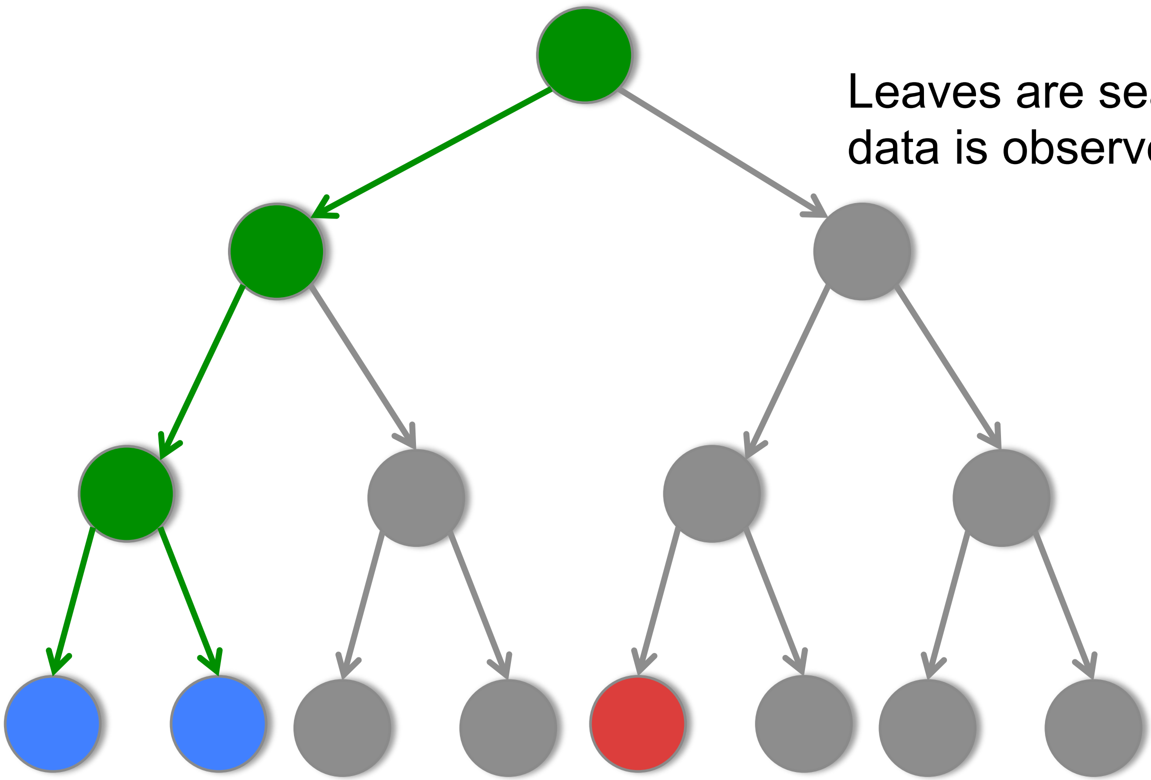
Need some way to indicate that you are completely searched
(tell the parent)

DEPTH FIRST SEARCH



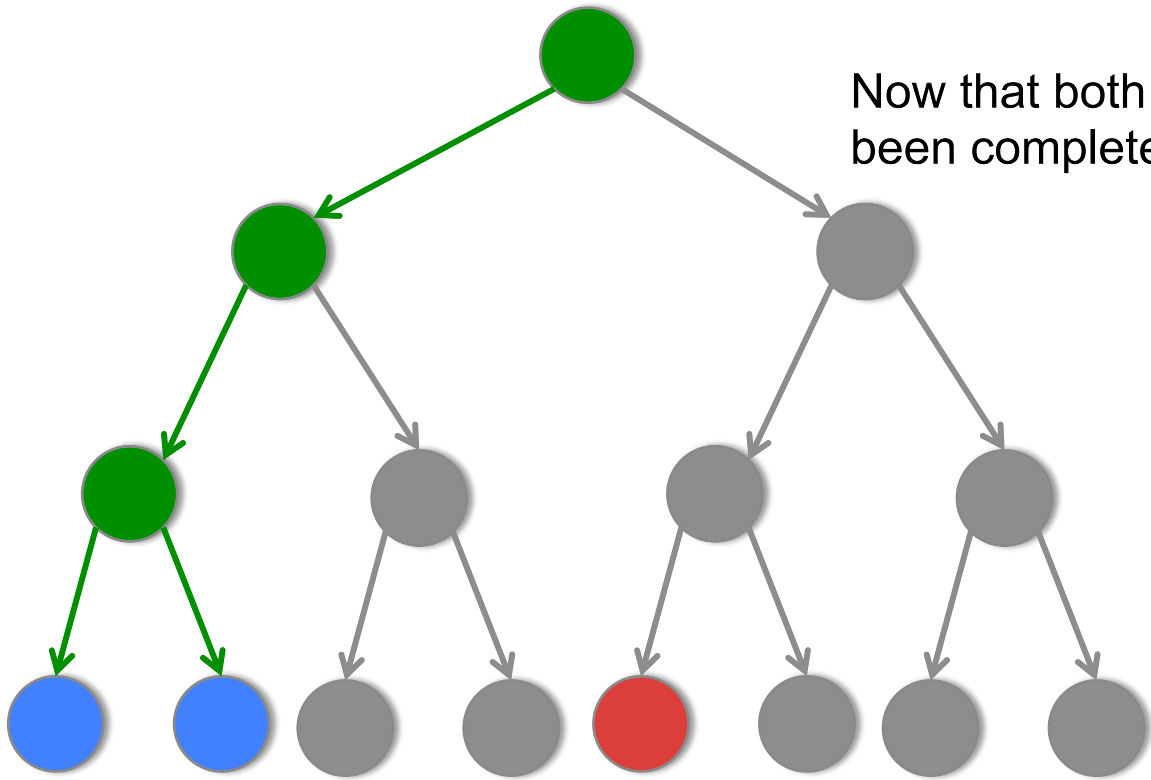
Parent now knows it is can search the other child

DEPTH FIRST SEARCH



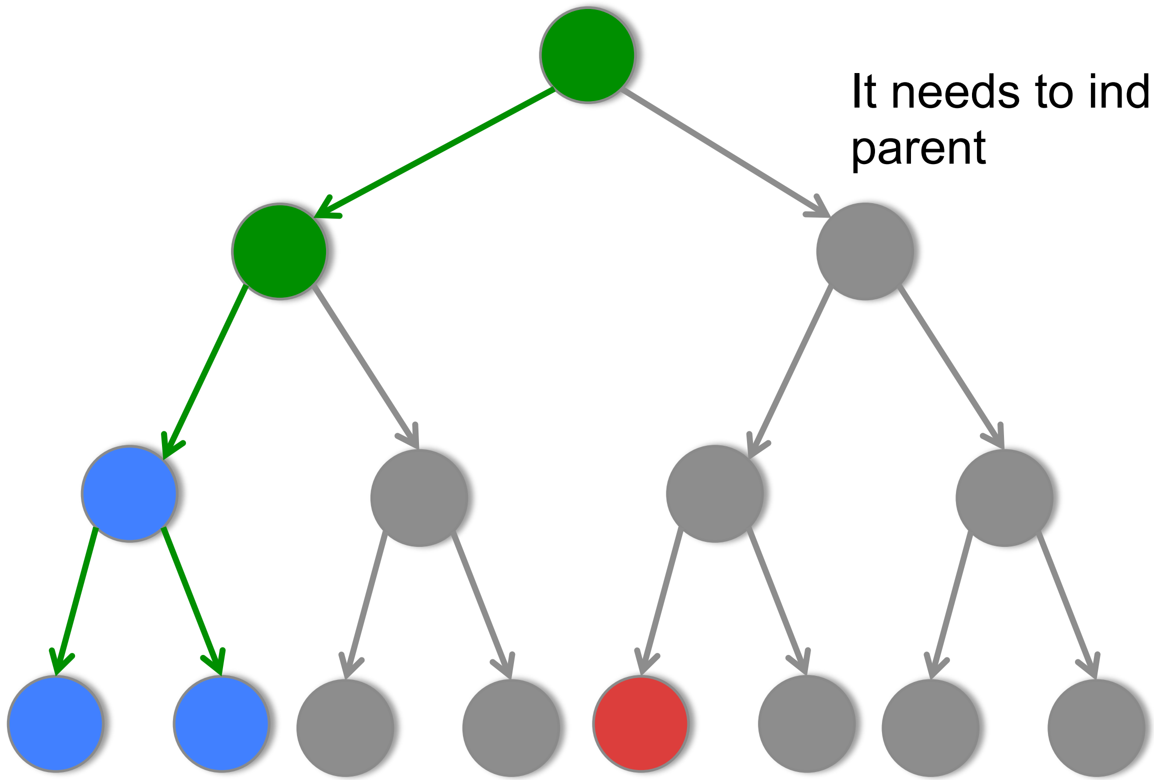
Leaves are searched when their data is observed

DEPTH FIRST SEARCH



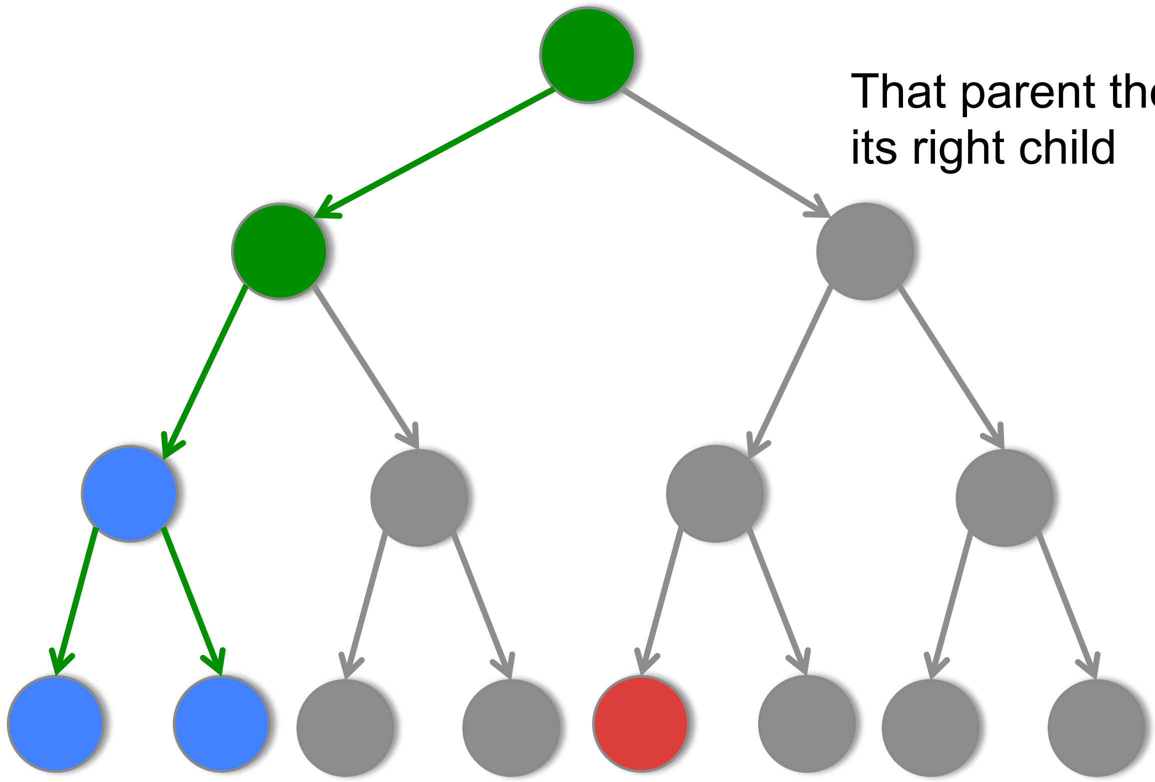
Now that both of its children have been completely searched

DEPTH FIRST SEARCH



It needs to indicate that to its parent

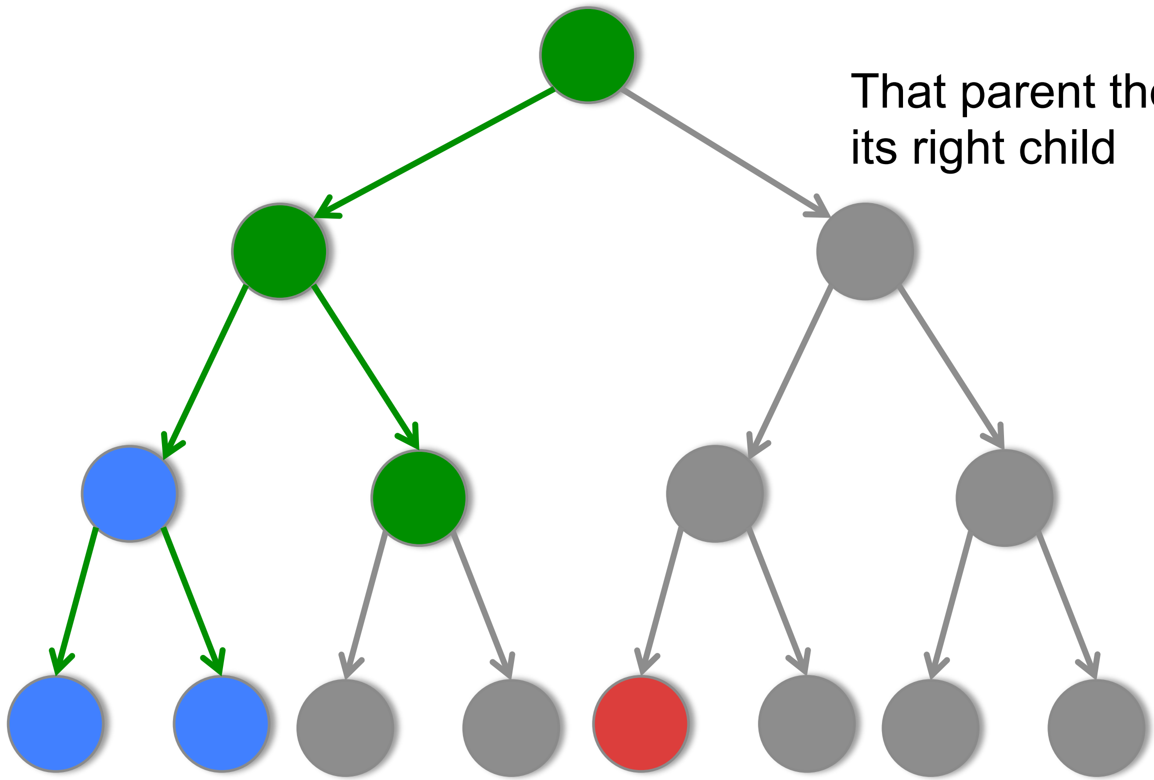
DEPTH FIRST SEARCH



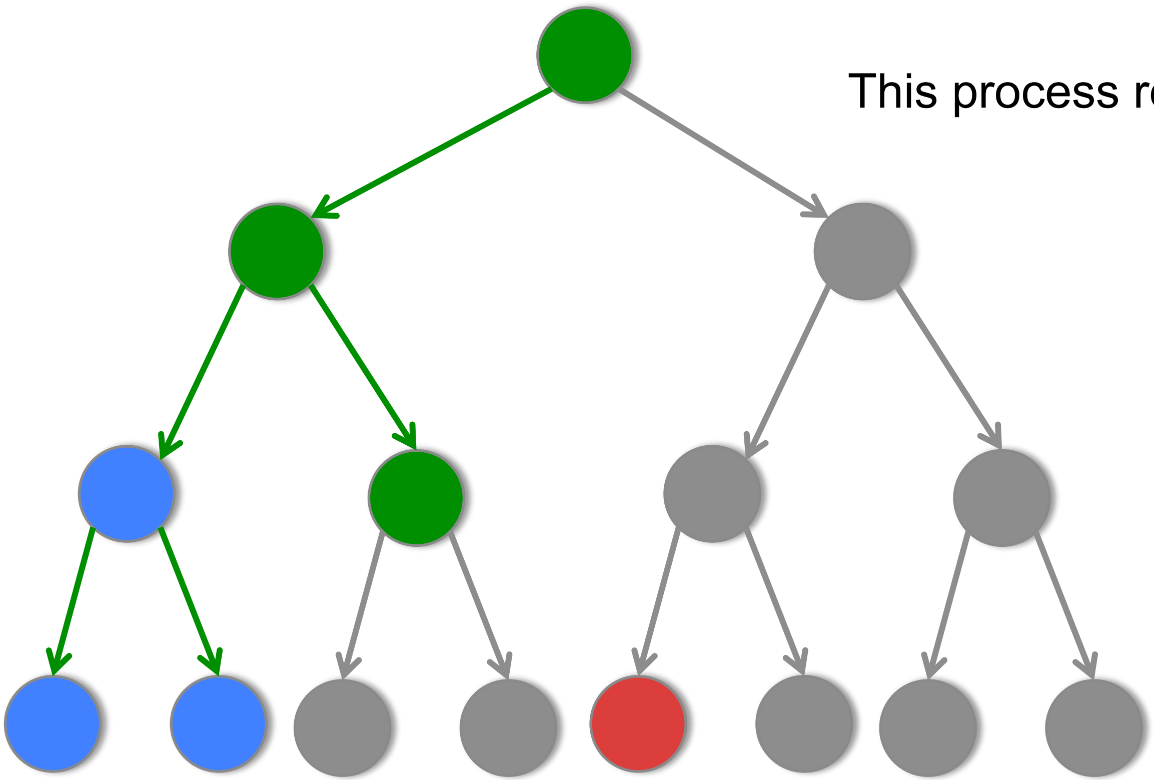
That parent then knows to search its right child

DEPTH FIRST SEARCH

That parent then knows to search its right child



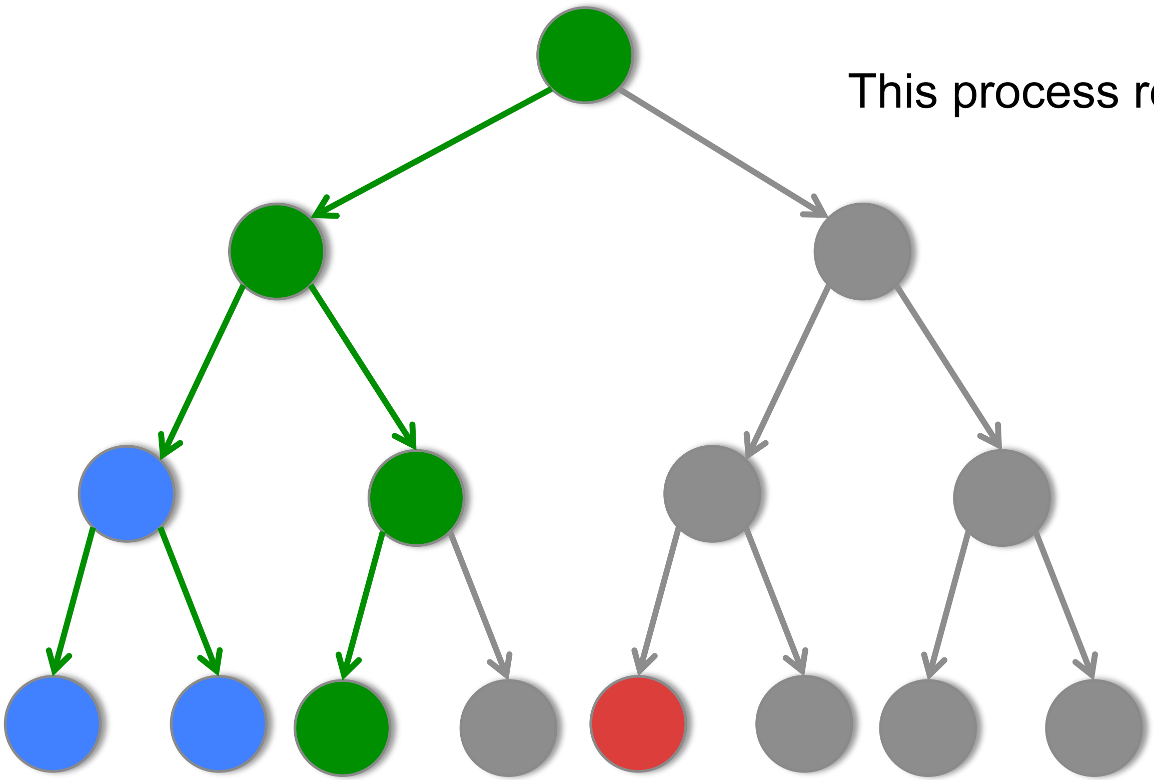
DEPTH FIRST SEARCH



This process repeats

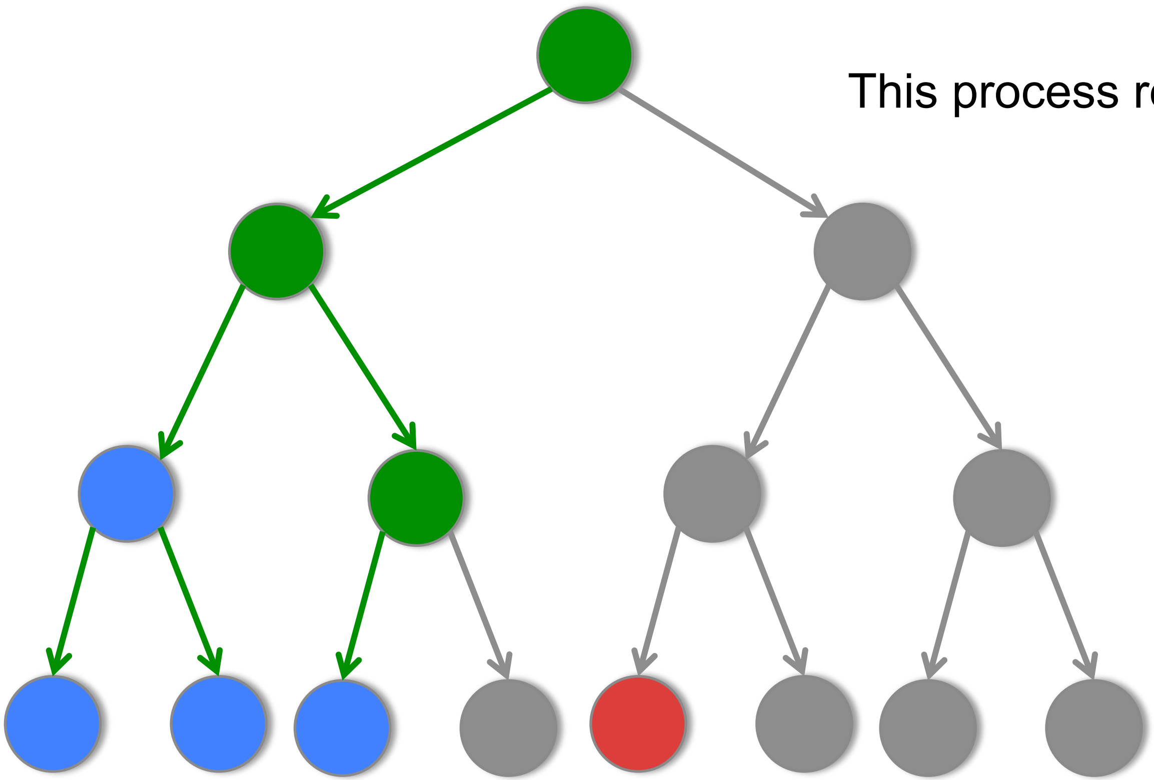
DEPTH FIRST SEARCH

This process repeats

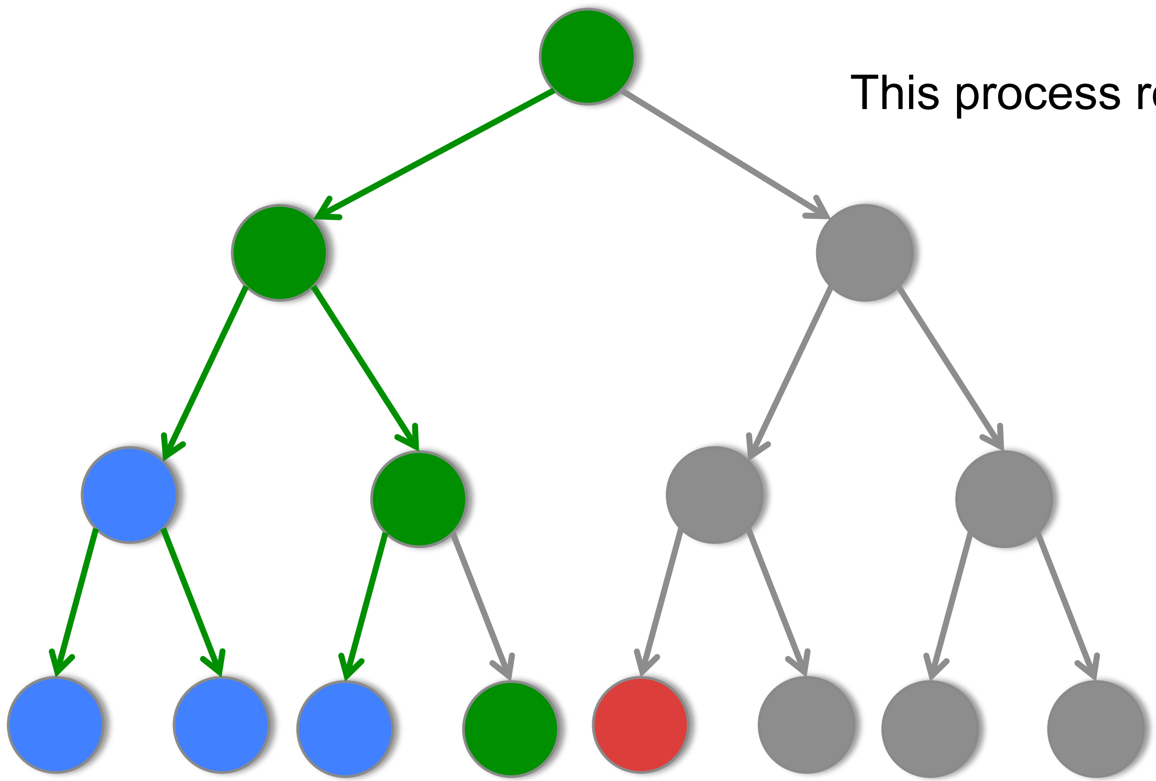


DEPTH FIRST SEARCH

This process repeats



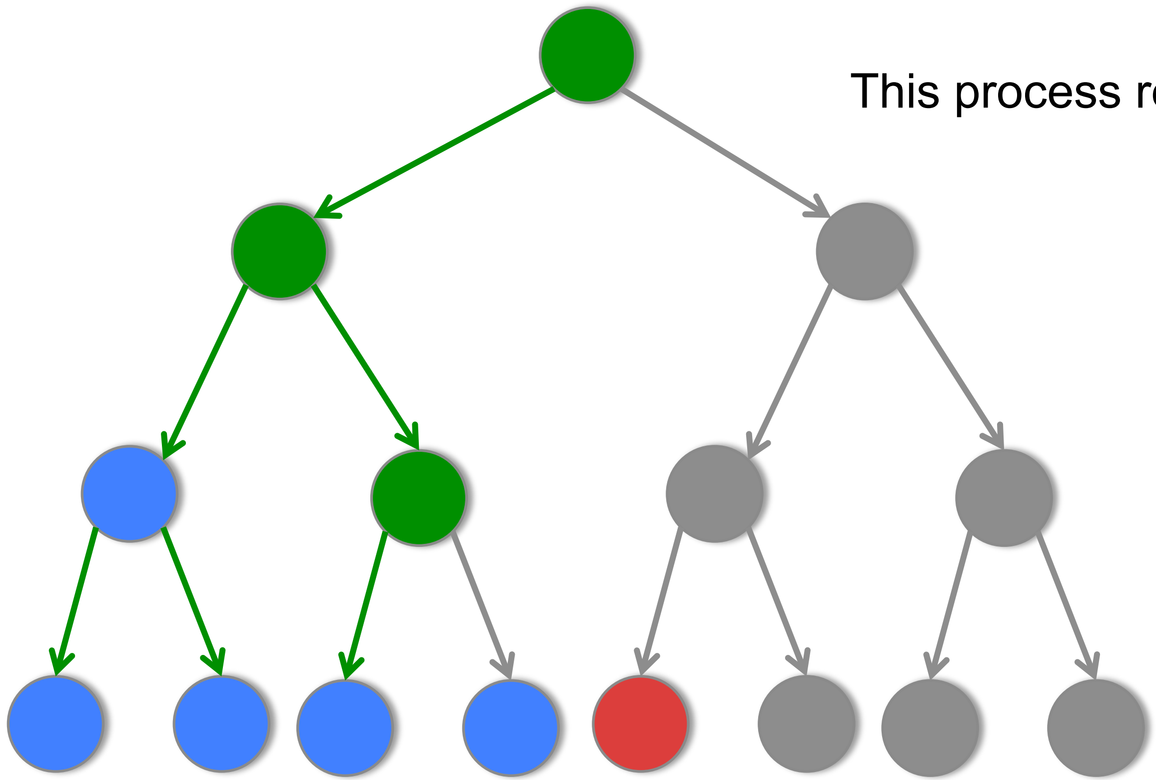
DEPTH FIRST SEARCH



This process repeats

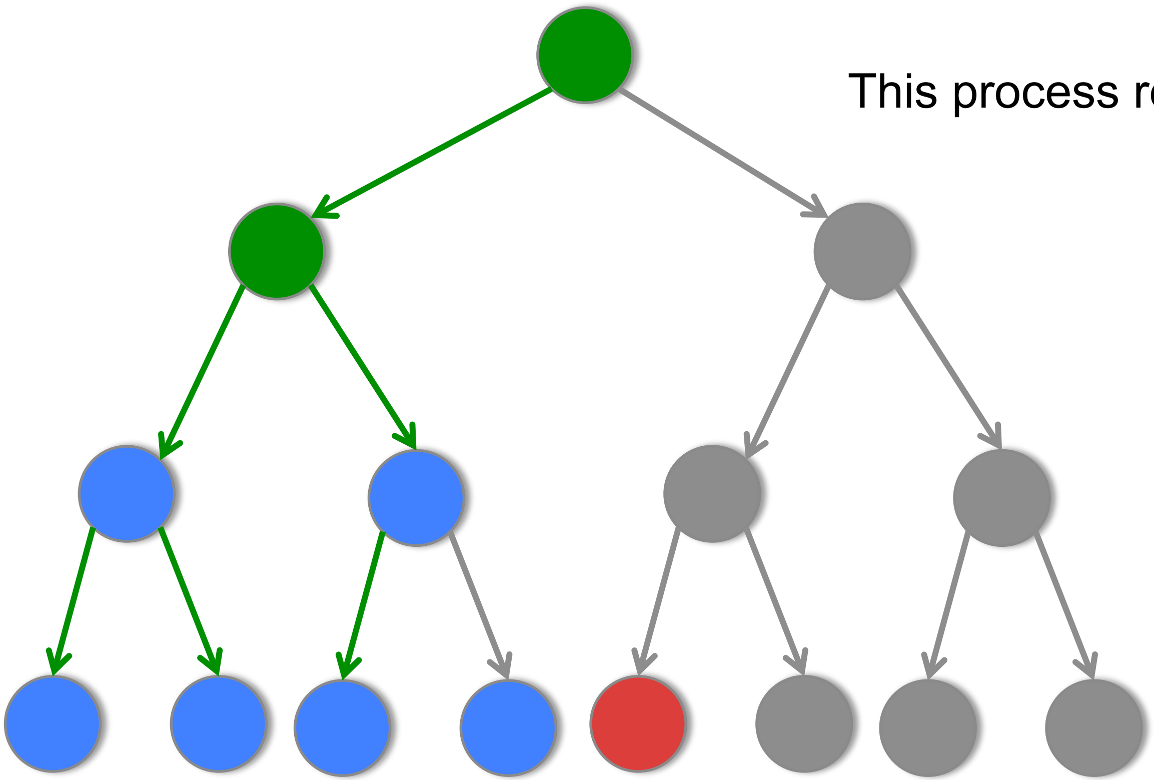
DEPTH FIRST SEARCH

This process repeats

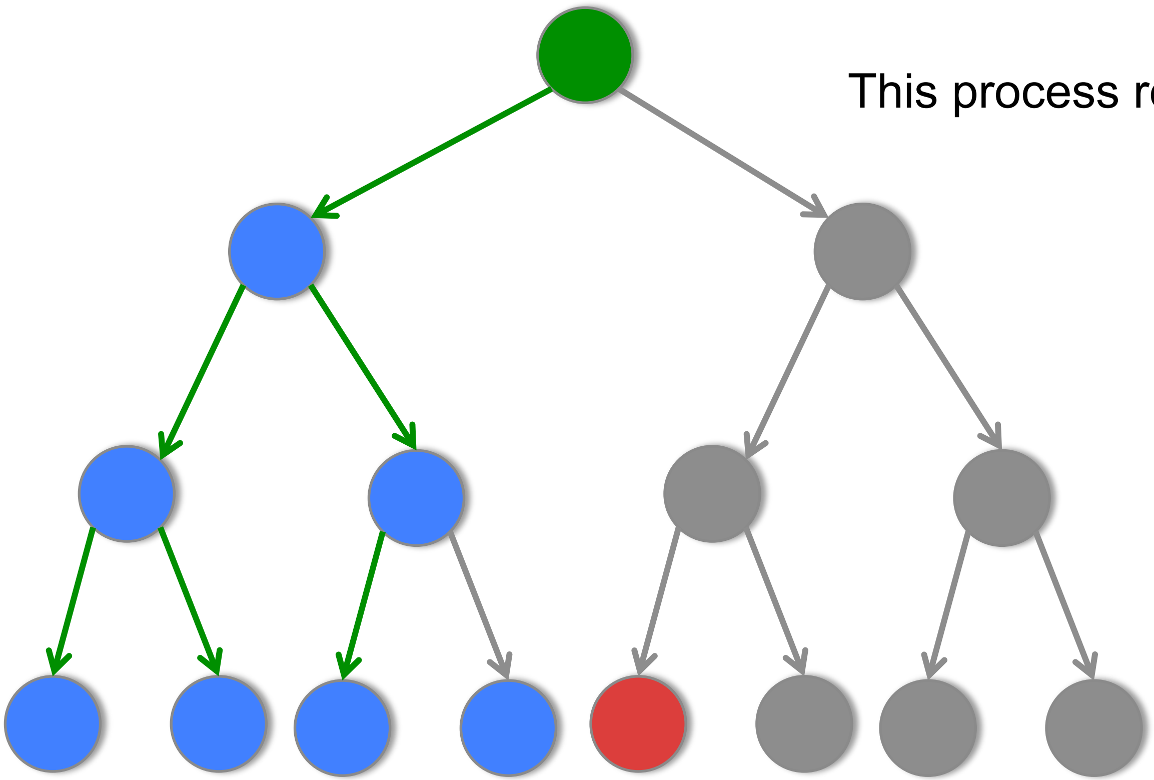


DEPTH FIRST SEARCH

This process repeats

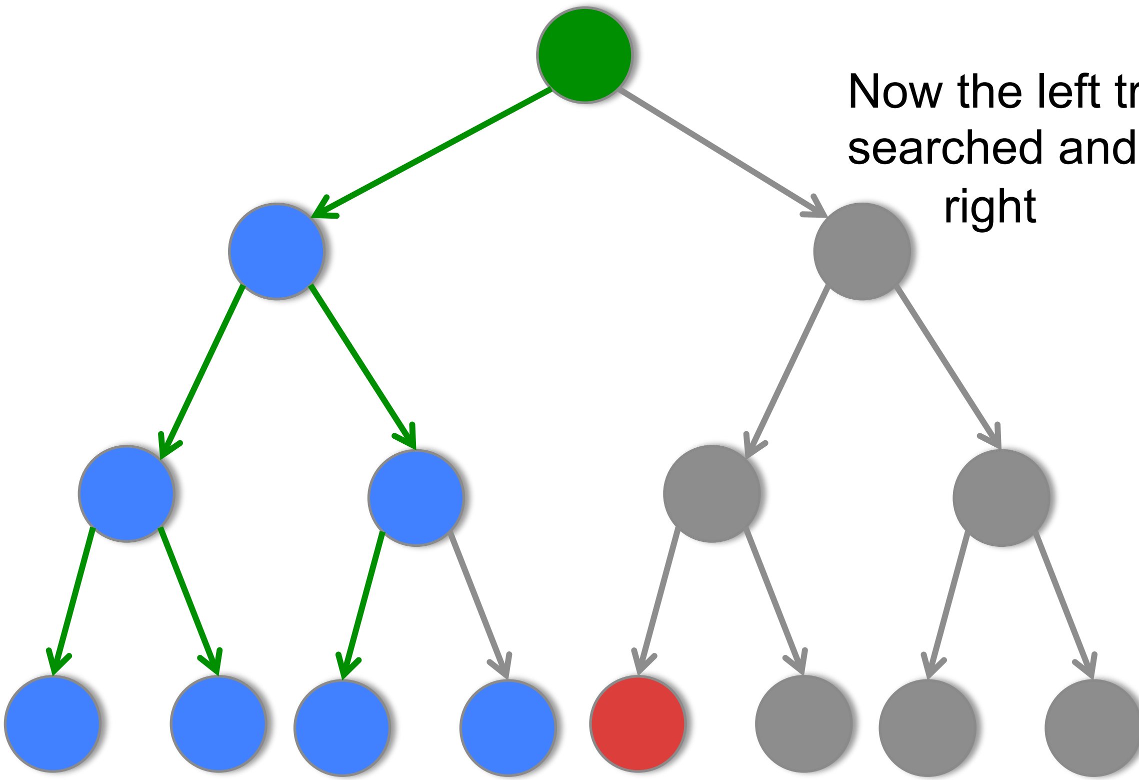


DEPTH FIRST SEARCH



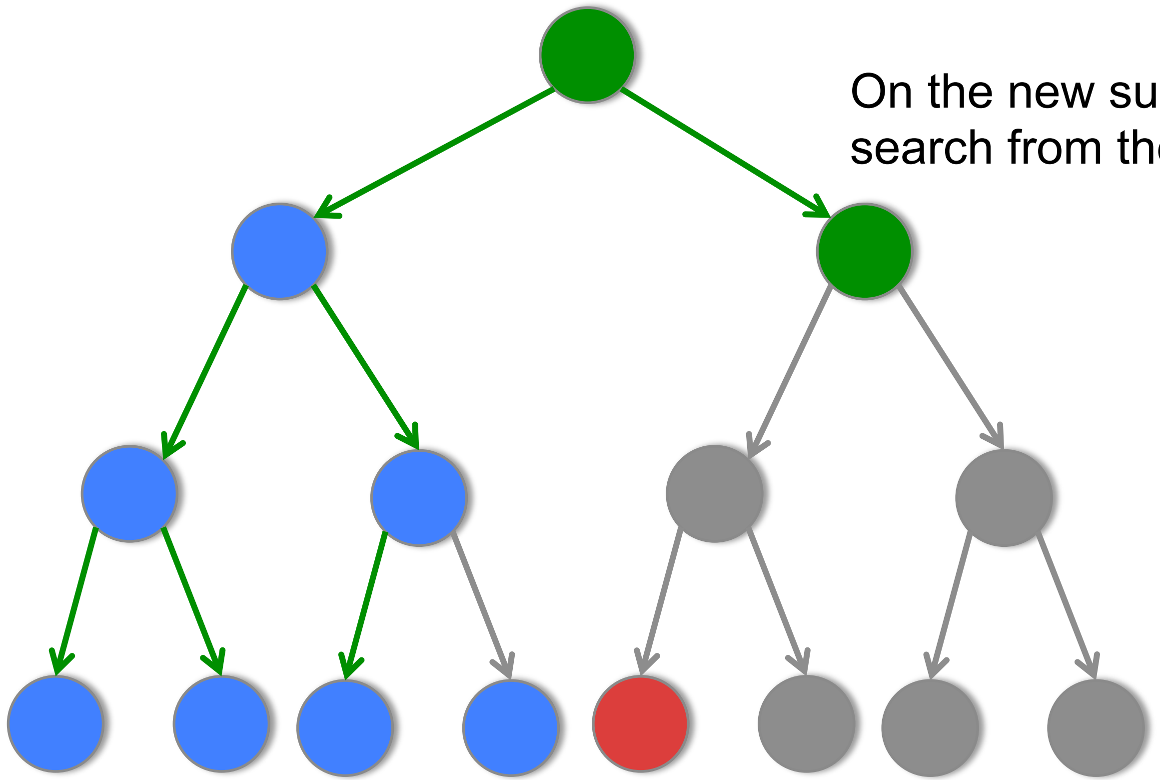
This process repeats

DEPTH FIRST SEARCH



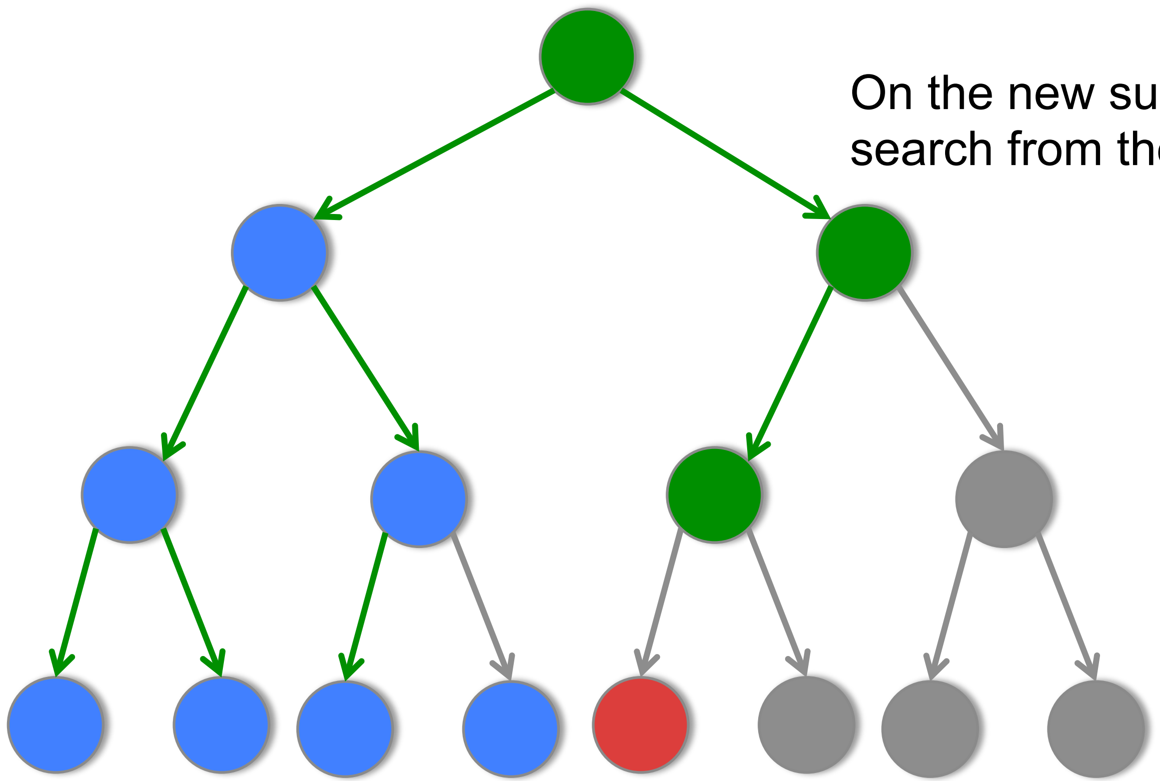
Now the left tree is completely searched and we can search the right

DEPTH FIRST SEARCH



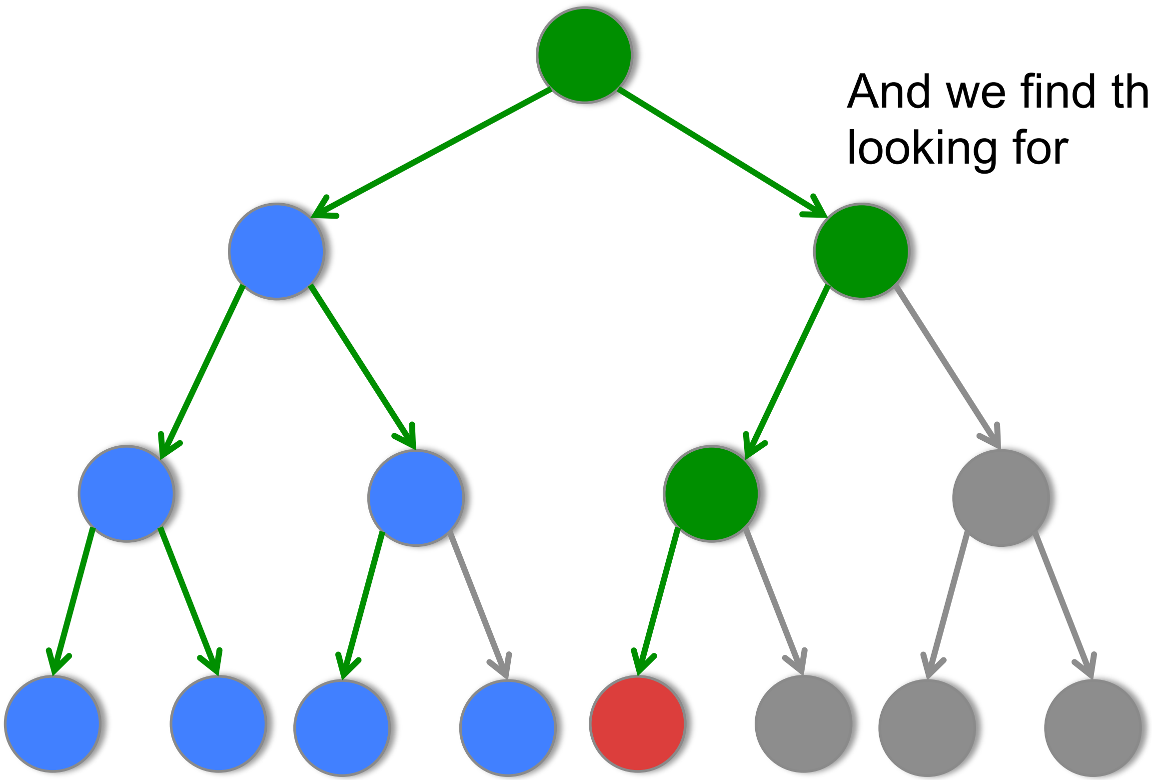
On the new subtree, we begin search from the left

DEPTH FIRST SEARCH



On the new subtree, we begin search from the left

DEPTH FIRST SEARCH



And we find the object we're looking for

DEPTH FIRST SEARCH

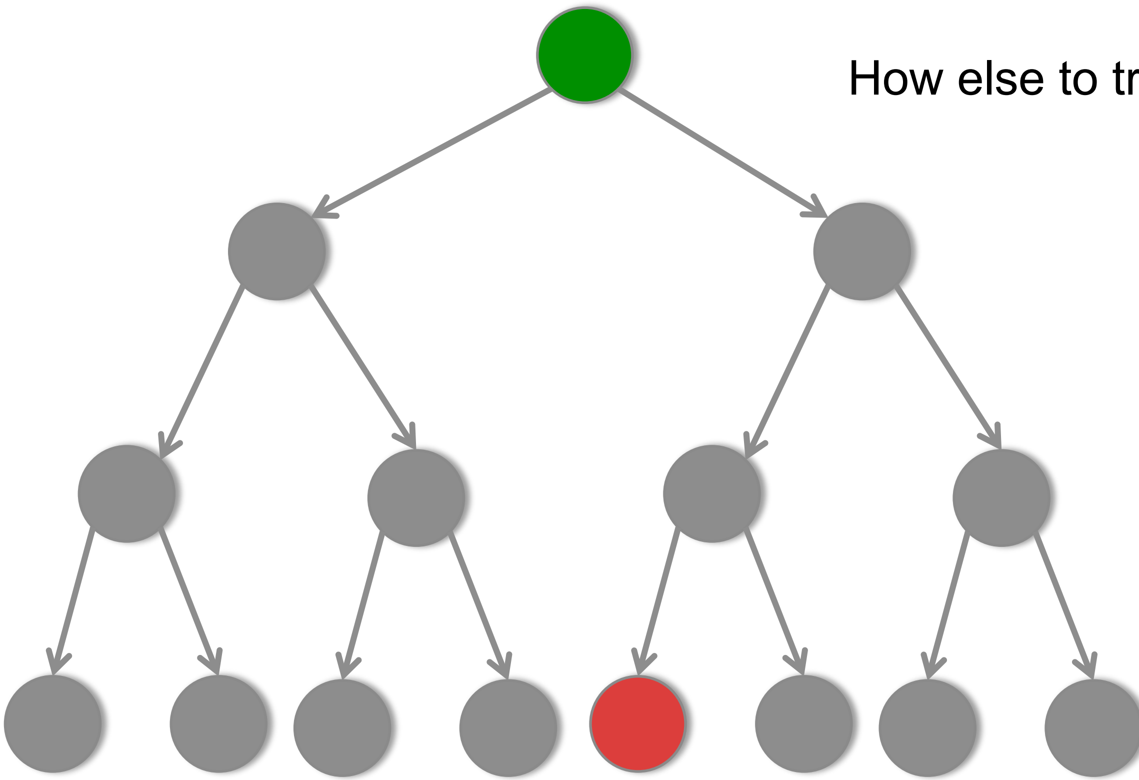
- **How does this work in application?**
  - For each node, it searches its left subtree entirely and then moves to the right tree
  - Here search works by breaking the problem down into sub-problems
  - This is a good indication that we use recursion

DEPTH FIRST SEARCH

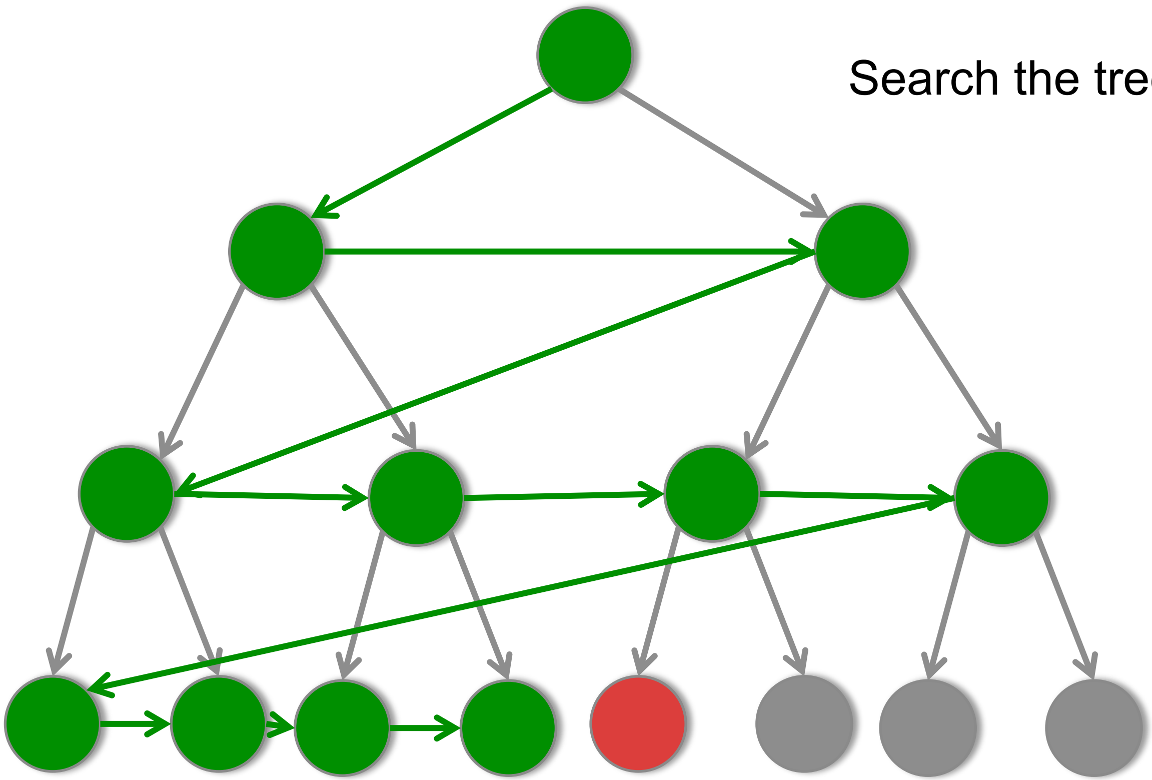
- Treat each subtree as a subproblem and solve recursively.
- Will go to maximum depth first.
- When the node is found, the result will return up the stack
- What might be a different approach?

ALTERNATE APPROACH

How else to traverse?



ALTERNATE APPROACH



Search the tree from top to bottom

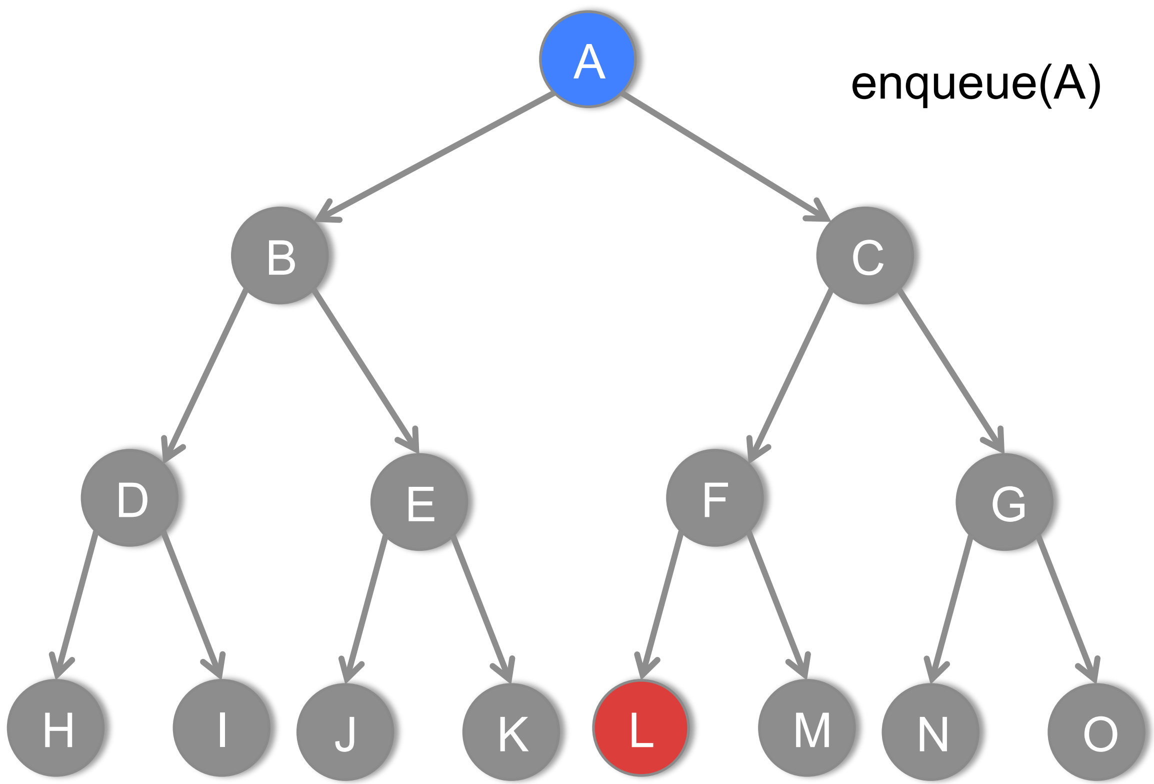
BREADTH FIRST SEARCH

- **Consider the approach**
  - Start with the root
  - Search all nodes of depth 1
  - Search all nodes of depth 2
  - …
  - *How do we get this ordering?*

BREADTH FIRST SEARCH

- **What if we use a Queue?**
  - Enqueue the root
  - Then what?

BREADTH FIRST SEARCH



Queue:

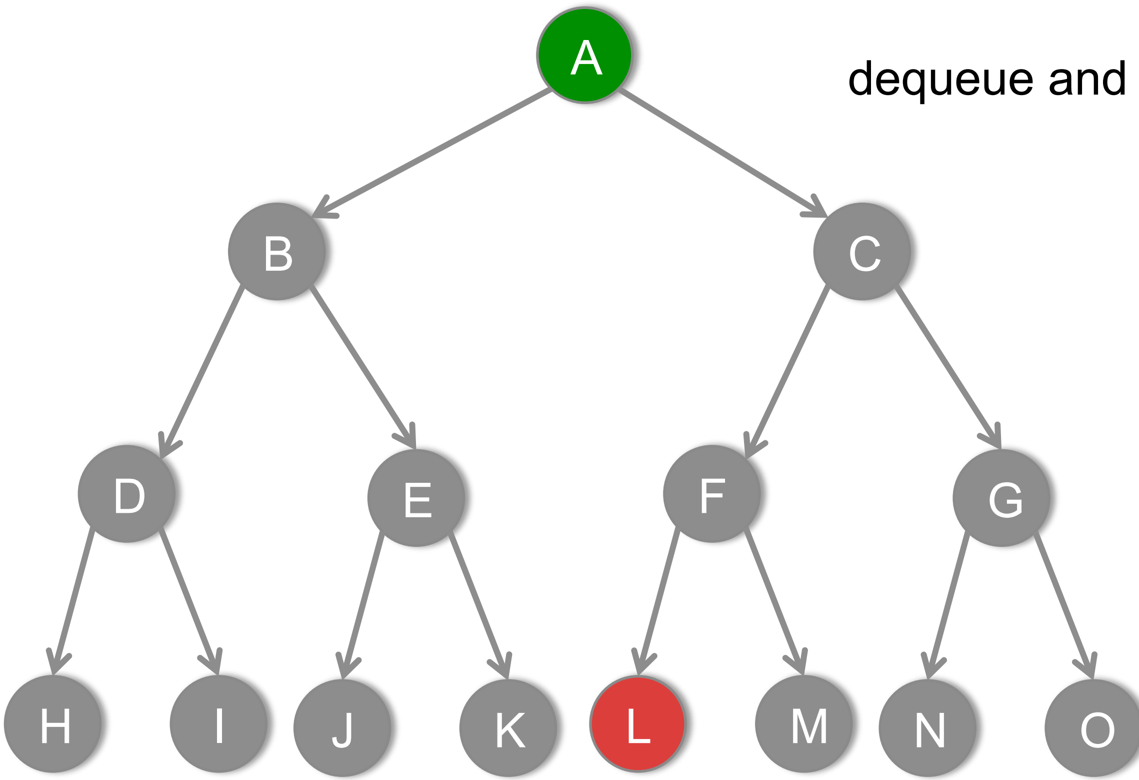
BREADTH FIRST SEARCH

- **What if we use a Queue?**

```
enqueue the root

while the queue has elements:
      dequeue the node
      if it matches our search string
            return true
      if it doesn't,
            enqueue its non-null children
return false;
```

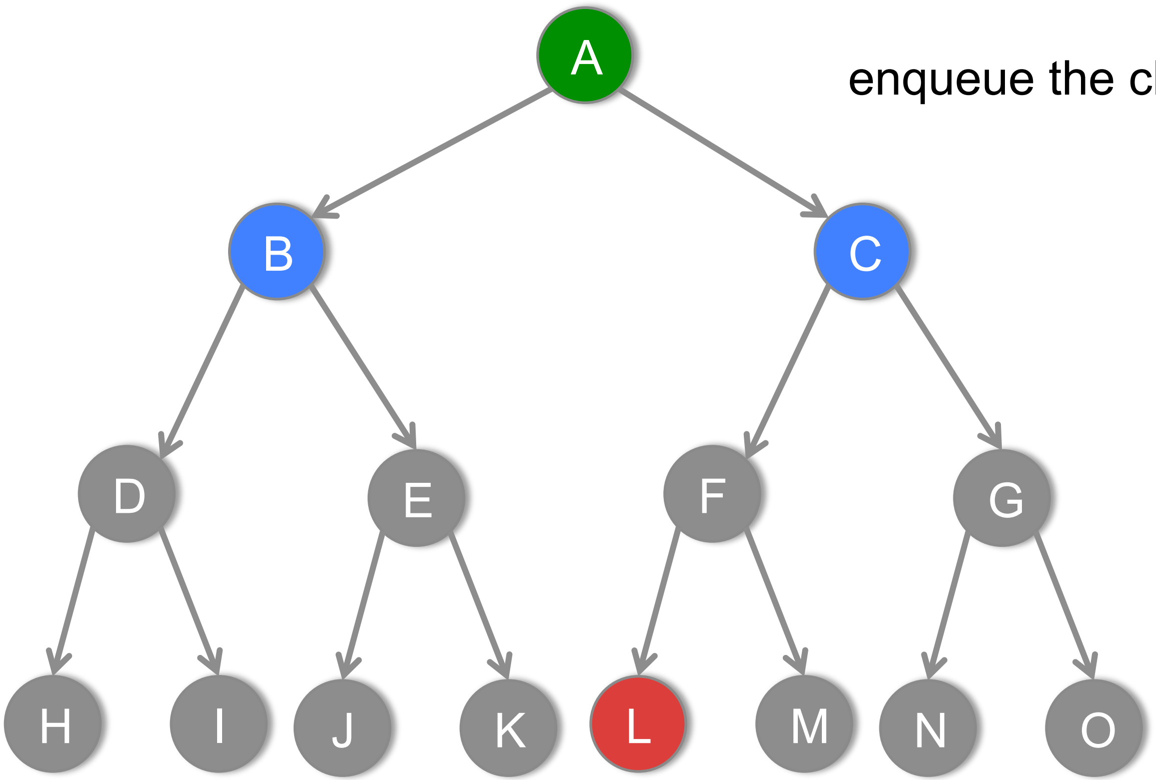
BREADTH FIRST SEARCH



dequeue and check the node

Queue:

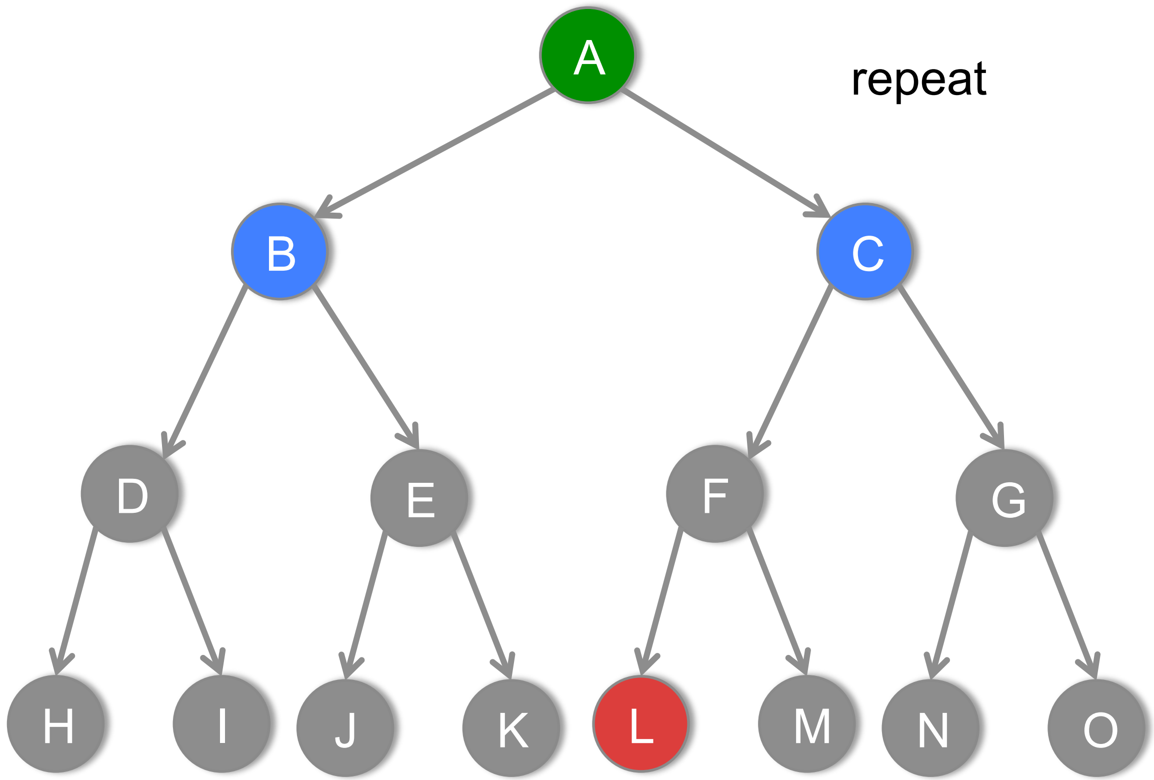
BREADTH FIRST SEARCH



enqueue the children

Queue: B | C |

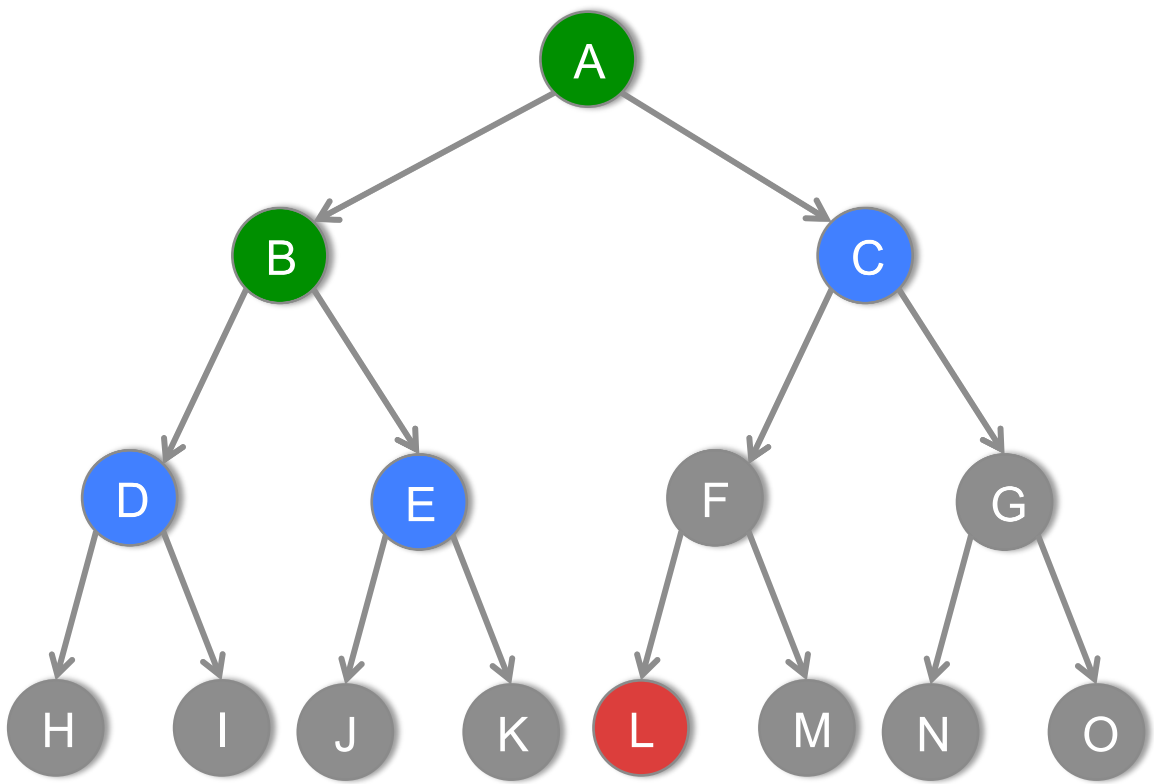
BREADTH FIRST SEARCH



repeat

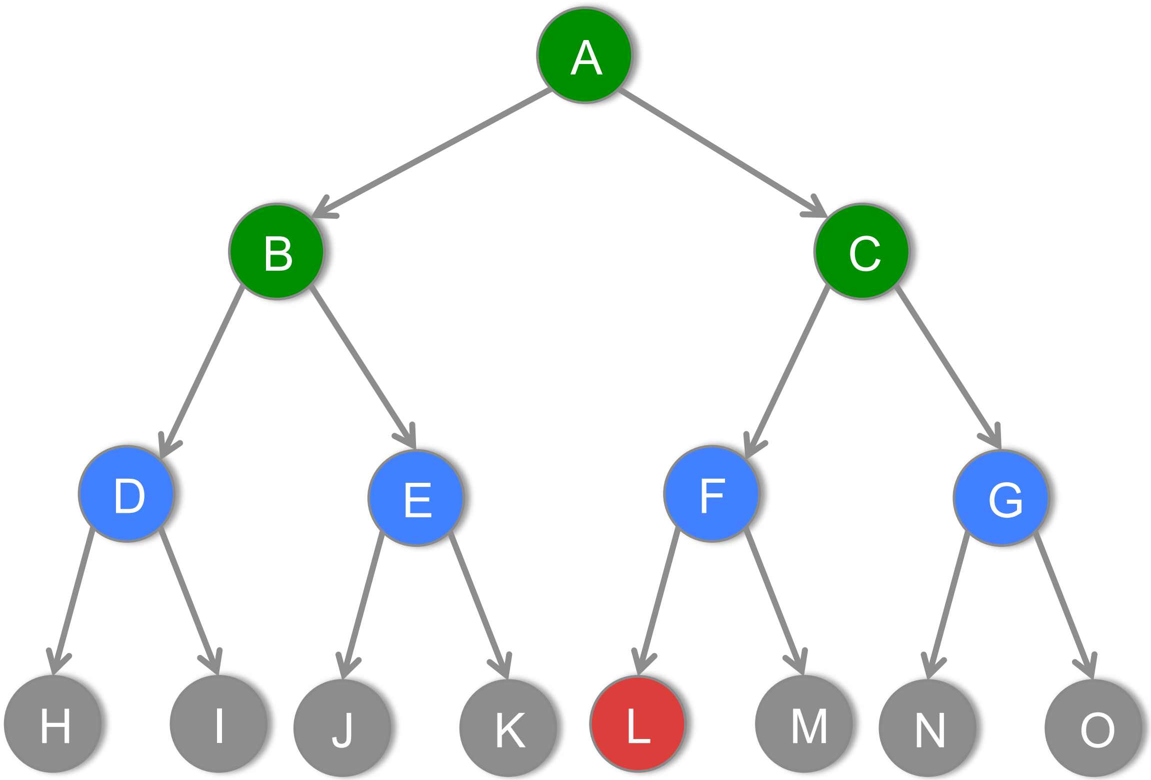
Queue: | B | C | |

BREADTH FIRST SEARCH



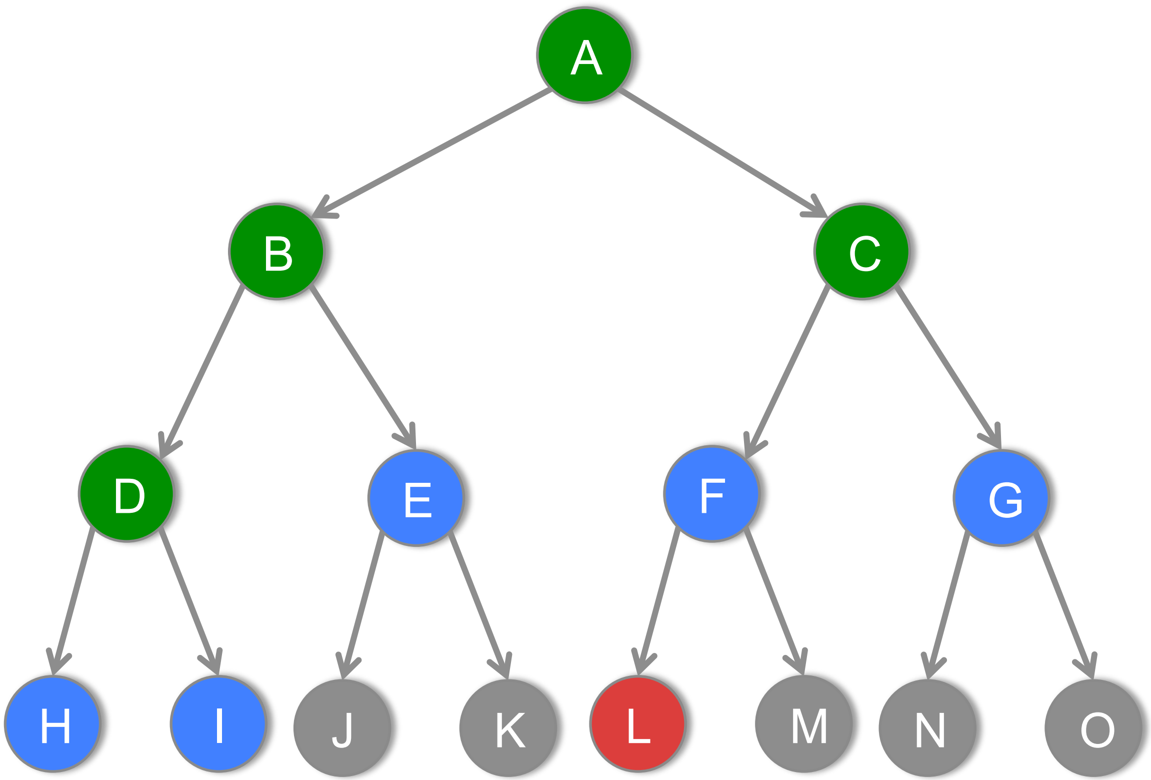
Queue: C | D | E |

BREADTH FIRST SEARCH



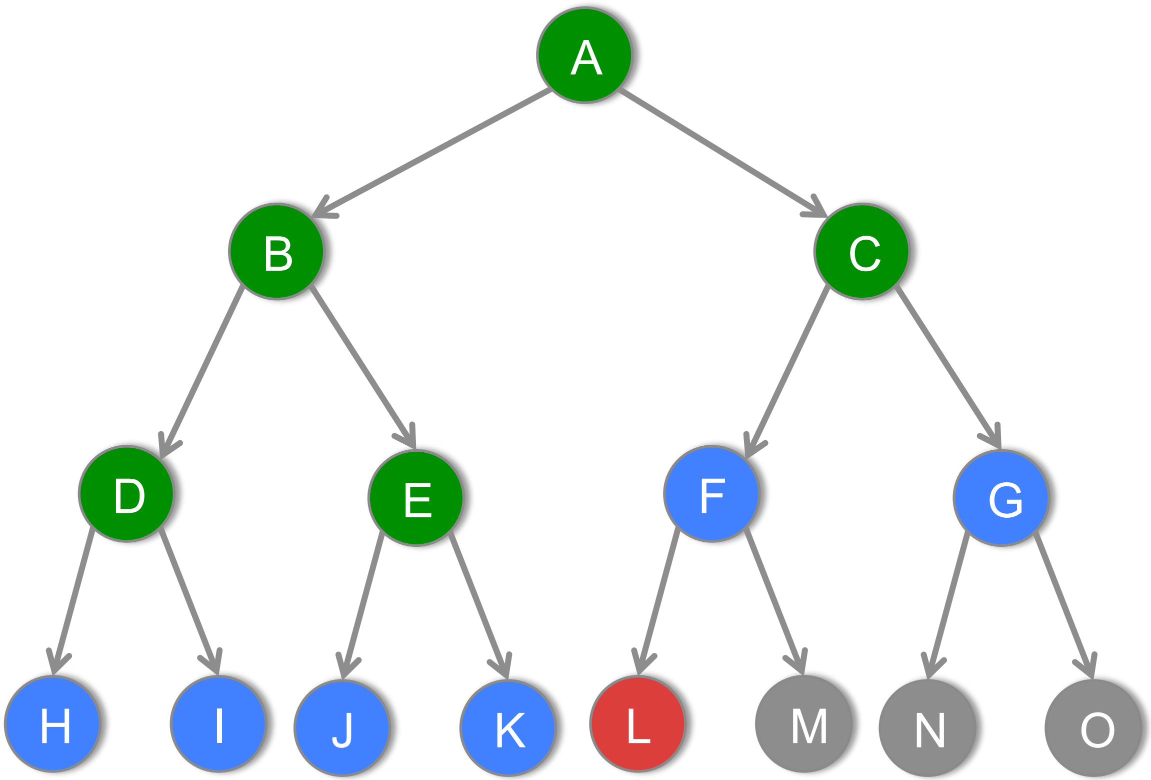
Queue: D | E | F | G |

BREADTH FIRST SEARCH



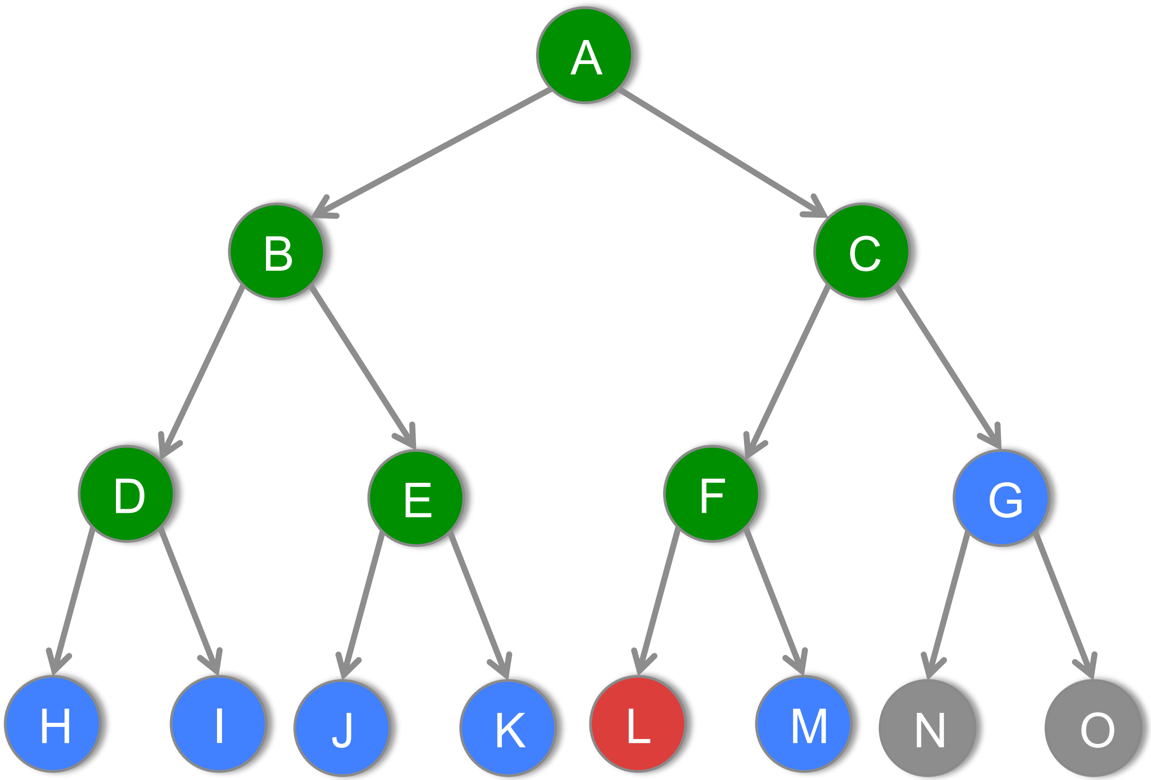
Queue: | E | F | G | H | I | |
|---|

BREADTH FIRST SEARCH



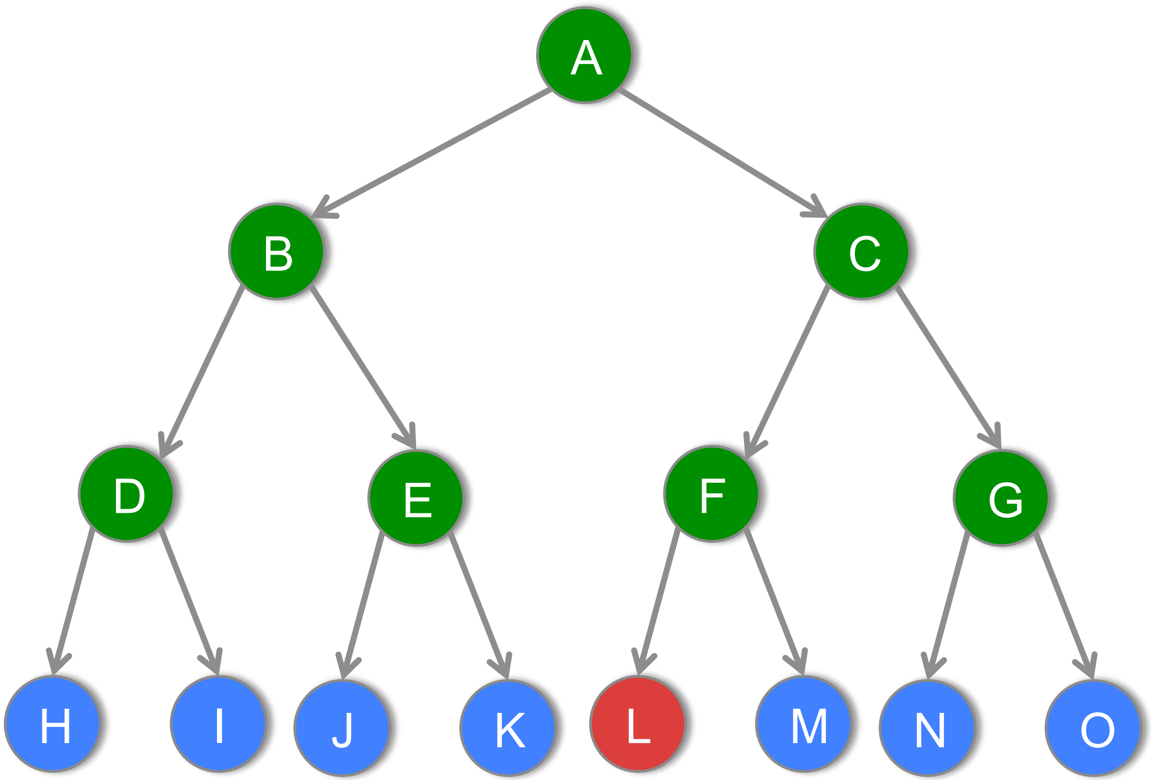
Queue: F | G | H | I | J | K |

BREADTH FIRST SEARCH



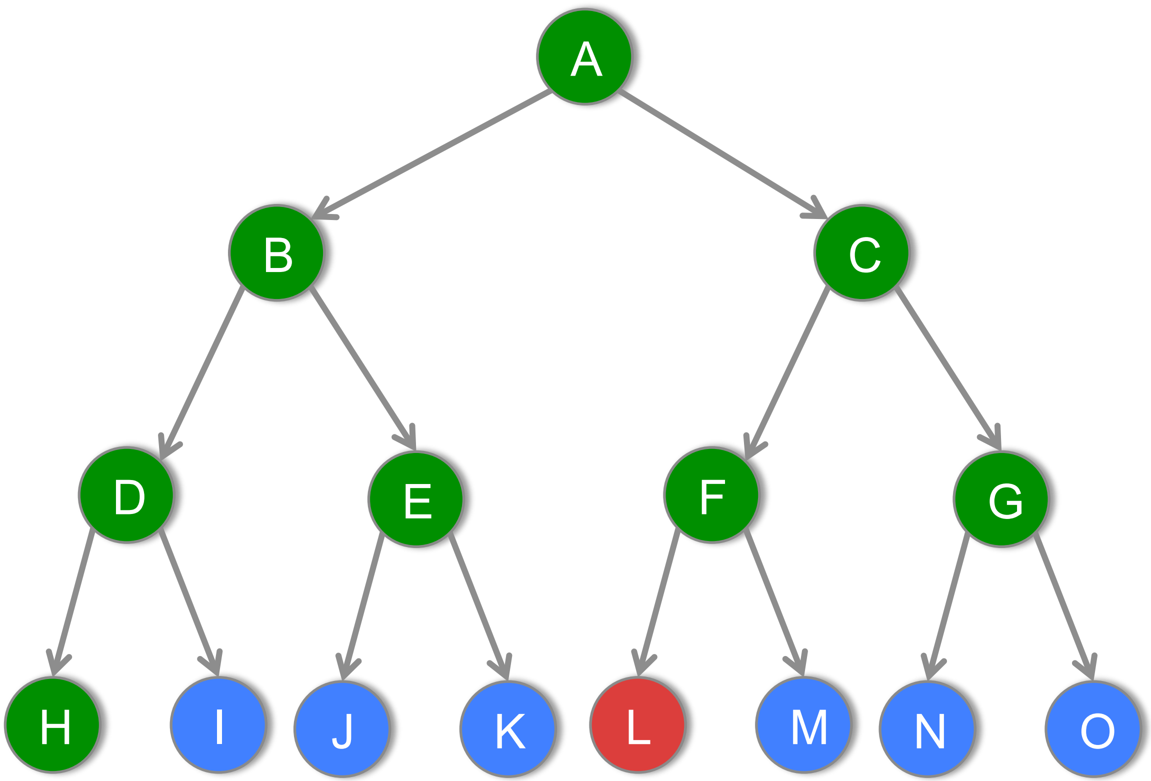
Queue: G | H | I | J | K | L | M

BREADTH FIRST SEARCH



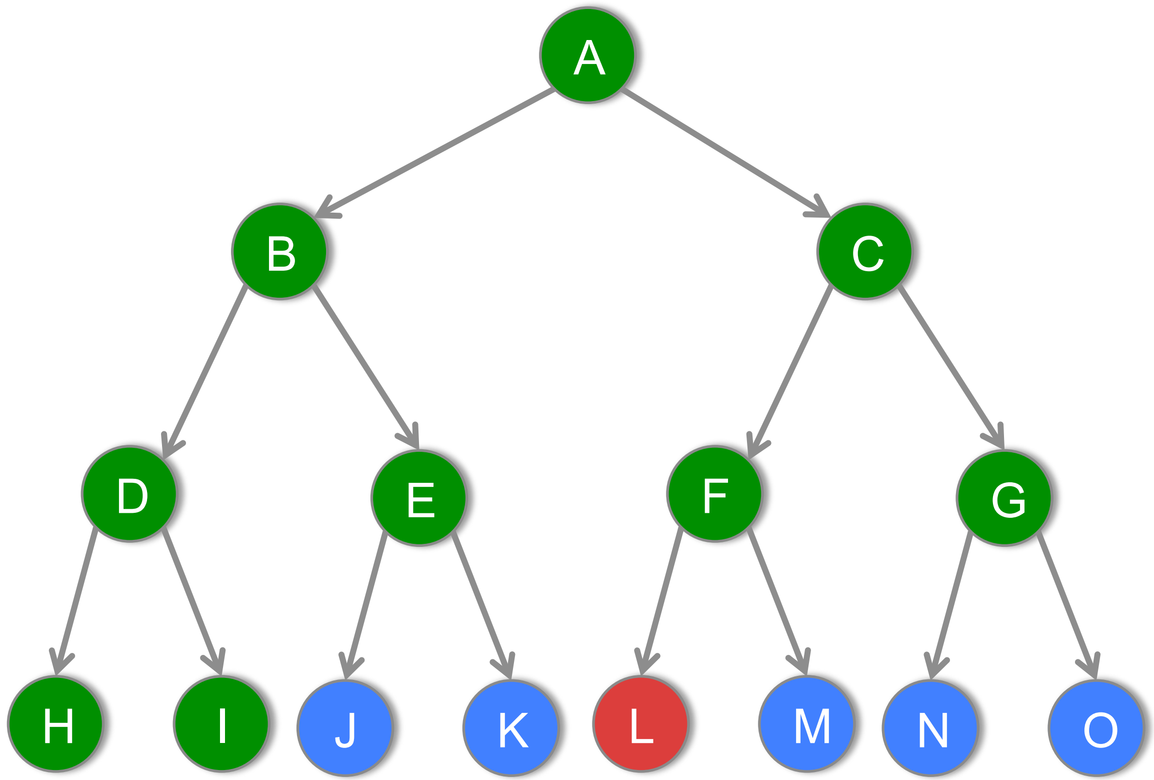
Queue: | H | I | J | K | L | M | N | O |

BREADTH FIRST SEARCH



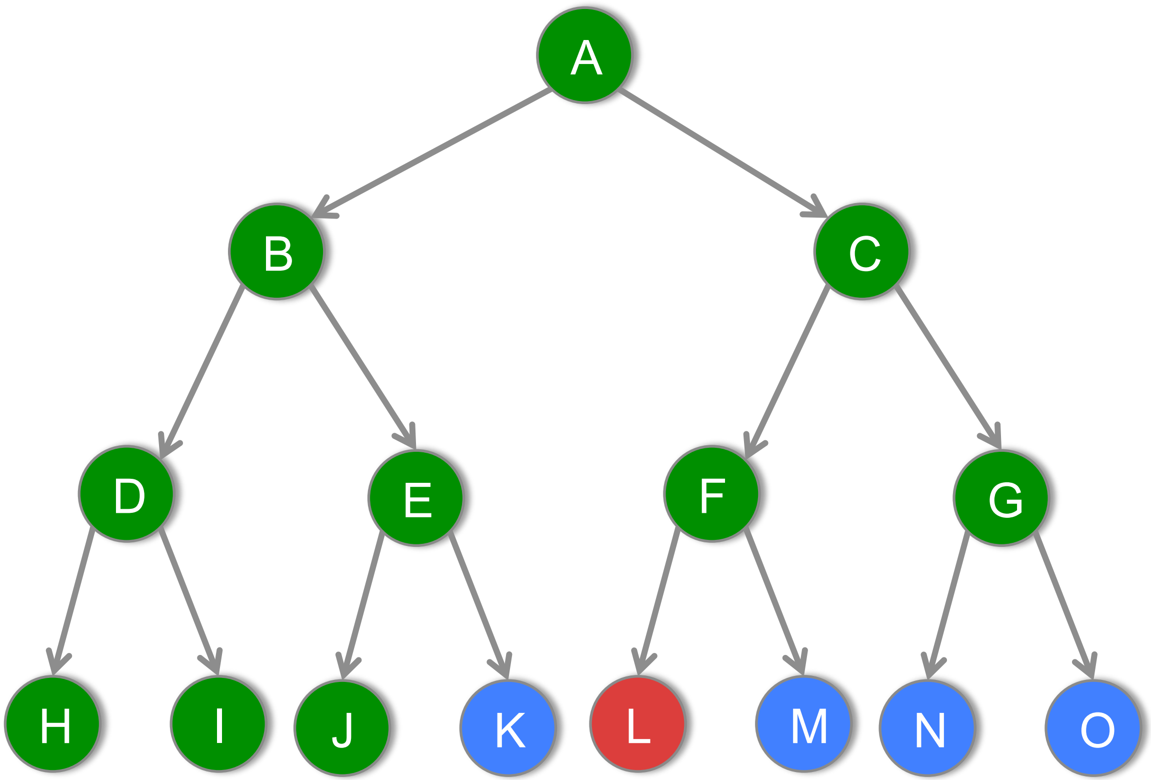
Queue: I | J | K | L | M | N | O

BREADTH FIRST SEARCH



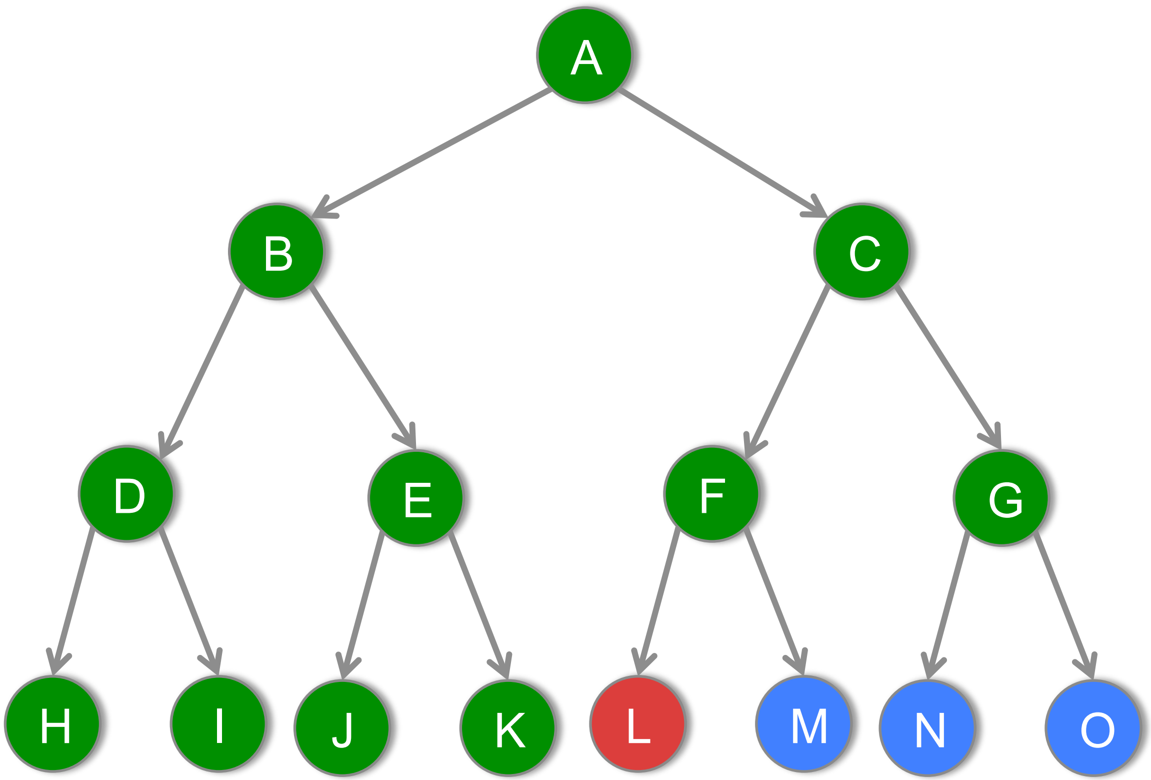
Queue: J | K | L | M | N | O

BREADTH FIRST SEARCH



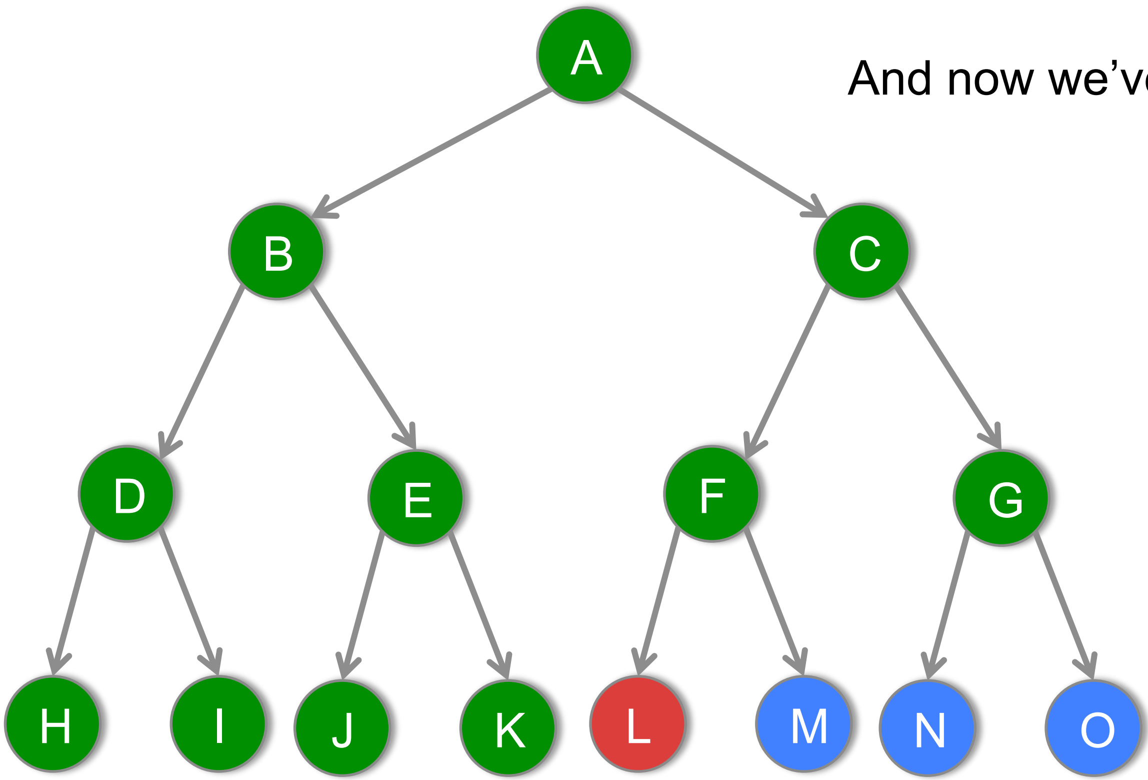
Queue: K | L | M | N | O

BREADTH FIRST SEARCH



Queue: L | M | N | O

BREADTH FIRST SEARCH



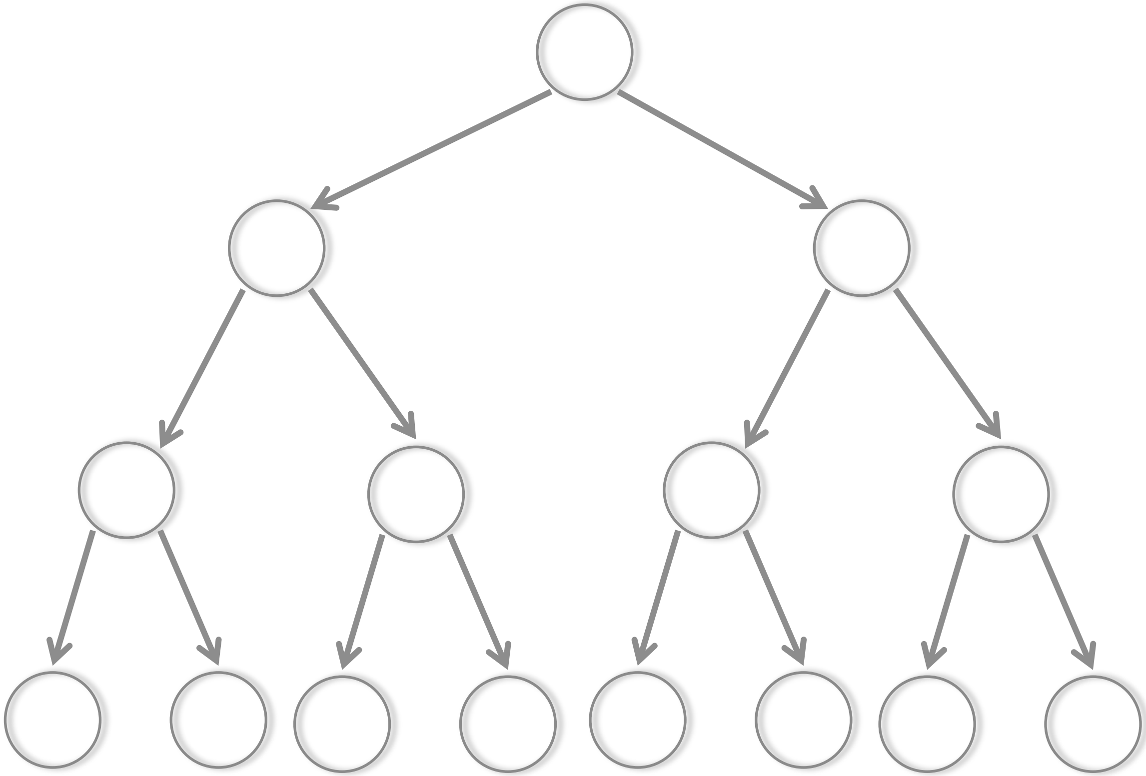
And now we've found it!

Queue: L | M | N | O

REVIEW

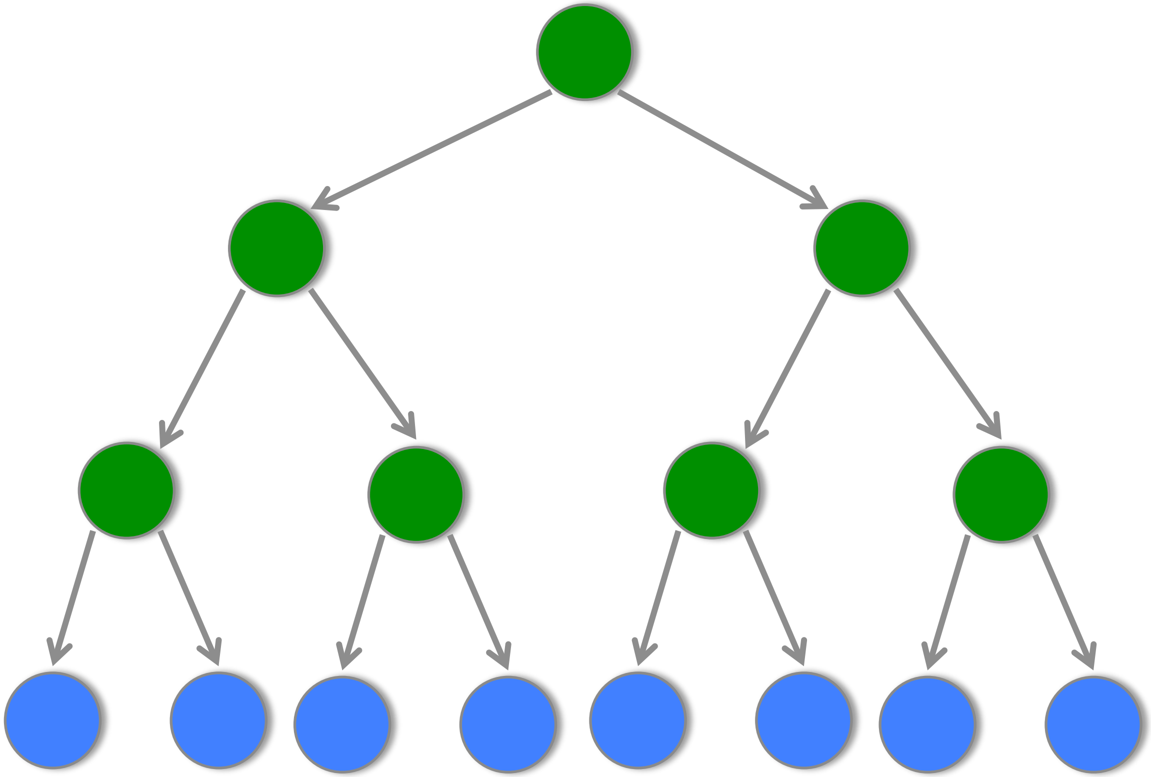
- **Breadth First Search**
  - Enqueue the root
  - While the queue has elements
    - Dequeue
    - Process
    - Enqueue children
  - How much memory does this take?

SEARCH MEMORY USE



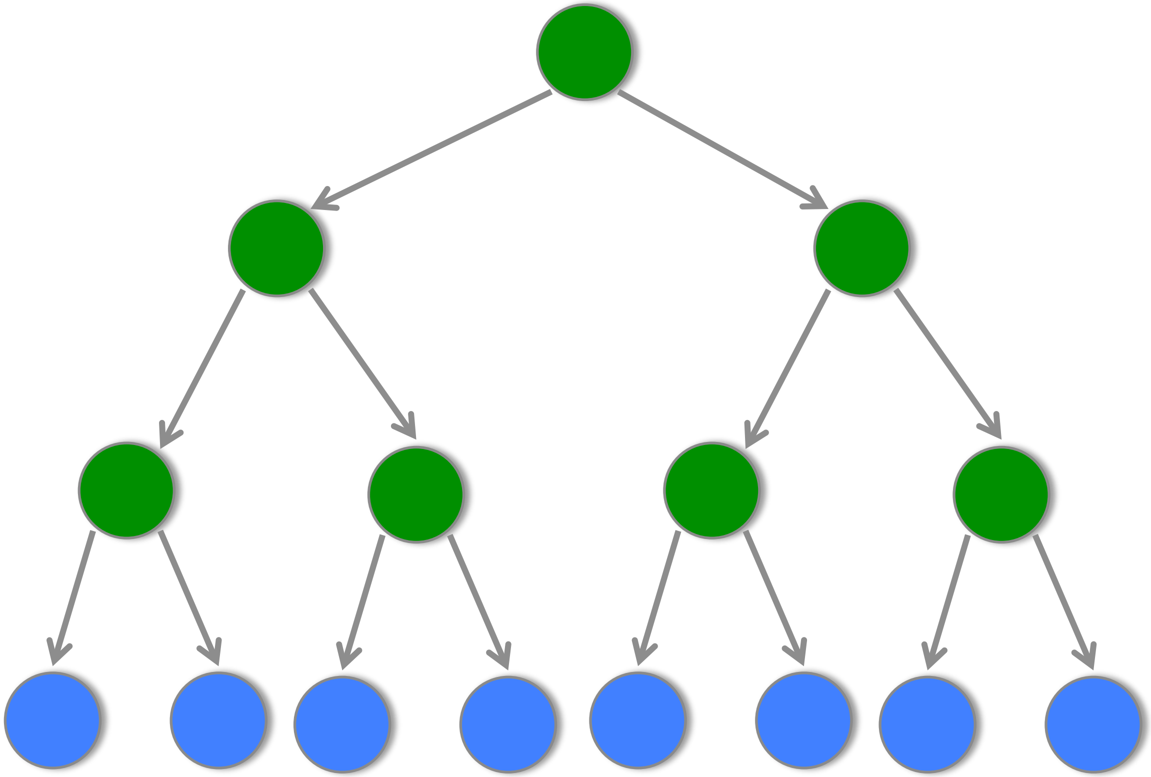
- **When does the queue have the most elements?**

SEARCH MEMORY USE



- **At the widest point in the traversal**

SEARCH MEMORY USE



- **At the widest point in the traversal**
  - How many elements is this?

SEARCH MEMORY USE

- **Breadth First Search**
  - In a perfect tree (where every row is complete) of size **n**, how many elements are in the last row?

SEARCH MEMORY USE

- **Breadth First Search**
  - In a perfect tree (where every row is complete) of size **n**, how many elements are in the last row?
    - **N/2**

SEARCH MEMORY USE

- **Breadth First Search**
  - In a perfect tree (where every row is complete) of size **n**, how many elements are in the last row?
    - **ceiling(N/2)**

SEARCH MEMORY USE

- **Breadth First Search**
  - In a perfect tree (where every row is complete) of size **n**, how many elements are in the last row?
    - **ceiling(N/2)**, this is important to know!

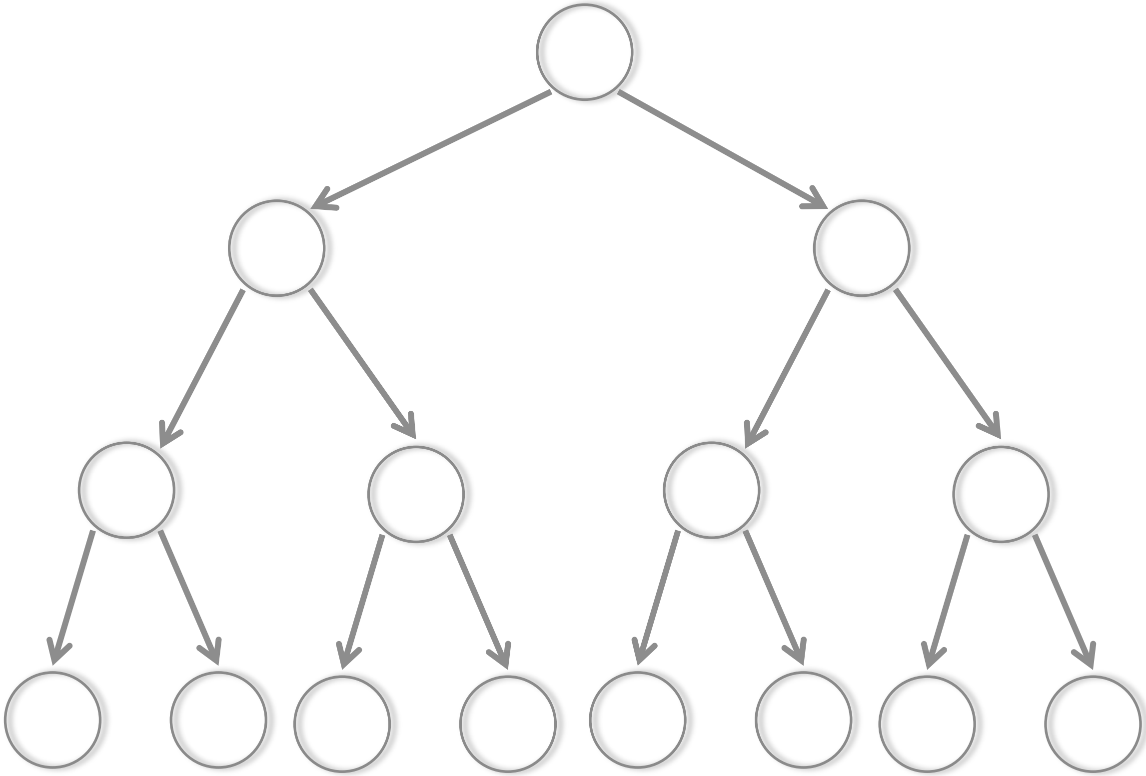
SEARCH MEMORY USE

- **Breadth First Search**
  - In a perfect tree (where every row is complete) of size **n**, how many elements are in the last row?
    - **ceiling(N/2)**, this is important to know!
    - **O(n)** memory usage!

SEARCH MEMORY USE

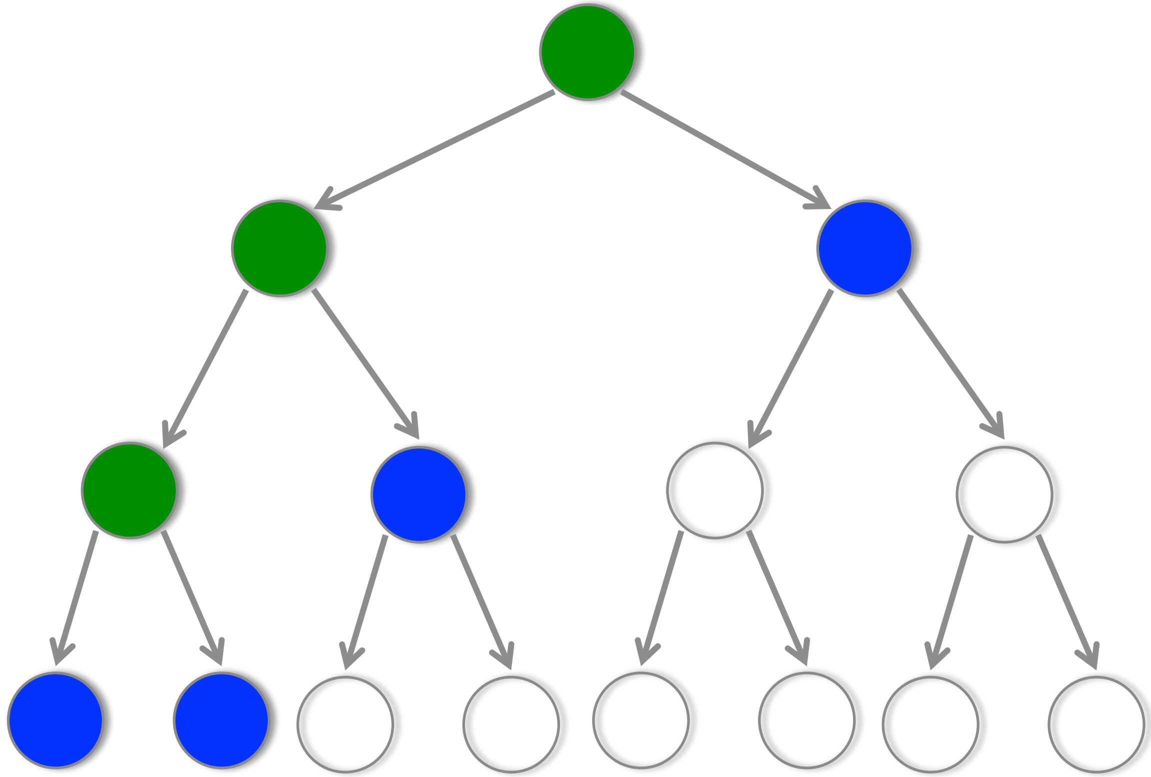
- **What about depth first search?**
  - When does the stack have the most elements on it?

SEARCH MEMORY USE



- **When does the stack have the most elements?**
  - When it's at the bottom

SEARCH MEMORY USE



- **When does the stack have the most elements?**
  - When it's at the bottom

SEARCH MEMORY USE

- How many elements are in the stack in this worst case?

SEARCH MEMORY USE

- **How many elements are in the stack in this worst case?**
  - The height of the tree

SEARCH MEMORY USE

- **How many elements are in the stack in this worst case?**
  - The height of the tree, **O(n)** if the tree is one-sided, but **O(log n)** if the tree is balanced

SEARCH MEMORY USE

- **How many elements are in the stack in this worst case?**
  - The height of the tree, **O(n)** if the tree is one-sided, but **O(log n)** if the tree is balanced
  - We will discuss balance later

SEARCH MEMORY USE

- **How many elements are in the stack in this worst case?**
  - The height of the tree, **O(n)** if the tree is one-sided, but **O(log n)** if the tree is balanced
  - We will discuss balance later
  - Classic exam question! Consider memory AND execution times

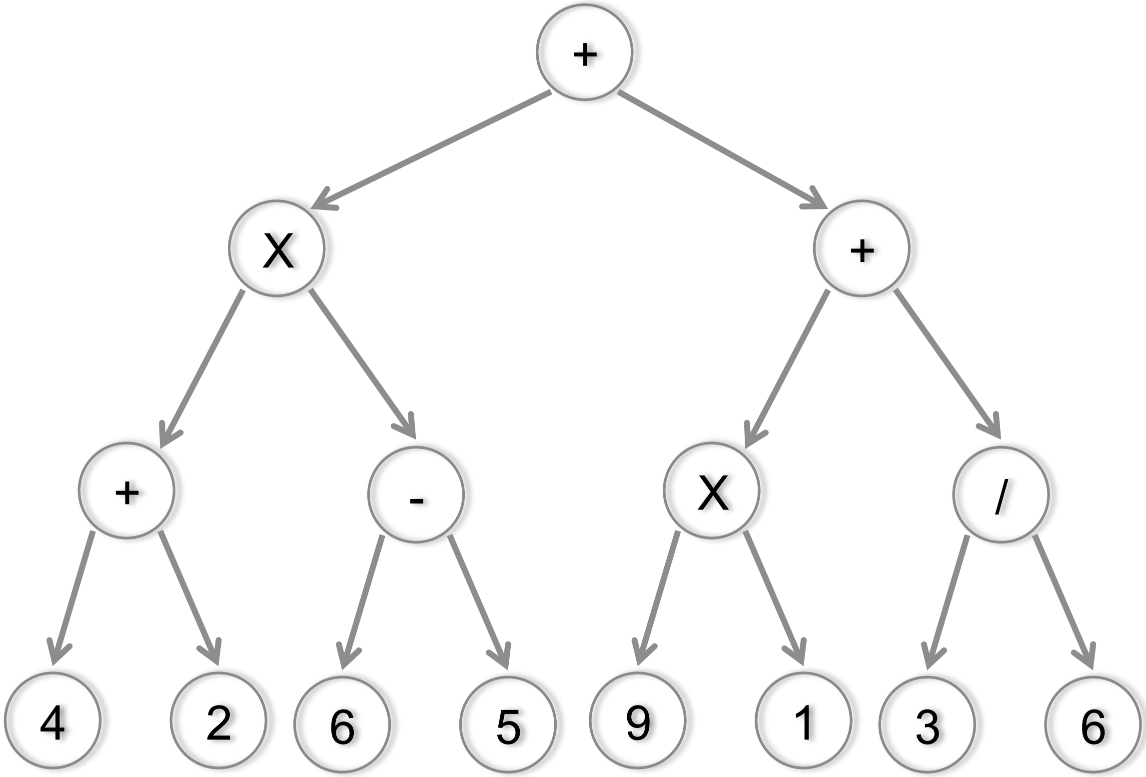
REVIEW

- **Ordering**
  - What is the difference between these three implementations
    - Process; DFS(left); DFS(right)
    - DFS(left); Process; DFS(right)
    - DFS(left); DFS(right); Process
  - *How does this impact the final output?*

REVIEW

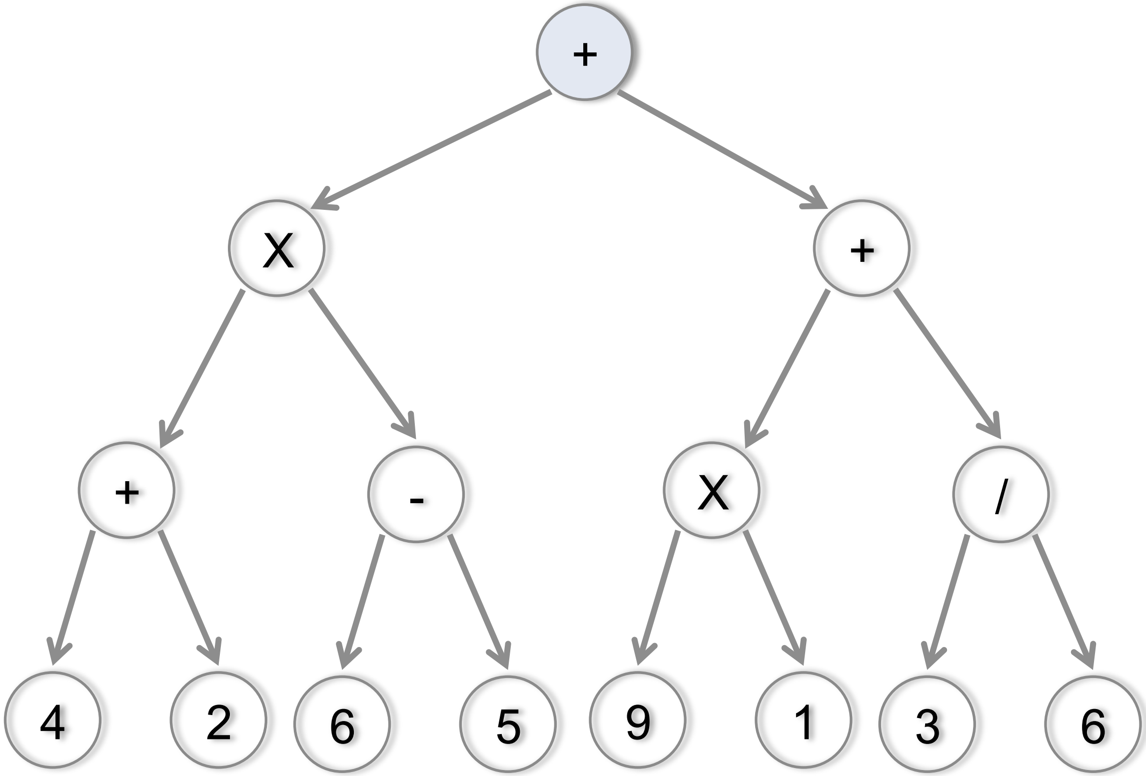
- **Ordering**
  - Three traversal types
    - Pre-order
    - In-order
    - Post-order
- **Instruction (Parse) trees**

PREORDER TRAVERSAL



Stack:	
Output:	

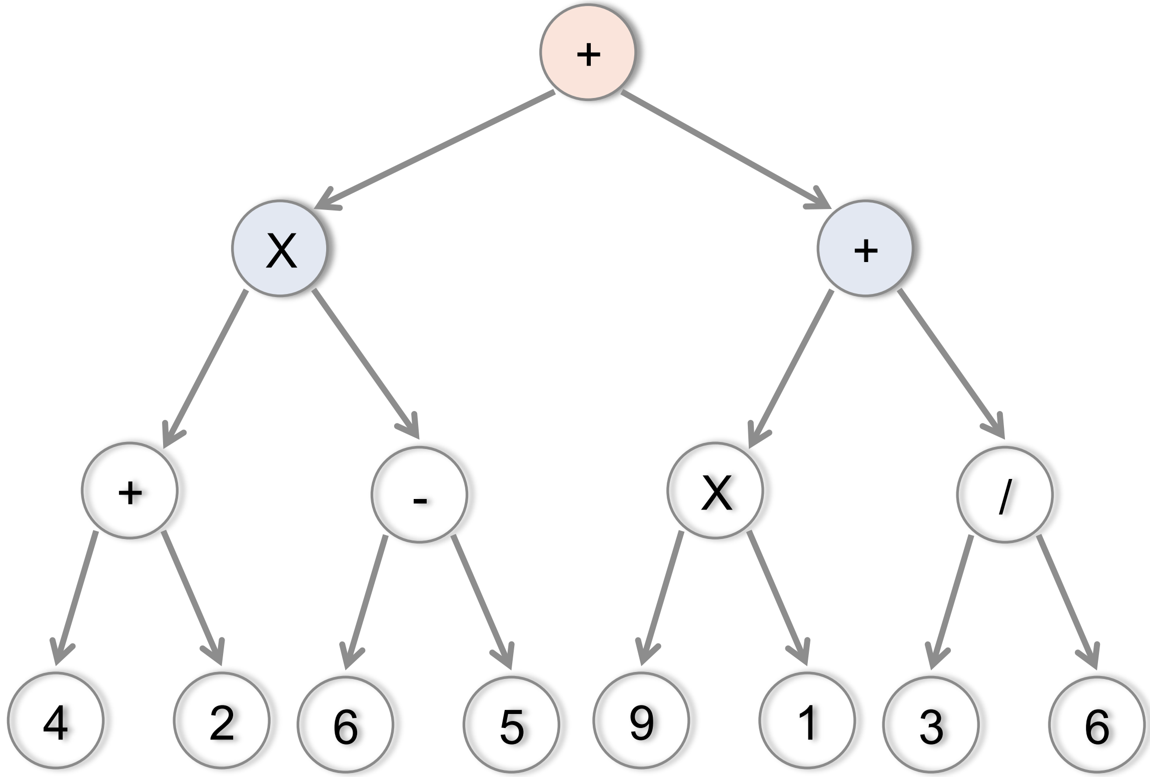
PREORDER TRAVERSAL



Add the root to the stack

Stack:	+ |
Output:	

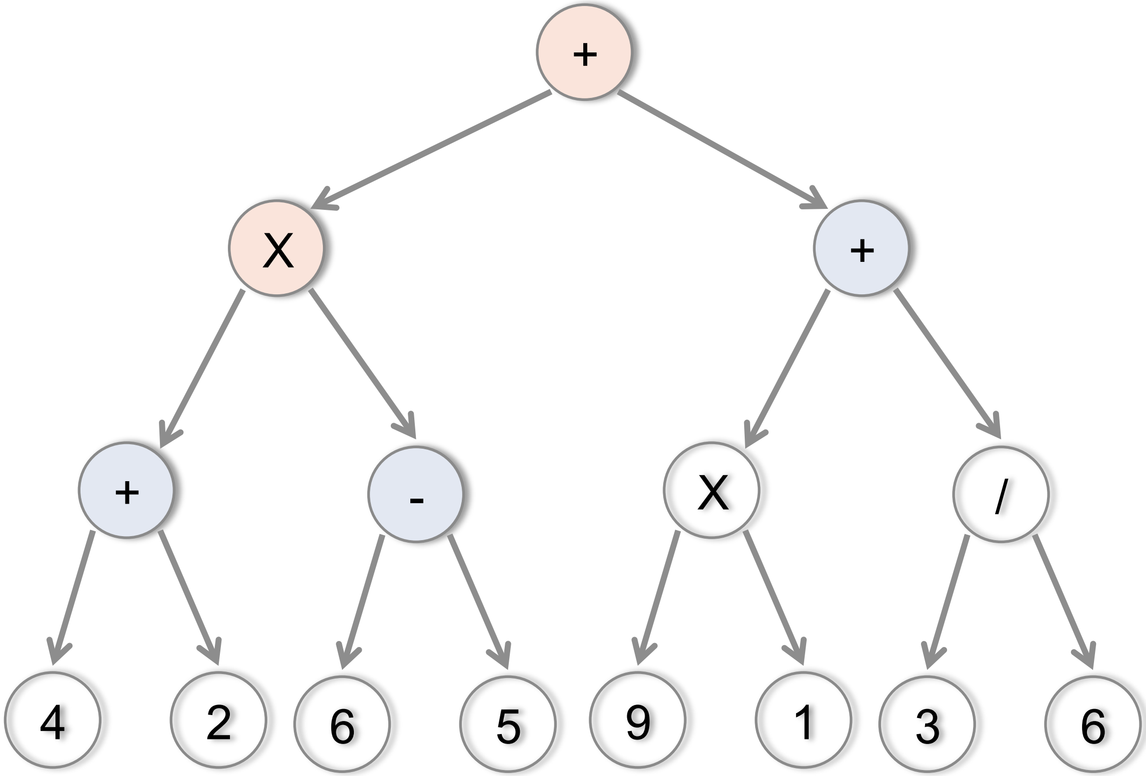
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	X | +
Output:	+

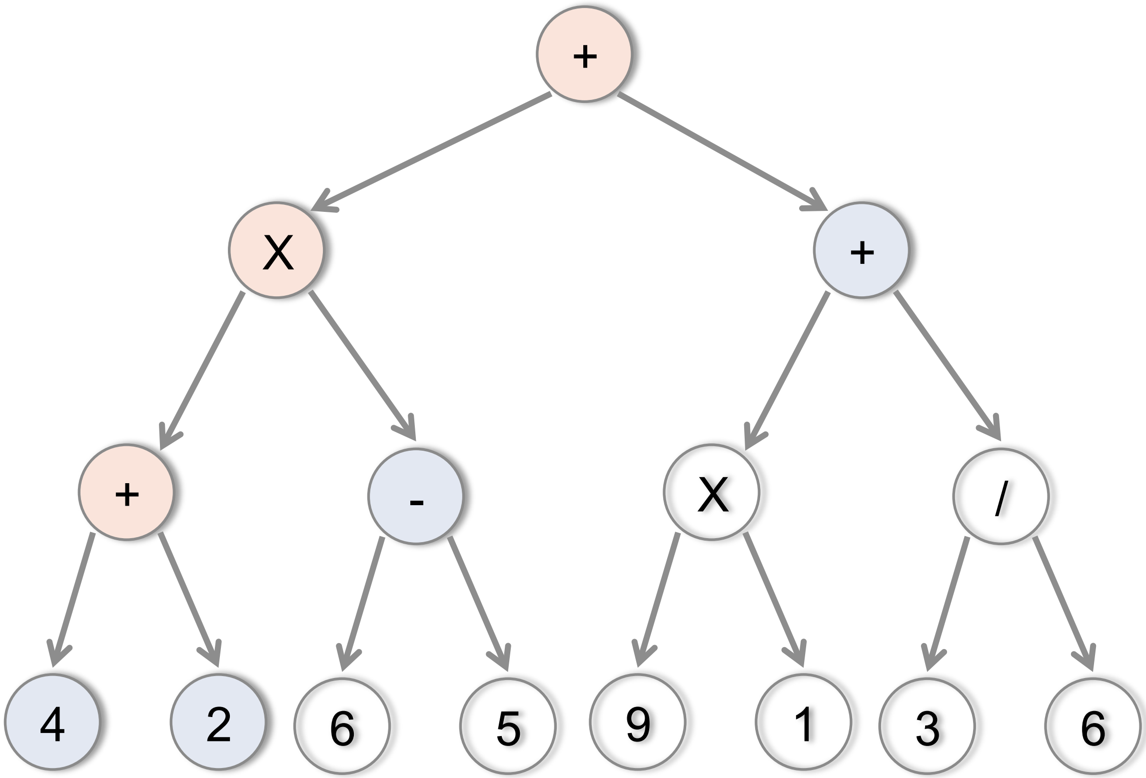
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	+ | - | +
Output:	+X

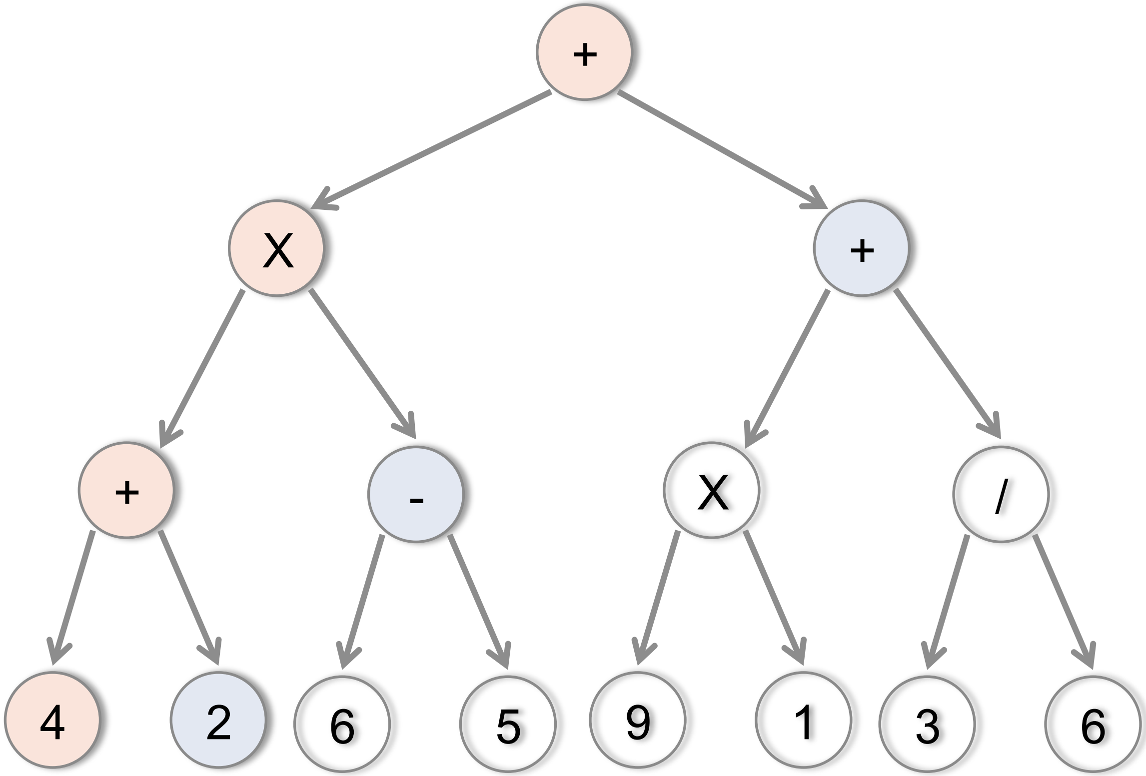
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	4 | 2 | - | +
Output:	+X+

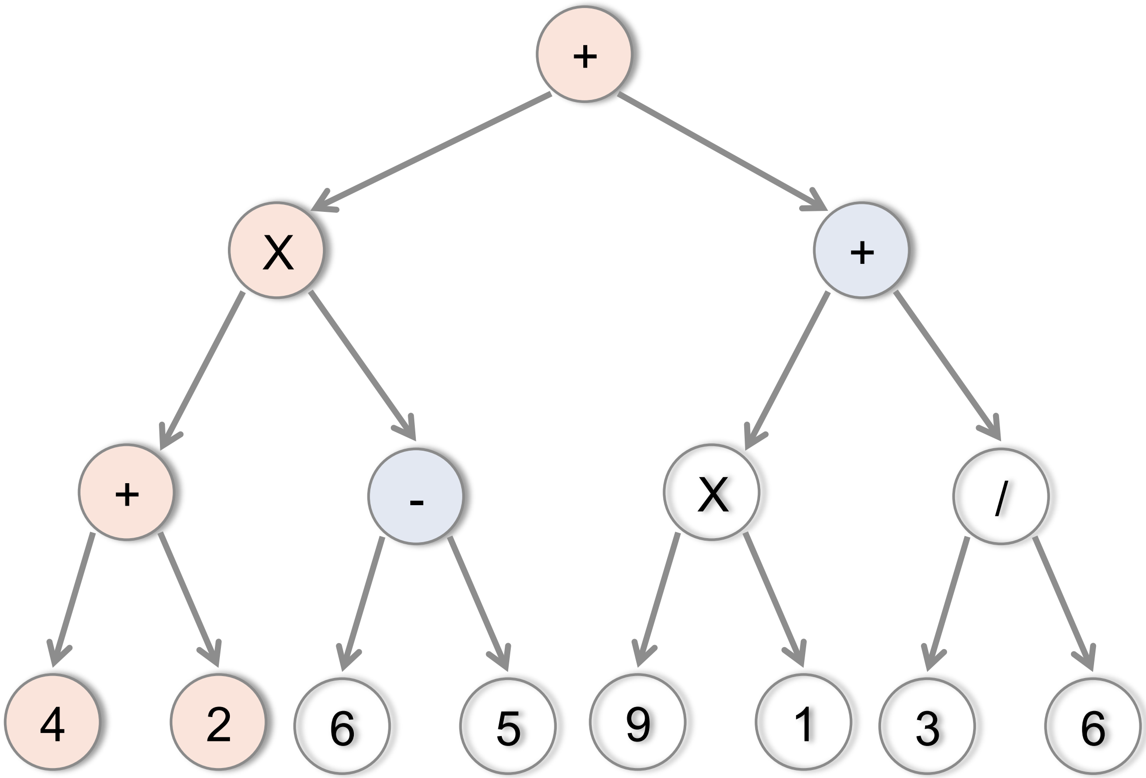
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	2 | - | +
Output:	+X+4

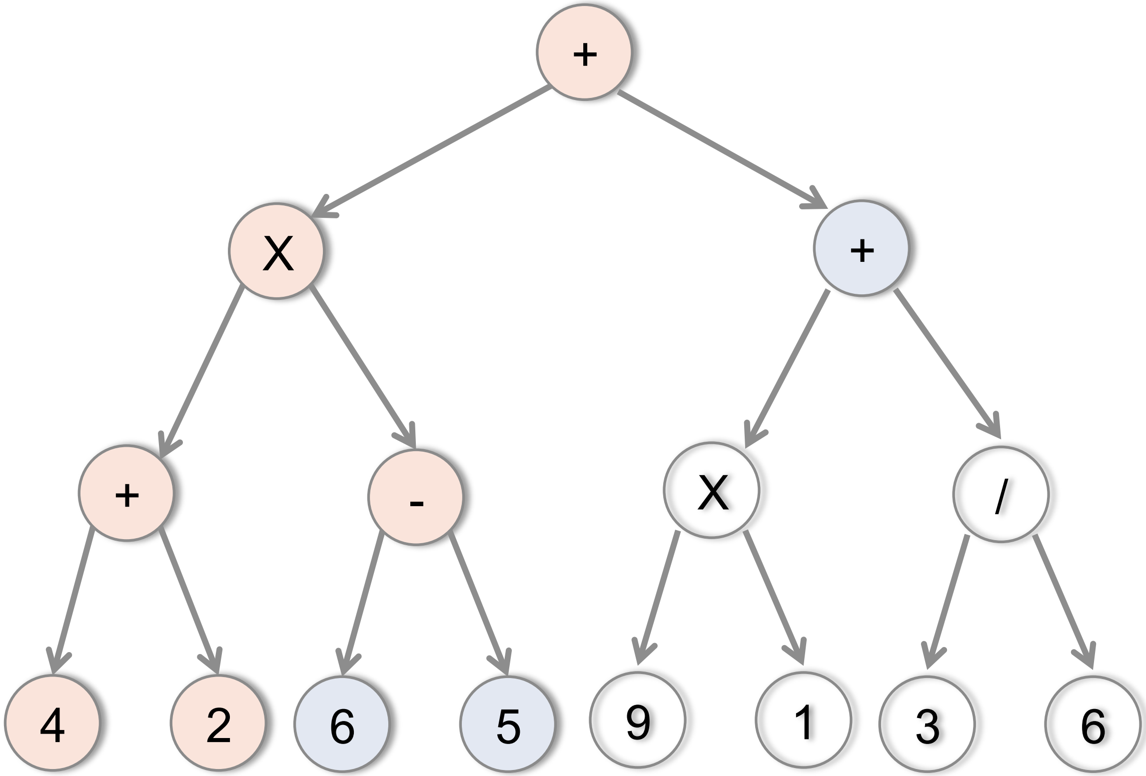
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	- | +
Output:	+X+42

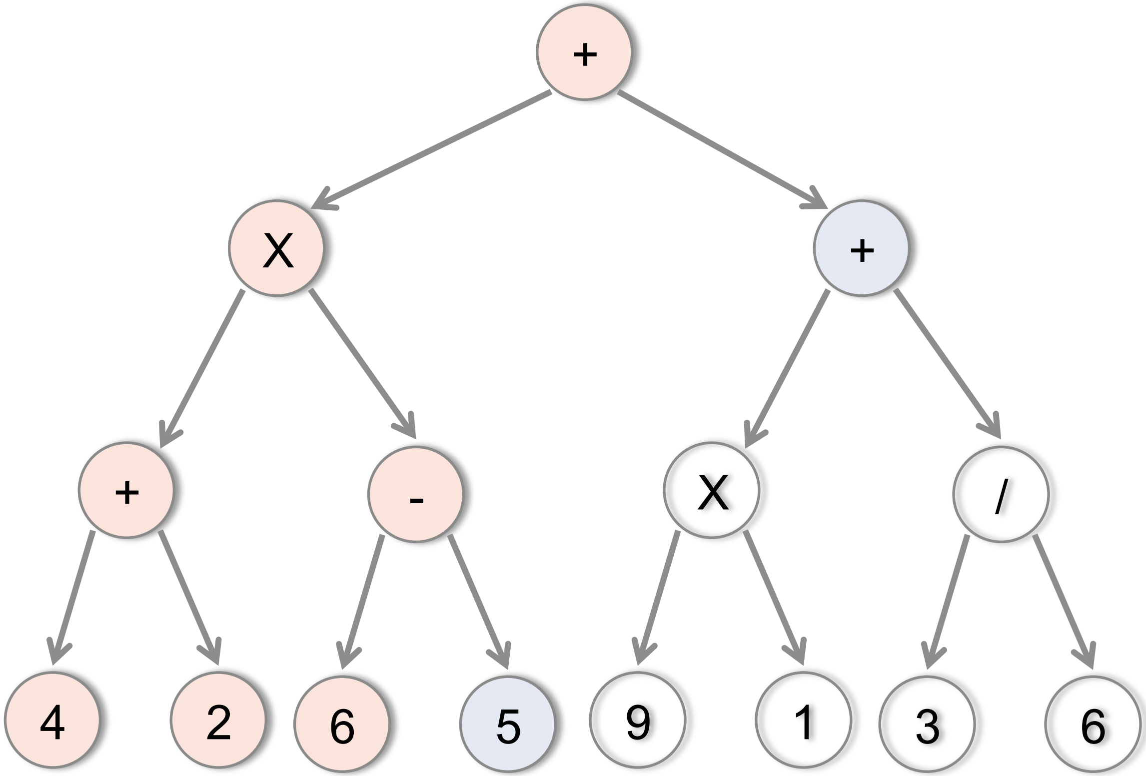
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	6 | 5 | +
Output:	+X+42-

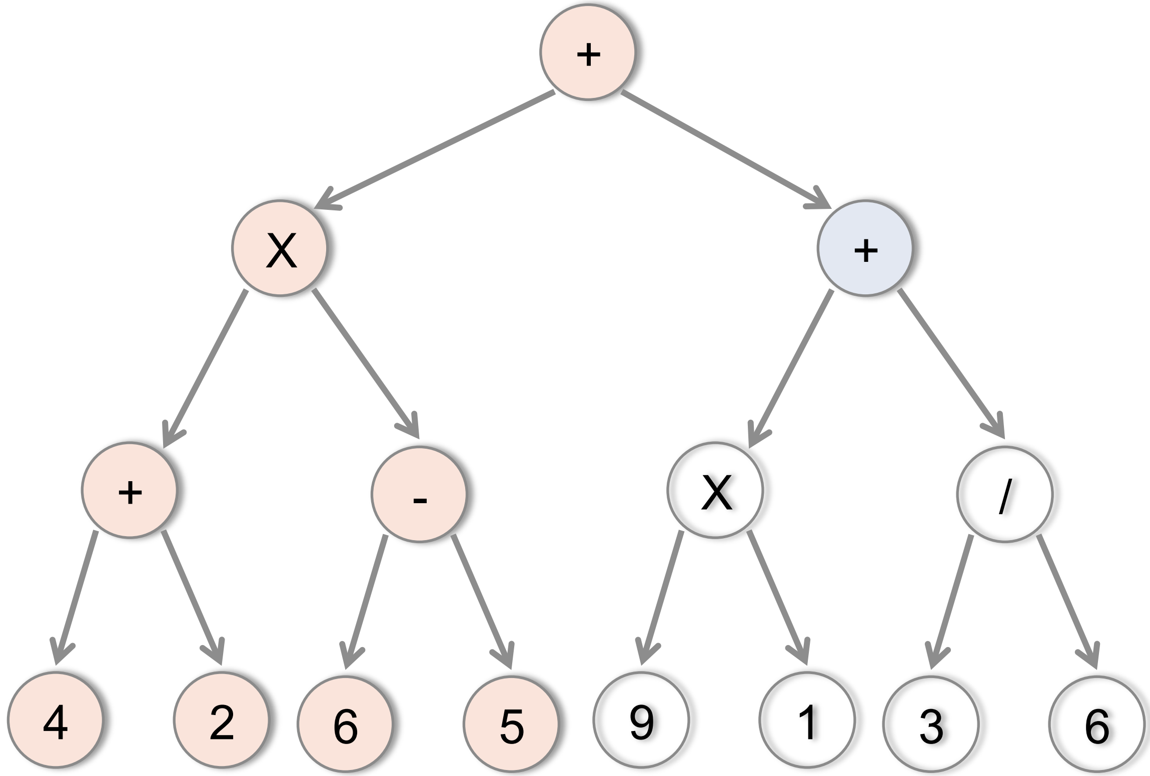
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	5 | +
Output:	+X+42-6

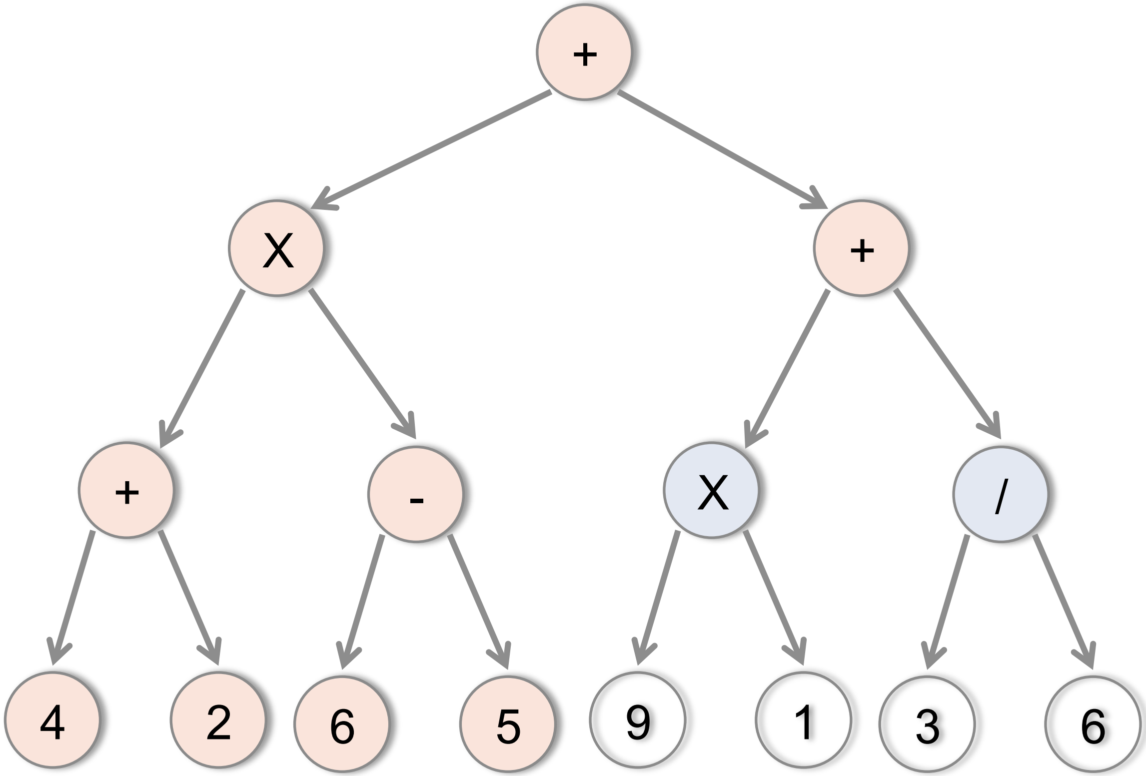
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	+
Output:	+X+42-65

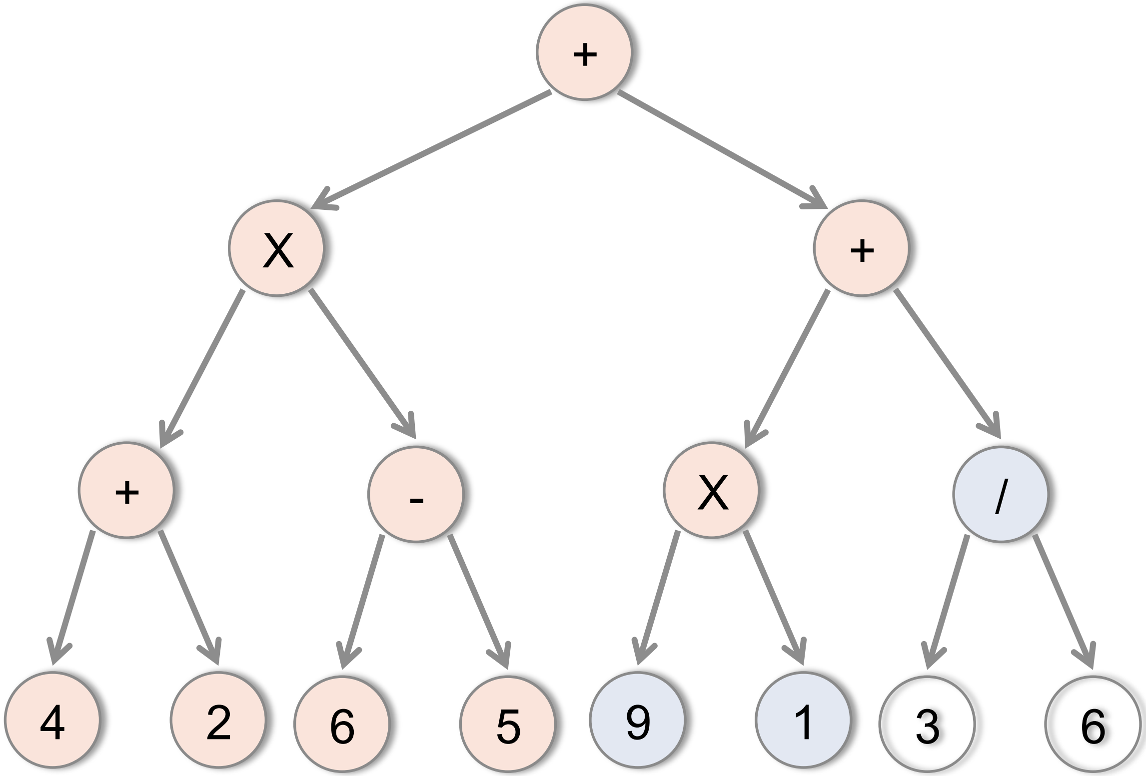
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	X | /
Output:	+X+42-65+

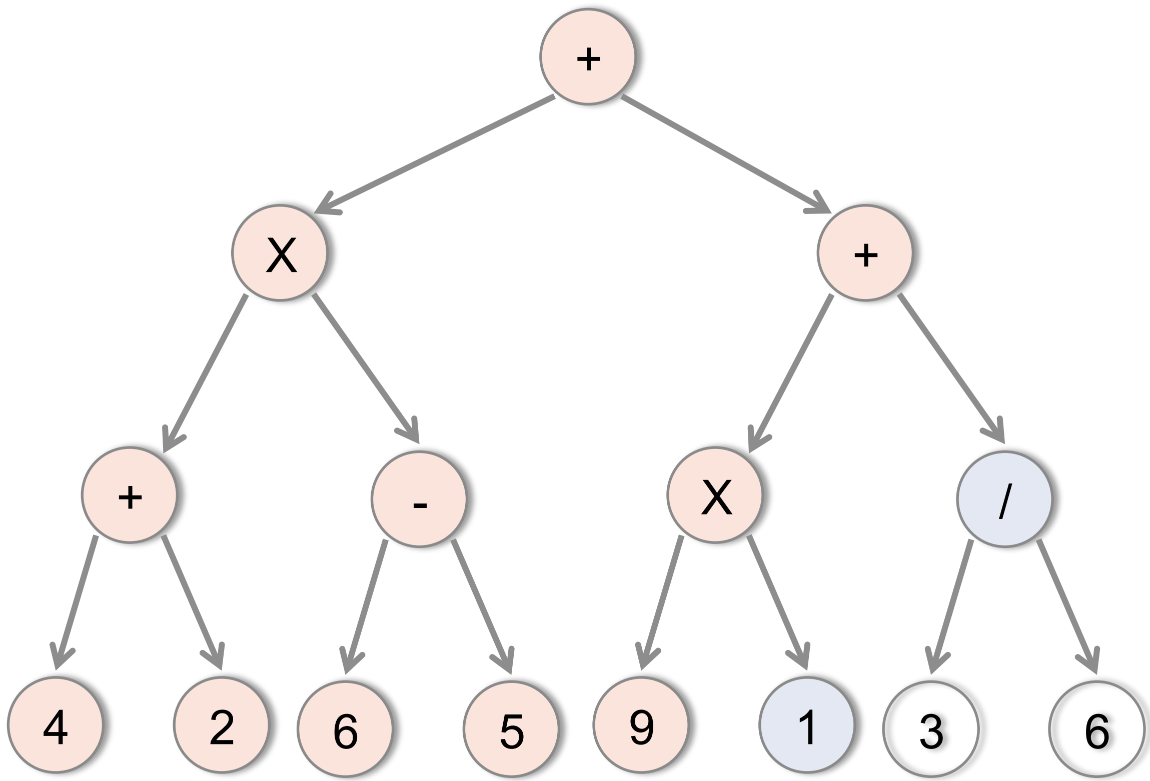
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	9 | 1 | /
Output:	+X+42-65+X

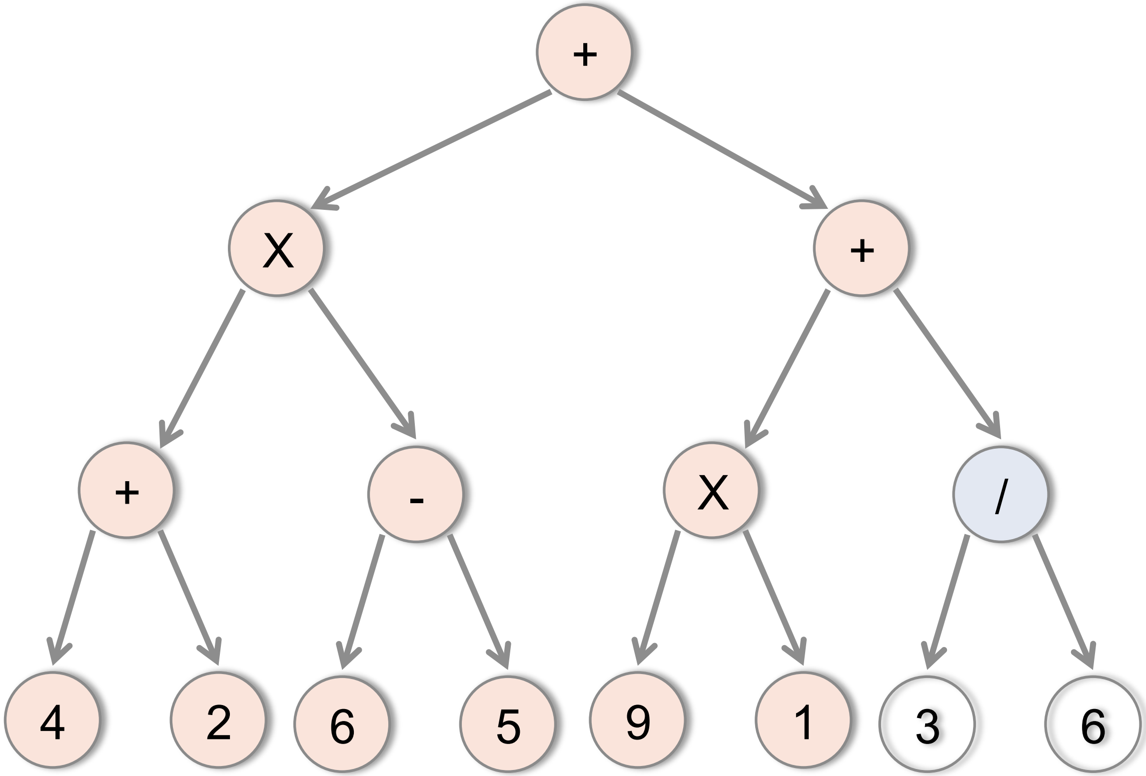
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	1 | /
Output:	+X+42-65+X9

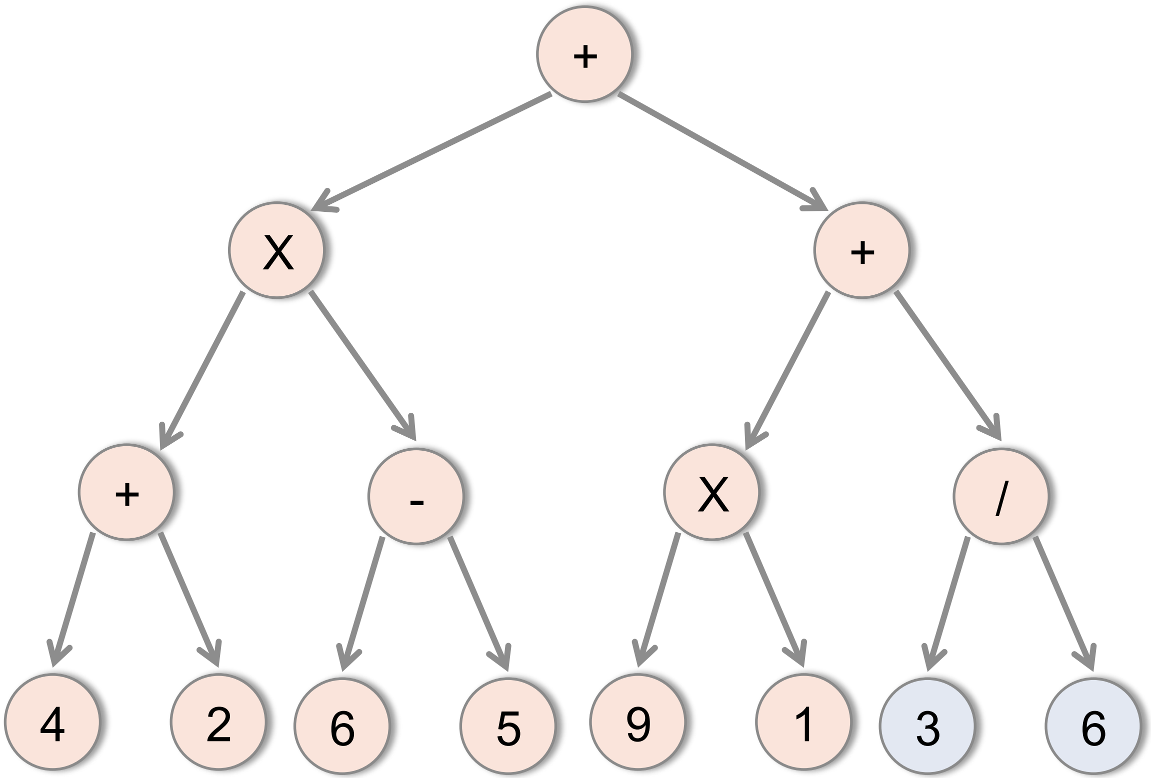
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	/
Output:	+X+42-65+X91

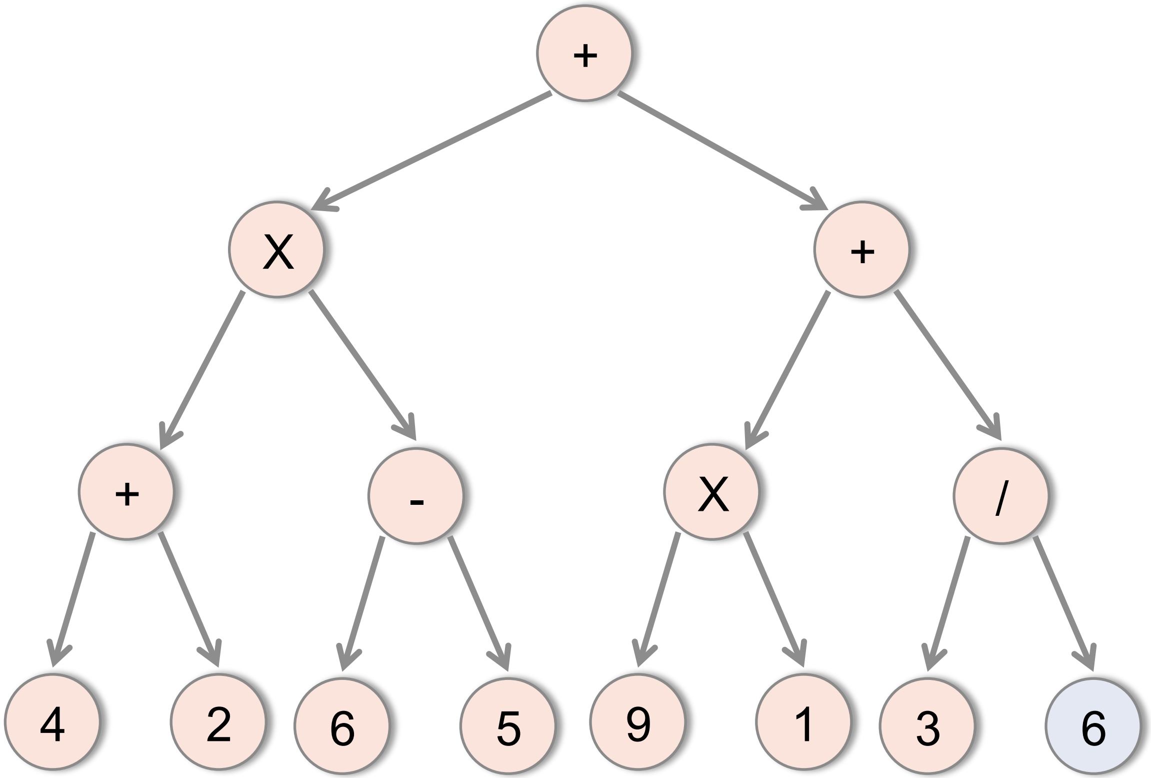
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	3 | 6
Output:	+X+42-65+X91/

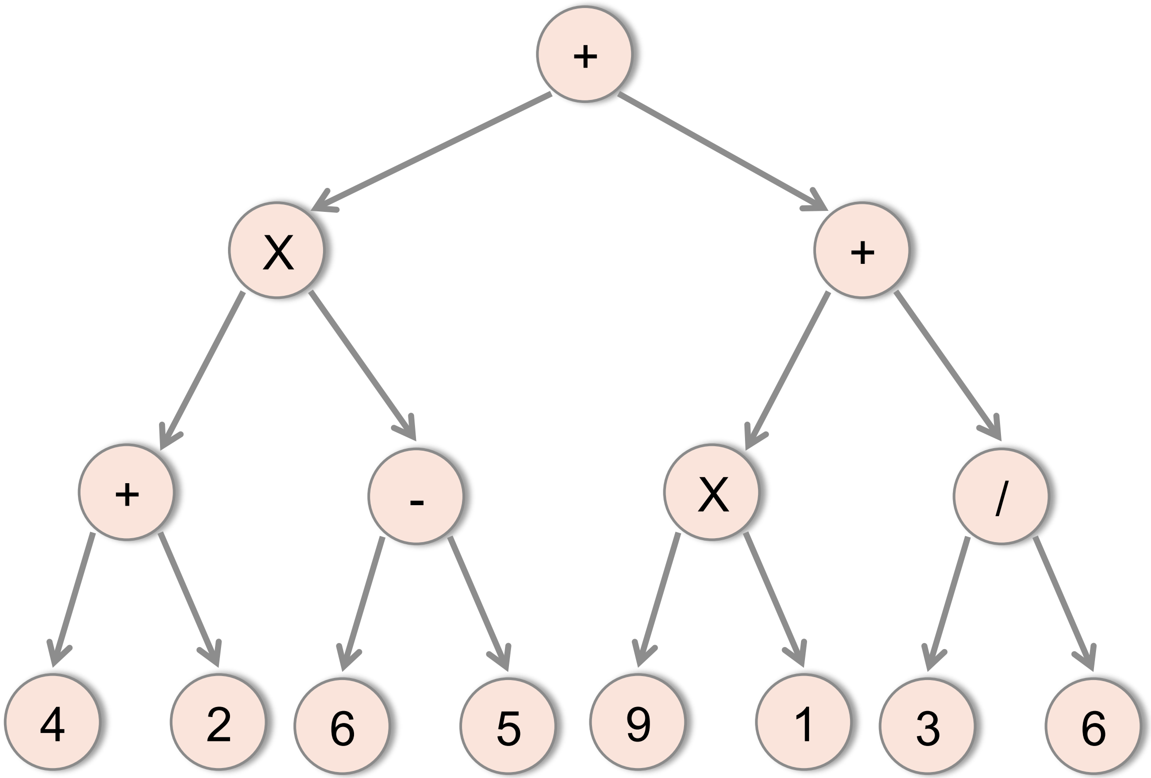
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	6
Output:	+X+42-65+X91/3

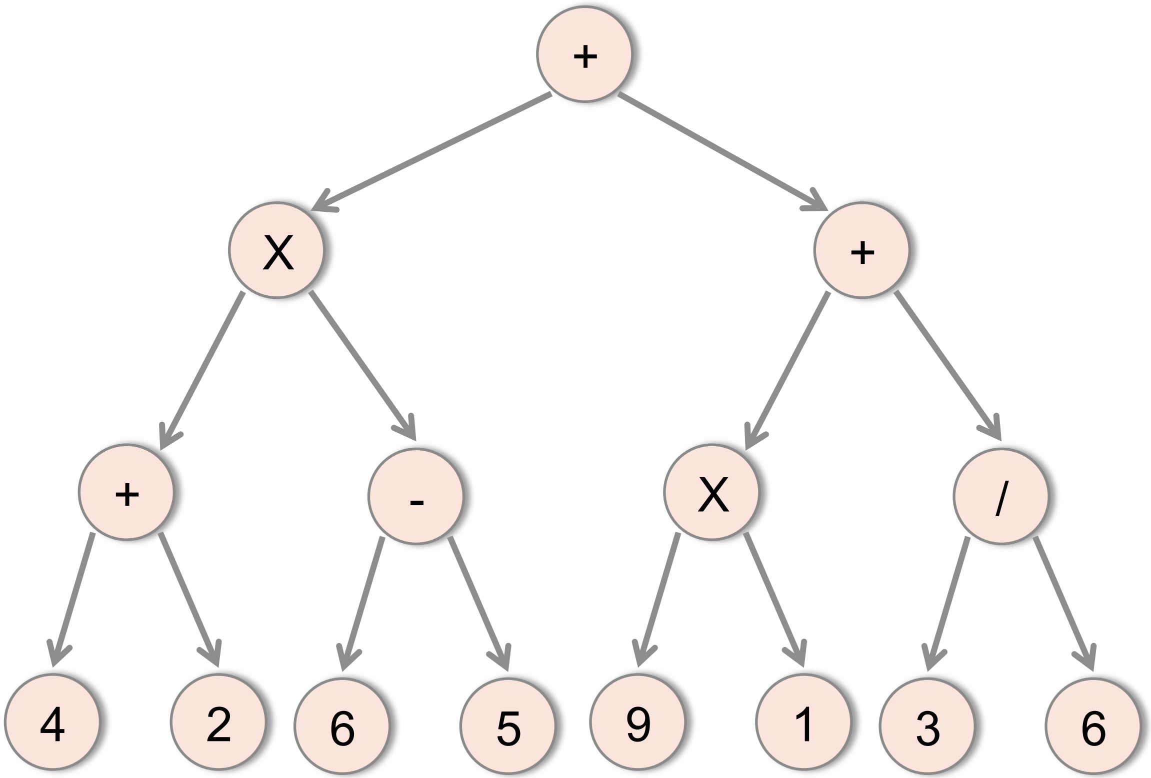
PREORDER TRAVERSAL



Process the node and then add children (right then left)

Stack:	
Output:	+X+42-65+X91/36

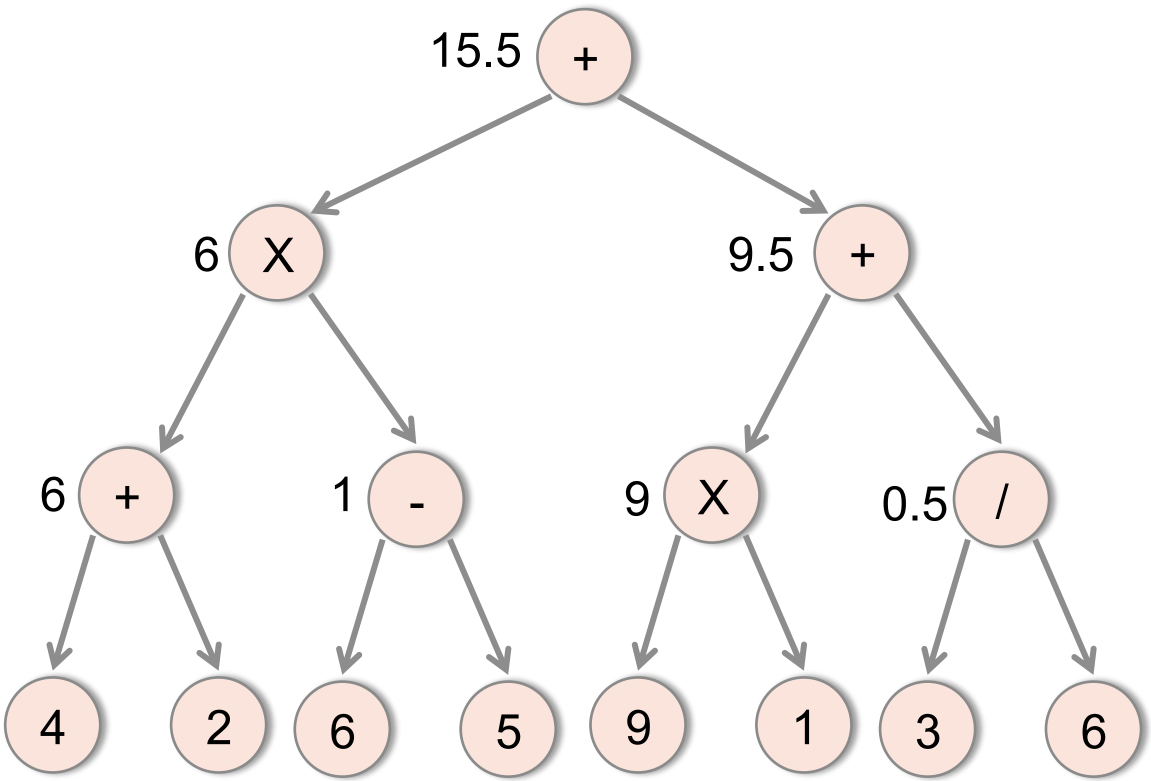
PREORDER TRAVERSAL



What does this evaluate to?

Stack:	
Output:	+X+42-65+X91/36

PREORDER TRAVERSAL



What does this evaluate to?

Stack:	
Output:	+X+42-65+X91/36

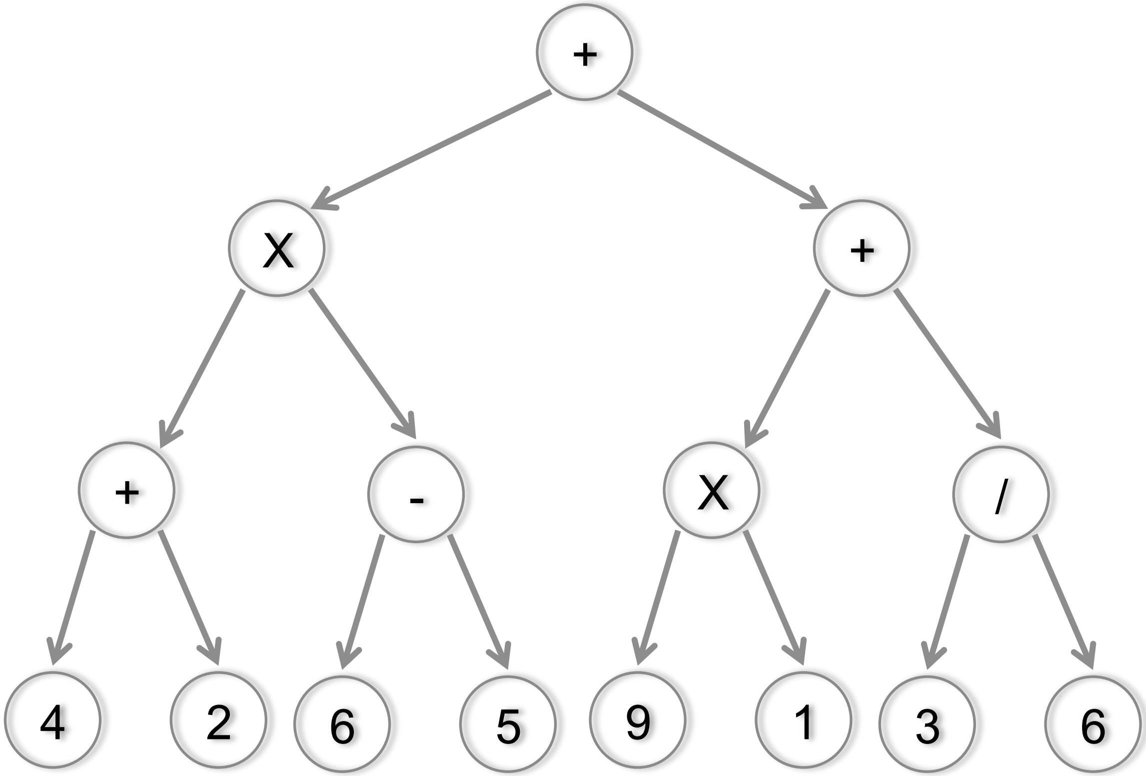
PREORDER TRAVERSAL

- **Knowing the rule of preorder, is that string ambiguous?**
  - +X+42-65+X91/36

PREORDER TRAVERSAL

- **Knowing the rule of preorder, is that string ambiguous?**
  - +X+42-65+X91/36
- **Given that preorder traversal is DFS with ordering:**
  - Process, Left, Right
- **What string results from postorder?**
  - Left Right Process?

POSTORDER TRAVERSAL



POSTORDER TRAVERSAL

- **Pre-order**
  - +X+42-65+X91/36
- **Post-order**
  - 42+65-X91X36/++

POSTORDER TRAVERSAL

- **Pre-order** (Polish Notation)
  - +X+42-65+X91/36
- **Post-order** (Reverse Polish Notation)
  - 42+65-X91X36/++

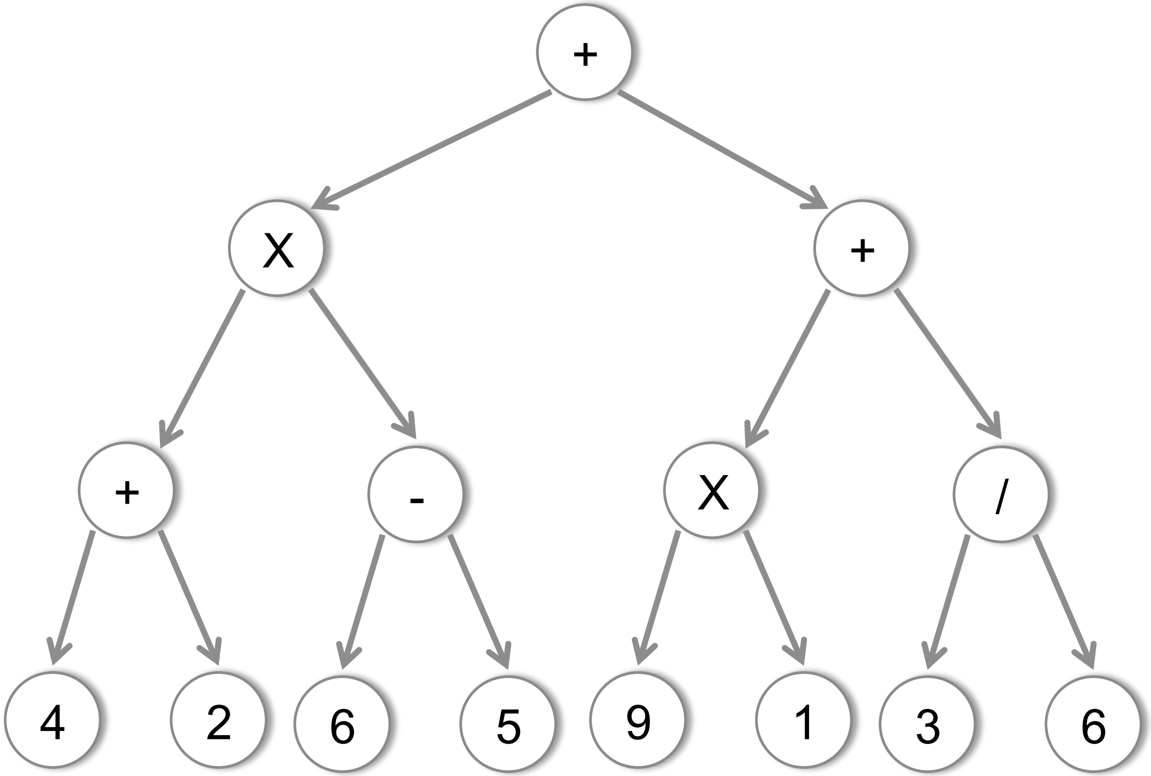
POSTORDER TRAVERSAL

- **Pre-order** (Polish Notation)
  - +X+42-65+X91/36
- **Post-order** (Reverse Polish Notation)
  - 42+65-X91X36/++
- **These are unambiguous strings**

POSTORDER TRAVERSAL

- **Pre-order** (Polish Notation)
  - +X+42-65+X91/36
- **Post-order** (Reverse Polish Notation)
  - 42+65-X91X36/++
- **These are unambiguous strings**
- **What about the final ordering?**
  - Left, Process, Right?

IN-ORDER TRAVERSAL



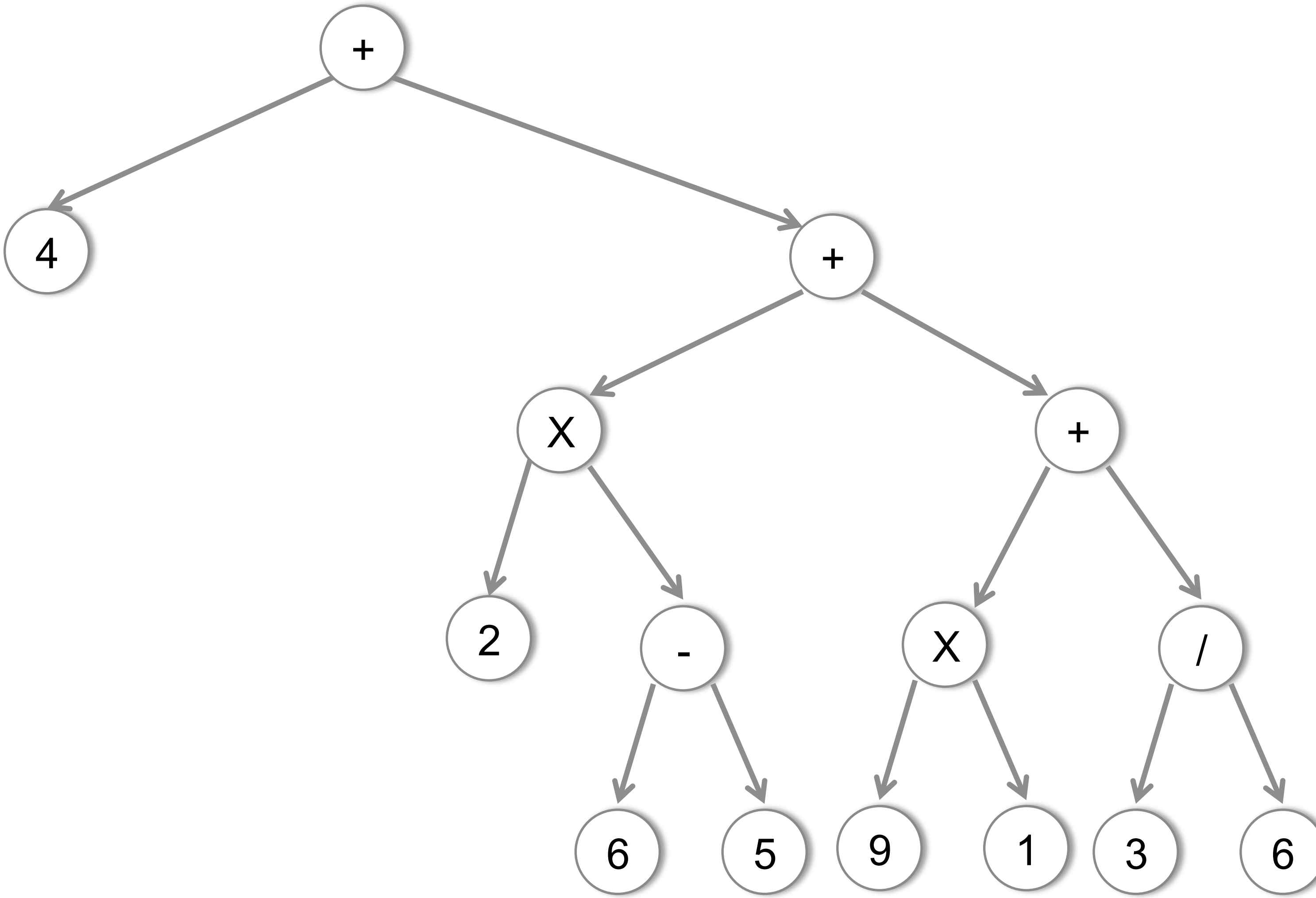
IN-ORDER TRAVERSAL

- **In-order**
  - 4+2X6-5+9X1+3/6

IN-ORDER TRAVERSAL

- **In-order**
  - 4+2X6-5+9X1+3/6
- **What is the problem here?**

IN-ORDER TRAVERSAL



TRAVERSALS

- **In-order**
  - 4+2X6-5+9X1+3/6
- **What is the problem here?**
  - There are multiple trees!

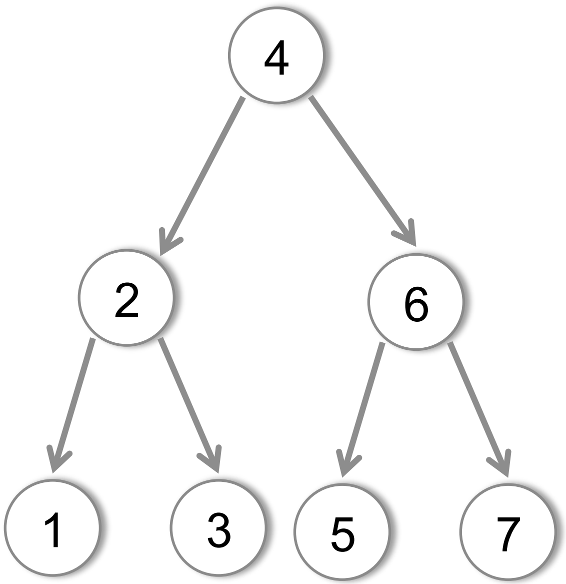
TRAVERSALS

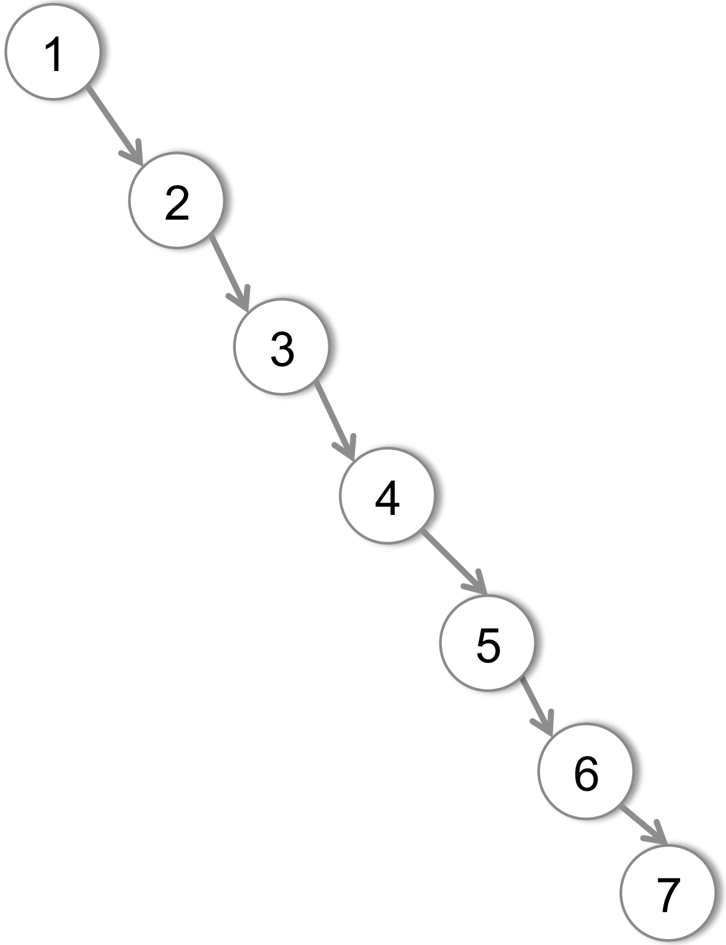
- **In-order**
  - 4+2X6-5+9X1+3/6
- **What is the problem here?**
  - There are multiple trees!
- **In order returns the left-to-right sorted order**
  - In-order traversal of a BST is sorted result

BALANCE AND HEIGHT

- If the same data can be represented multiple ways, what is best?

BALANCE AND HEIGHT





BALANCE AND HEIGHT

- **Height is key for how fast functions on our tree are!**
  - If we can structure the same data two different ways, we want to choose the better one.
  - Balanced is better for BSTs
  - Can we enforce balance?

BALANCE AND HEIGHT

- Balance

BALANCE AND HEIGHT

- **Balance**
  - How can we define balance?

BALANCE AND HEIGHT

- **Balance**
  - How can we define balance?
  - Abs(height(left) – height(right))

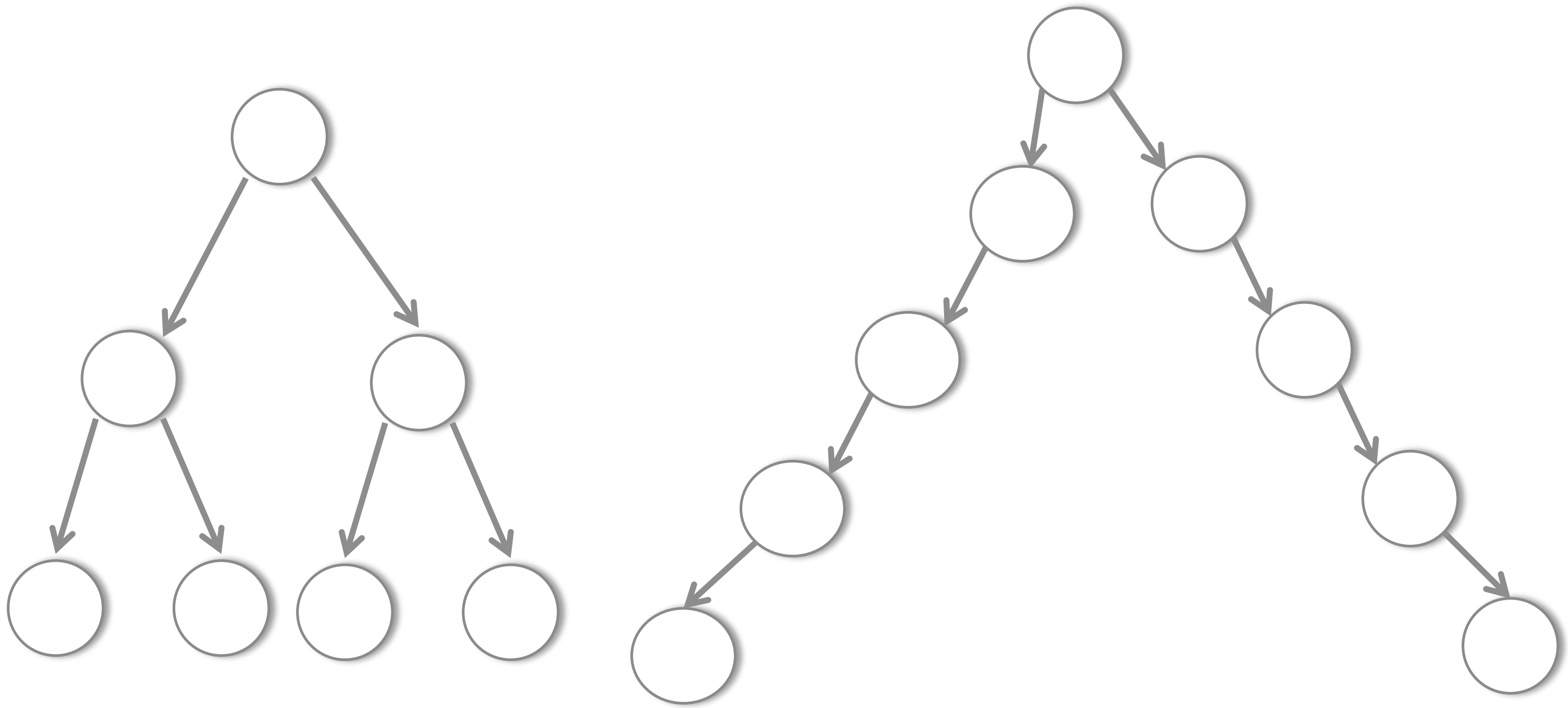
BALANCE AND HEIGHT

- **Balance**
  - How can we define balance?
  - Abs(height(left) – height(right))
  - If the heights of the left and right trees are balanced, the tree is balanced.

BALANCE AND HEIGHT

- **Balance**
  - How can we define balance?
  - Abs(height(left) – height(right))
  - If the heights of the left and right trees are balanced, the tree is balanced.
  - Anything wrong with this?

BALANCE AND HEIGHT

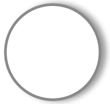


BALANCE AND HEIGHT

- **Not enough for the root to be balanced!**
- **All nodes must be balanced!**
- **Ideally, our "balance" property will say:**
  - For all nodes in the tree, height(left) = height(right)
  - What is the problem with this?
  - **Not always enforceable!**

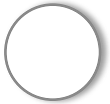
BALANCE AND HEIGHT

- **Consider adding an element to a tree.**
  - When the tree is empty, it is balanced
- **We add one element**



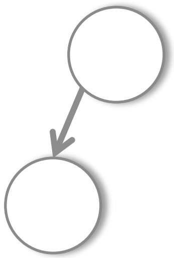
BALANCE AND HEIGHT

- **Consider adding an element to a tree.**
  - When the tree is empty, it is balanced
- **We add one element**
  - Height(left) = height(right) = 0



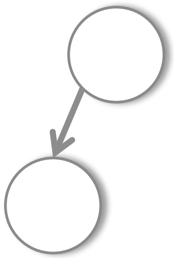
BALANCE AND HEIGHT

- **Consider adding an element to a tree.**
  - When the tree is empty, it is balanced
- **We add one element**
  - Height(left) = height(right) = 0
- **Add another element**



BALANCE AND HEIGHT

- **Consider adding an element to a tree.**
  - When the tree is empty, it is balanced
- **We add one element**
  - Height(left) = height(right) = 0
- **Add another element**
  - Oh no! There is no way to enforce balance!



BALANCE AND HEIGHT

- New property

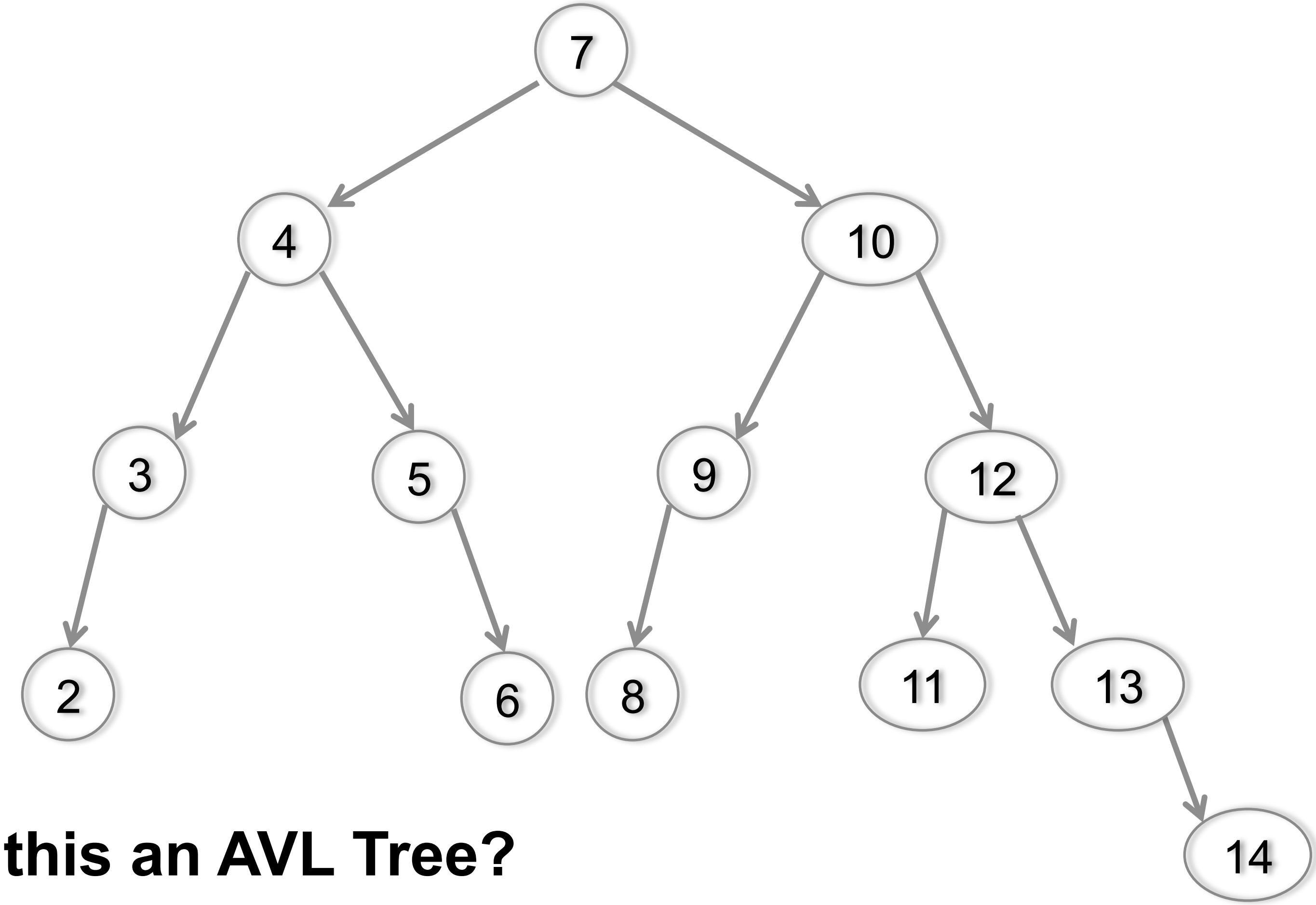
BALANCE AND HEIGHT

- **New property**
  - If Abs(height(left) – height(right)) is balance
  - We can only enforce if this is <=1
  - That is, the height left and right subtrees can differ by at most one
  - Still must preserve this for every node!
- **This is the AVL property**
- **AVL Trees are Binary Search Trees that have the AVL property**

BALANCE AND HEIGHT

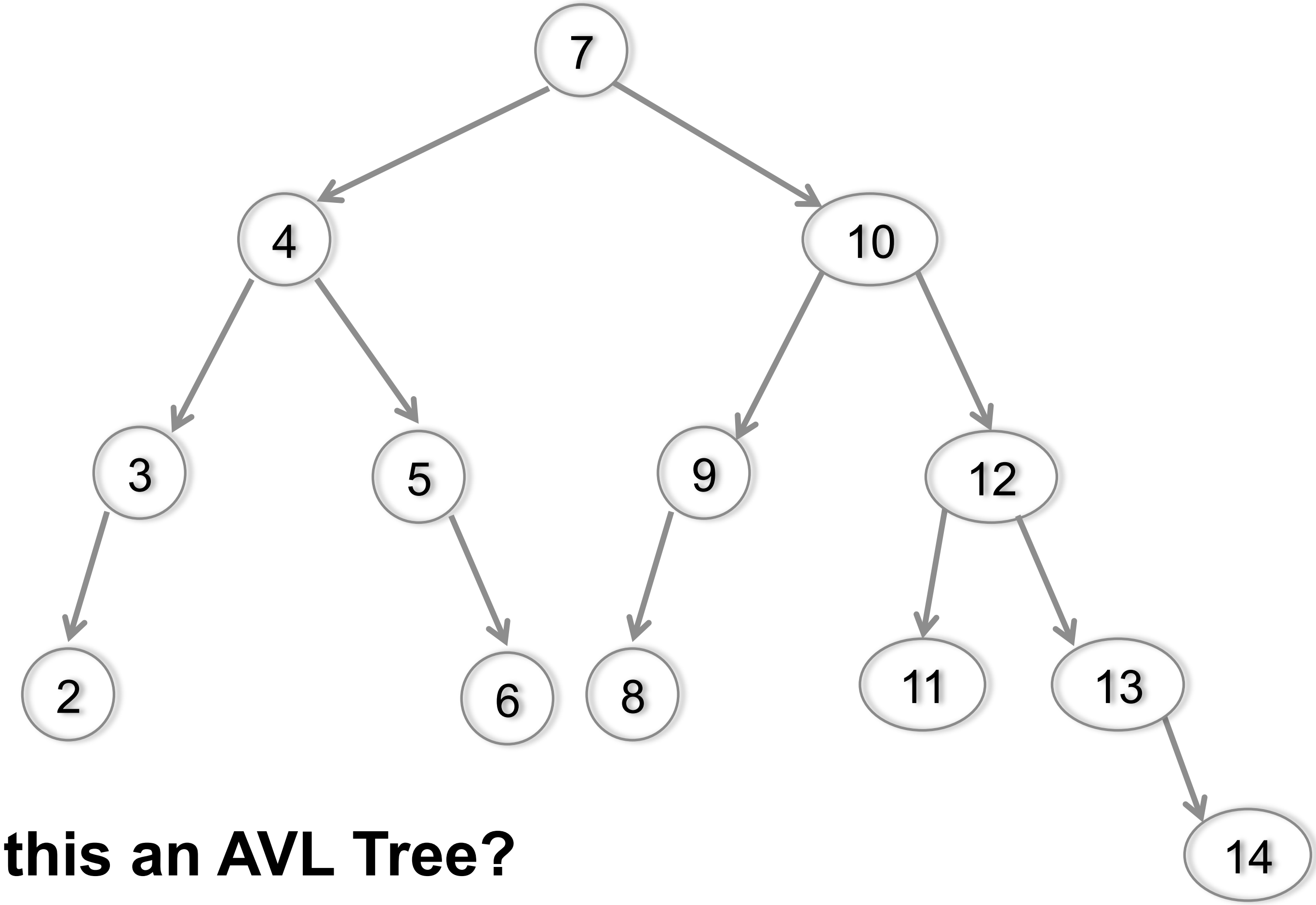
- **New property**
  - If Abs(height(left) – height(right)) is balance
  - We can only enforce if this is <=1
  - That is, the height left and right subtrees can differ by at most one
  - Still must preserve this for every node!
- **This is the AVL property**
- **AVL Trees are Binary Search Trees that have the AVL property**
  - They have worst case O(log n) find!

REVIEW



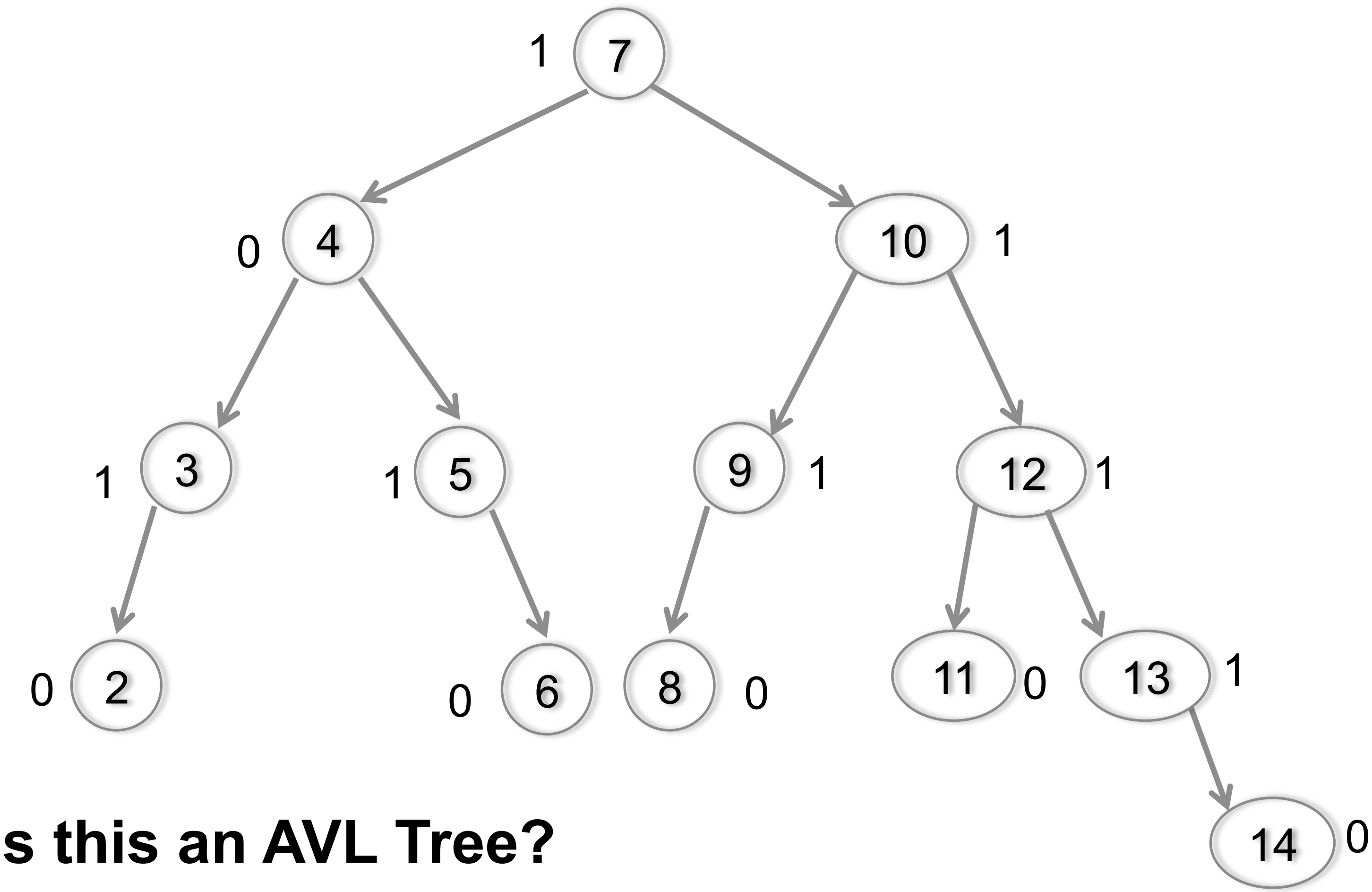
- **Is this an AVL Tree?**

REVIEW



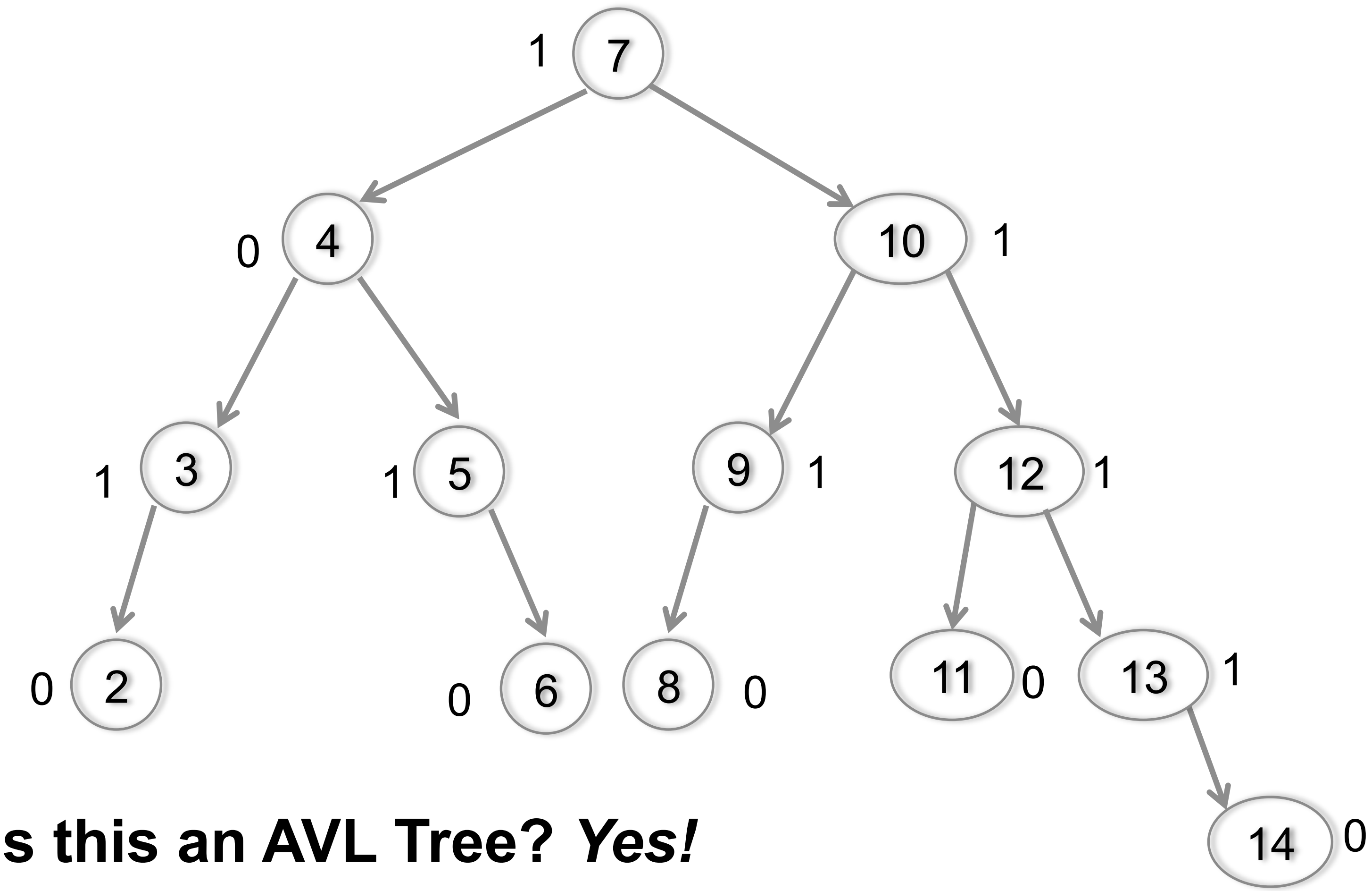
- **Is this an AVL Tree?**
  - Calculate balance for each node

REVIEW



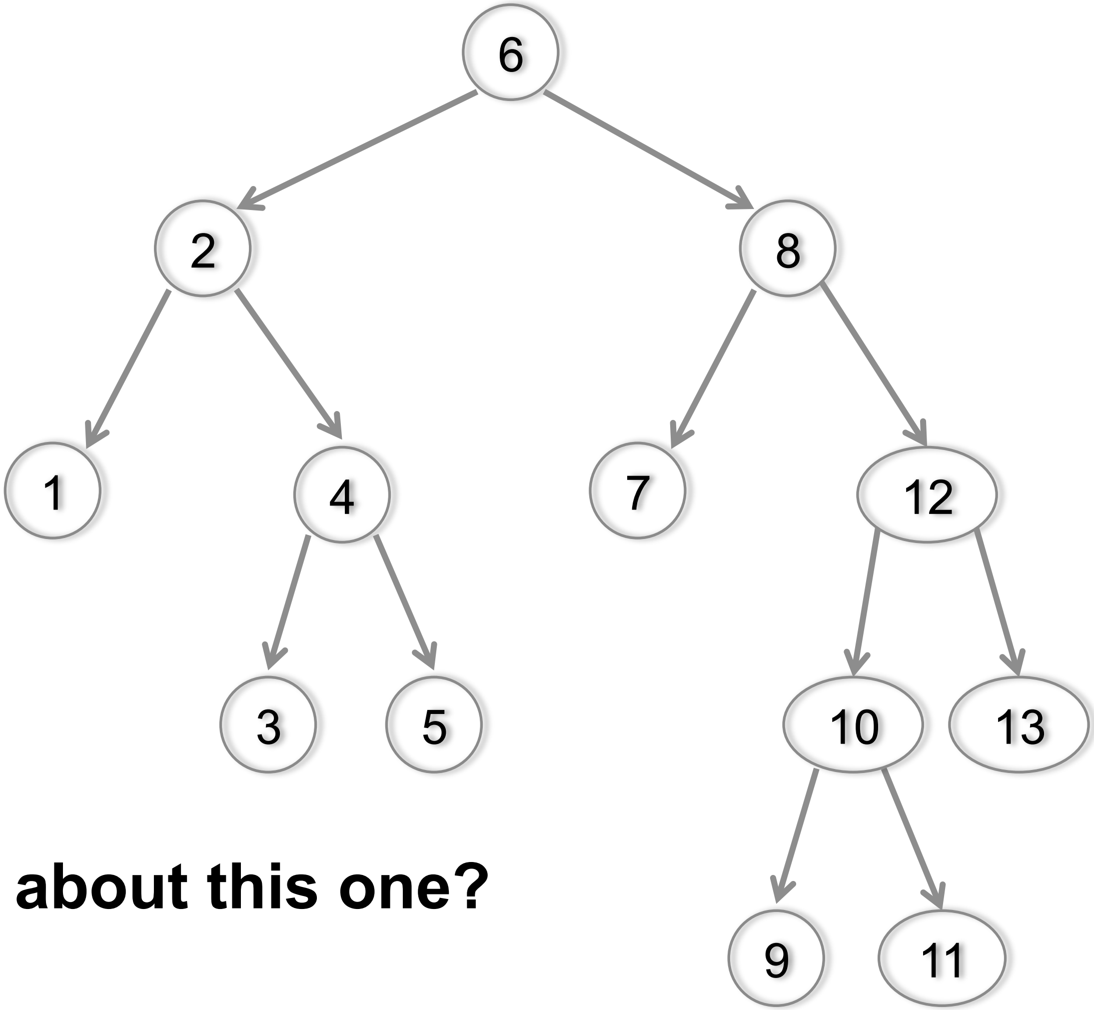
- **Is this an AVL Tree?**
  - Calculate balance for each node

REVIEW



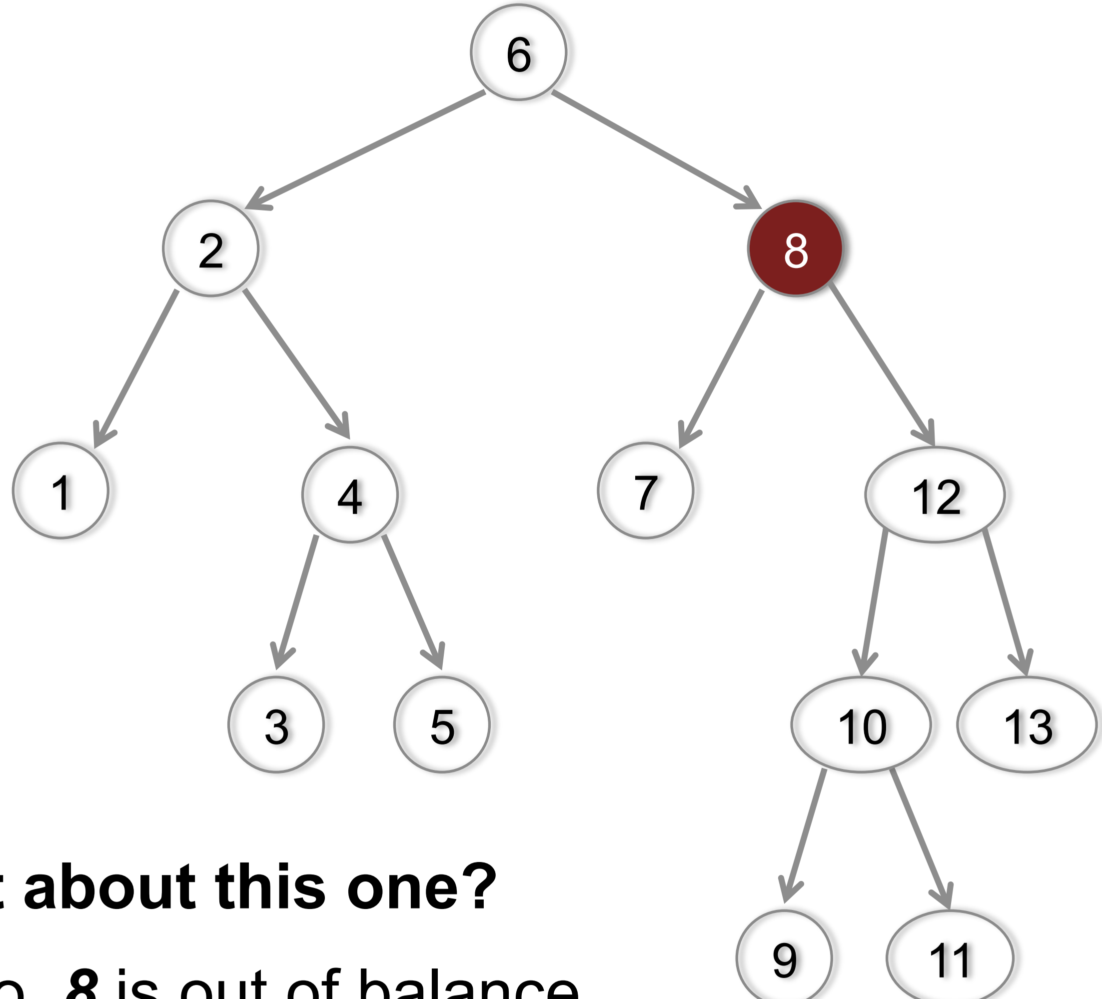
- **Is this an AVL Tree? *Yes!***
  - Calculate balance for each node

REVIEW



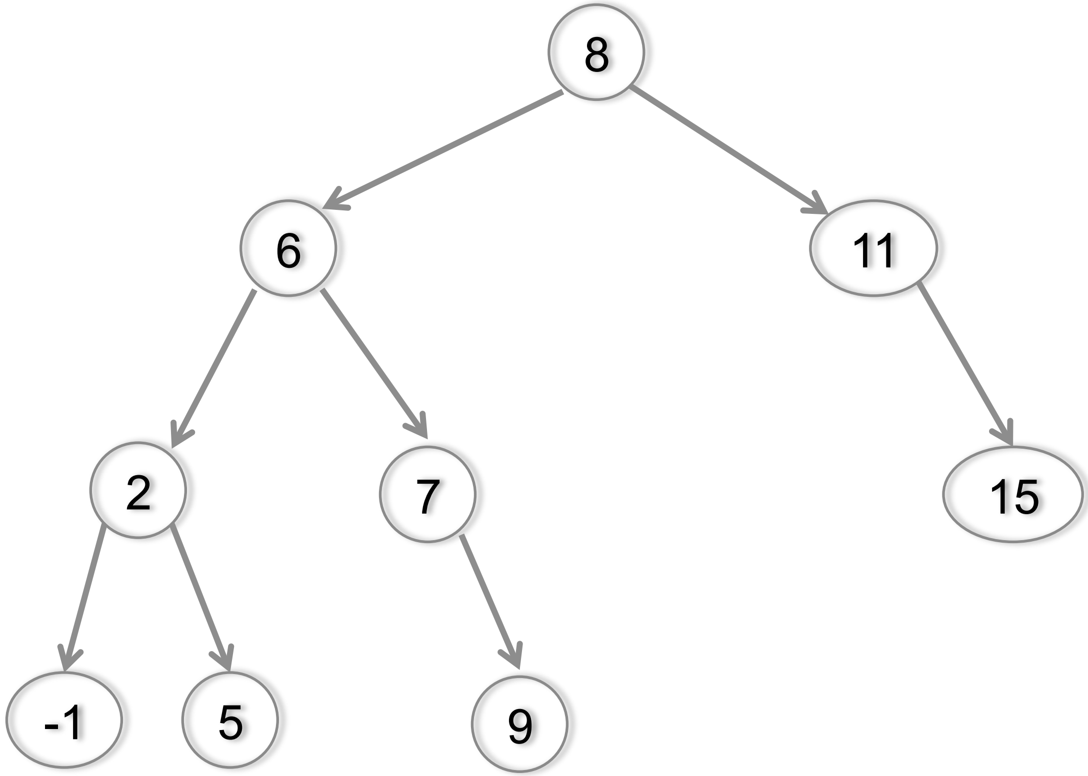
- **What about this one?**

REVIEW



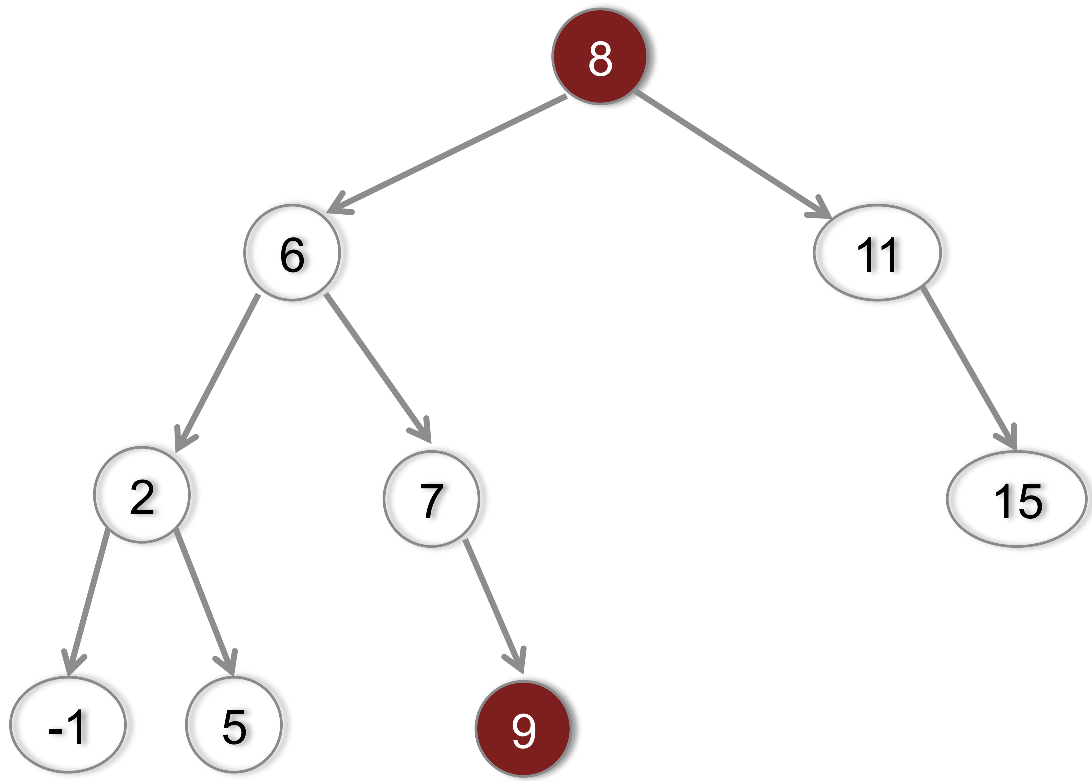
- **What about this one?**
  - No, ***8*** is out of balance

REVIEW



- **Is this an AVL Tree?**

REVIEW



- **Is this an AVL Tree?**
  - No, AVL trees must still maintain Binary Search

AVL OPERATIONS

- Since AVL trees are also BST trees, they should support the same functionality

AVL OPERATIONS

- **Since AVL trees are also BST trees, they should support the same functionality**
  - Insert(key k, value v)
  - Find(key k)
  - Delete(key k)

AVL OPERATIONS

- **Since AVL trees are also BST trees, they should support the same functionality**
  - Insert(key k, value v)
  - *Find(key k): Same as BST!*
  - Delete(key k)

AVL OPERATIONS

- **Since AVL trees are also BST trees, they should support the same functionality**
  - Insert(key k, value v)
  - *Find(key k): Same as BST!*
  - *Delete(key k)*: *Not presented in this course*

AVL OPERATIONS

- **Since AVL trees are also BST trees, they should support the same functionality**
  - Insert(key k, value v)
  - *Find(key k): Same as BST!*
  - *Delete(key k)*: *Not presented in this course*
- **For insert, we should maintain AVL property as we build**

AVL OPERATIONS

- **Since AVL trees are also BST trees, they should support the same functionality**
  - Insert(key k, value v)
  - *Find(key k): Same as BST!*
  - *Delete(key k)*: *Not presented in this course*
- **For insert, we should maintain AVL property as we build**

AVL OPERATIONS

- **Insert(key k, value v):**

AVL OPERATIONS

- **Insert(key k, value v):**
  - Insert the key value pair into the dictionary

AVL OPERATIONS

- **Insert(key k, value v):**
  - Insert the key value pair into the dictionary
  - Verify that balance is maintained

AVL OPERATIONS

- **Insert(key k, value v):**
  - Insert the key value pair into the dictionary
  - Verify that balance is maintained
  - If not, correct the tree

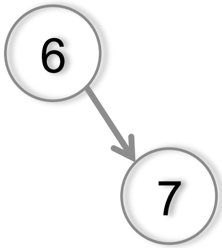
AVL OPERATIONS

- **Insert(key k, value v):**
  - Insert the key value pair into the dictionary
  - Verify that balance is maintained
  - If not, correct the tree
- **How do we correct the tree?**

AVL INSERT

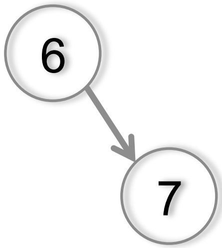
- **Start with the single root**

AVL INSERT



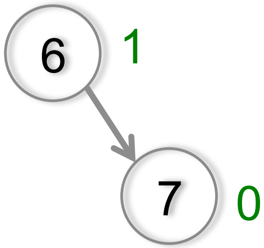
- **Add 7 to the tree**

AVL INSERT



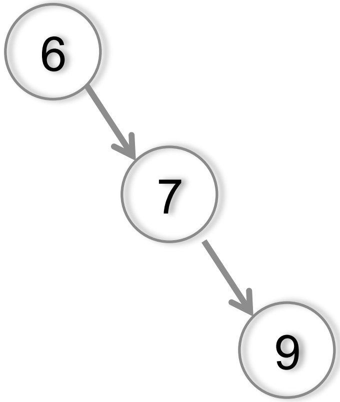
- **Add 7 to the tree. Is balance preserved?**

AVL INSERT



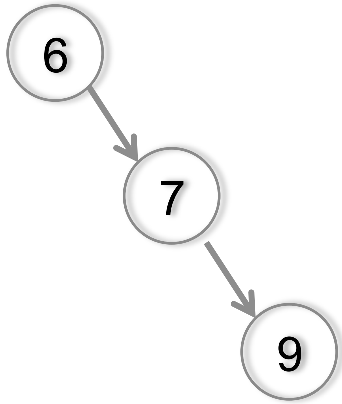
- **Add 7 to the tree. Is balance preserved?**
  - Yes

AVL INSERT



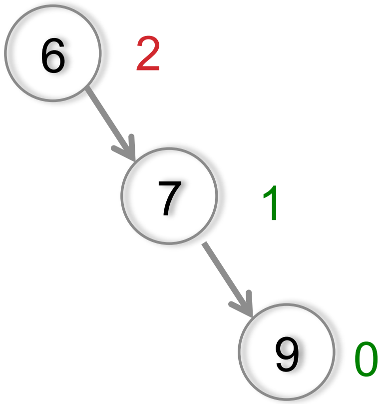
- **Add 9 to the tree**

AVL INSERT



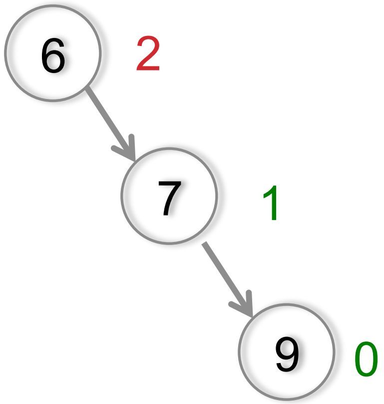
- **Add 9 to the tree. Is balance preserved?**

AVL INSERT



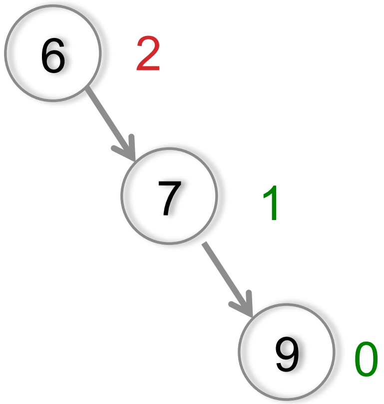
- **Add 9 to the tree. Is balance preserved?**
  - No.

AVL INSERT



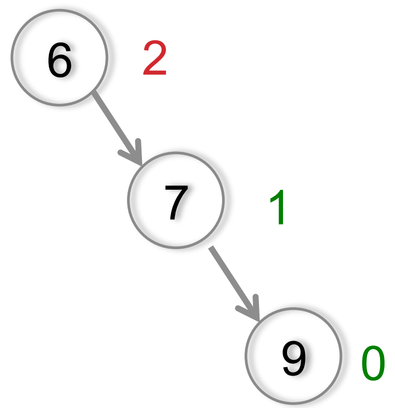
- **How do we correct this imbalance?**

AVL INSERT



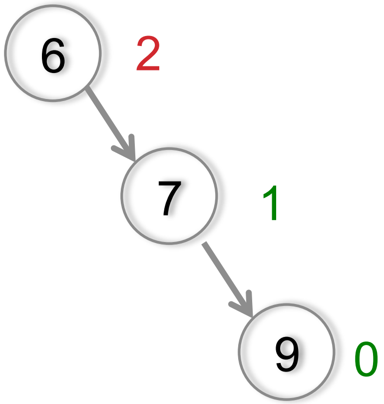
- **How do we correct this imbalance?**
  - Important to preserve binary search

AVL INSERT



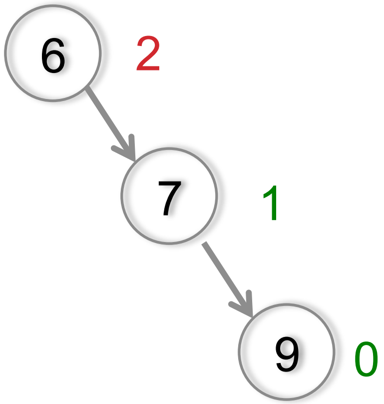
- **How do we correct this imbalance?**
  - Important to preserve binary search

AVL INSERT

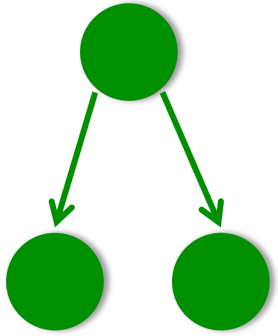


- **What shape do we want?**

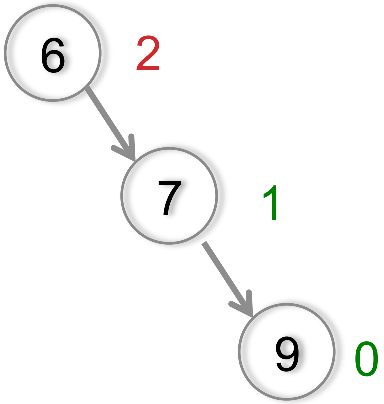
AVL INSERT



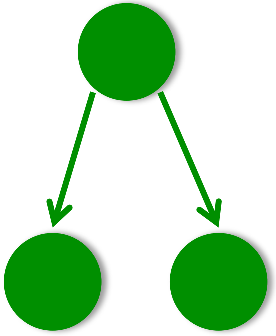
- **What shape do we want?**



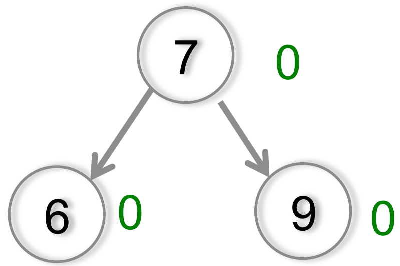
AVL INSERT



- **What shape do we want?**
  - What then do we have as the root?



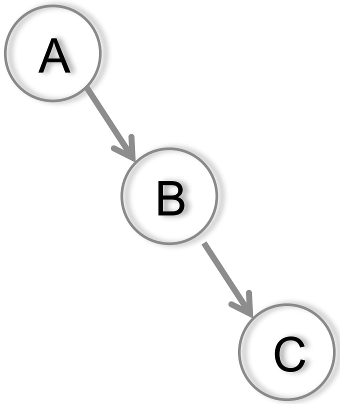
AVL INSERT



- **Since 7 must be the root, we "rotate" that node into position.**

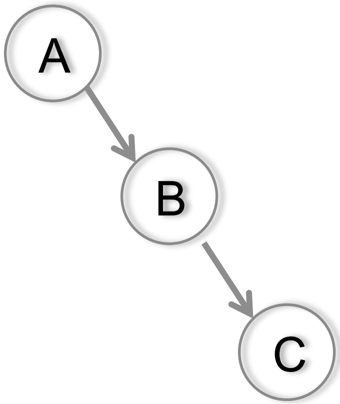
AVL “ROTATION”

- **To correct this case:**
  - B must become the root



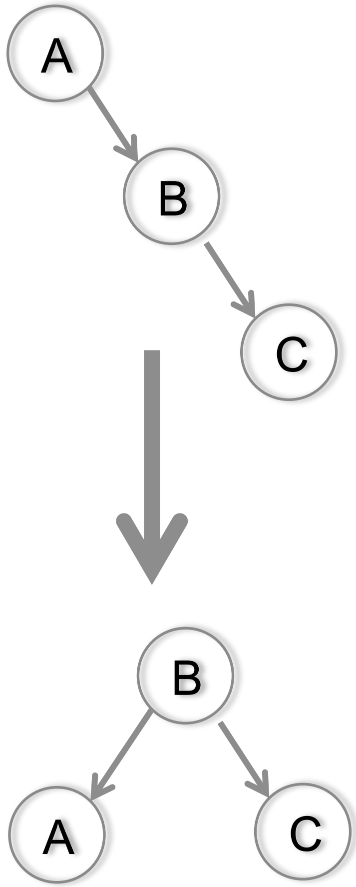
AVL "ROTATION"

- **To correct this case:**
  - B must become the root
  - We rotate B to the root position



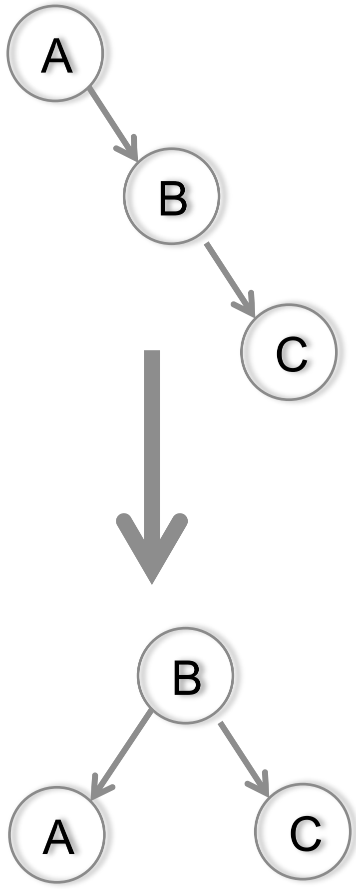
AVL "ROTATION"

- **To correct this case:**
  - B must become the root
  - We rotate B to the root position
  - A becomes the left child of B



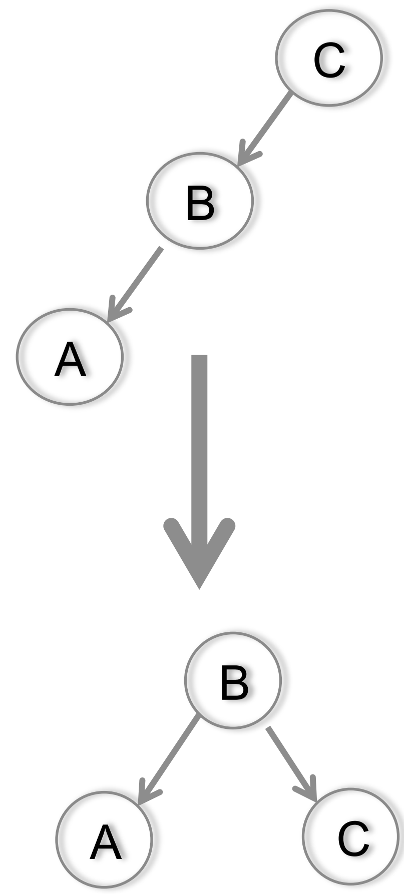
AVL “ROTATION”

- **To correct this case:**
  - B must become the root
  - We rotate B to the root position
  - A becomes the left child of B
  - This is called the “left rotation”



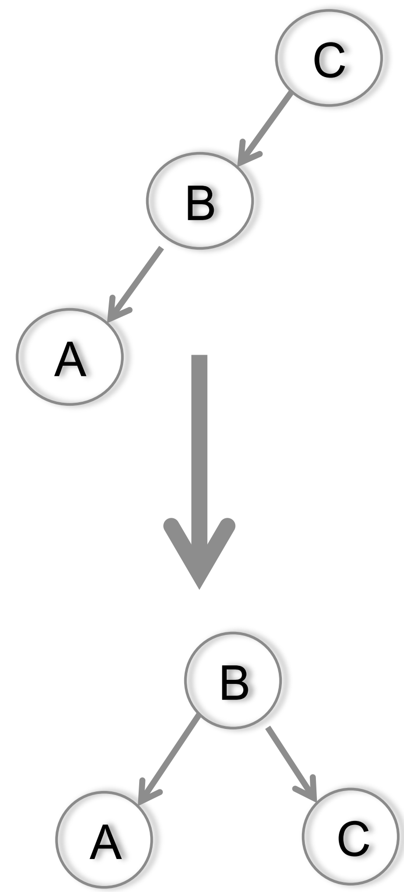
AVL "ROTATION"

- **Right rotation**



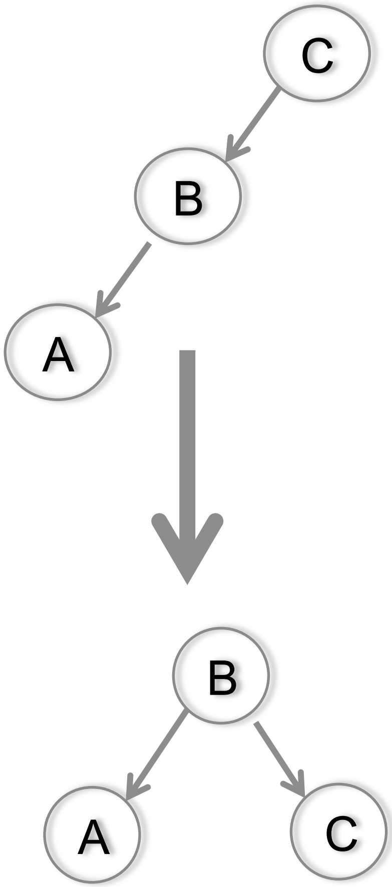
AVL "ROTATION"

- **Right rotation**
  - Symmetric concept



AVL "ROTATION"

- **Right rotation**
  - Symmetric concept
  - B must become the new root



AVL "ROTATION"

- These are the "single" rotations

AVL “ROTATION”

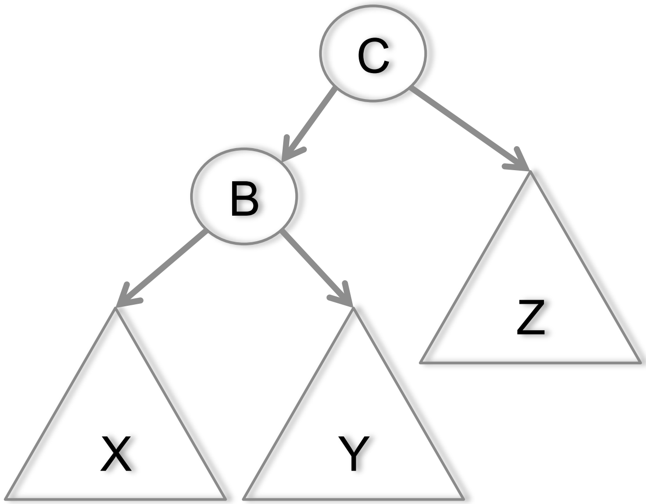
- **These are the “single” rotations**
  - In general, this rotation occurs when an addition is made to the right-right or left-left grandchild

AVL “ROTATION”

- **These are the “single” rotations**
  - In general, this rotation occurs when an addition is made to the right-right or left-left grandchild
  - **The balance might not be off on the parent! An insert might upset balance up the tree**

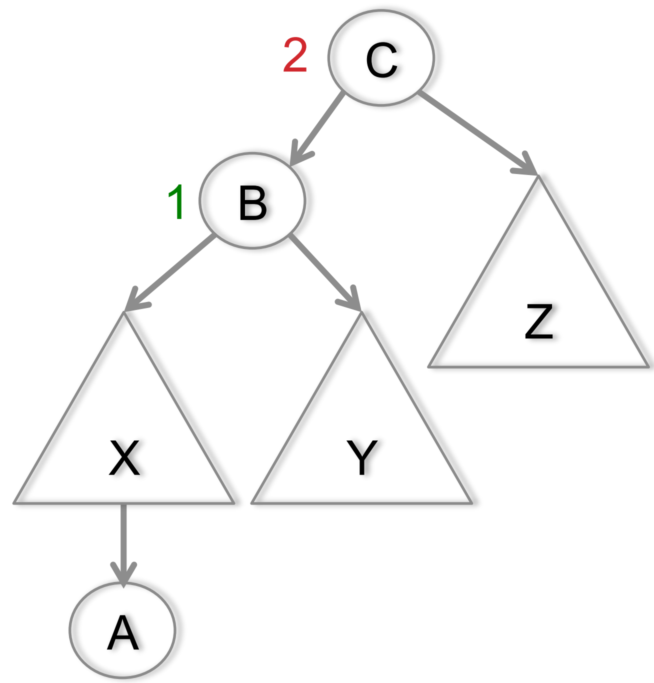
AVL "ROTATION"

- **General case**
  - Suppose this tree is balanced, {X,Y,Z} all have the same height



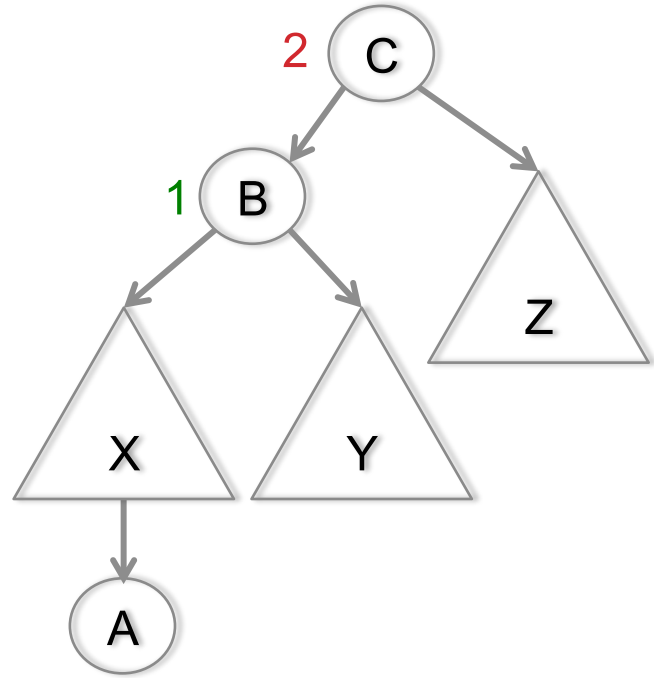
AVL "ROTATION"

- **General case**
  - Suppose this tree is balanced, {X,Y,Z} all have the same height
  - Adding A, puts C out of balance



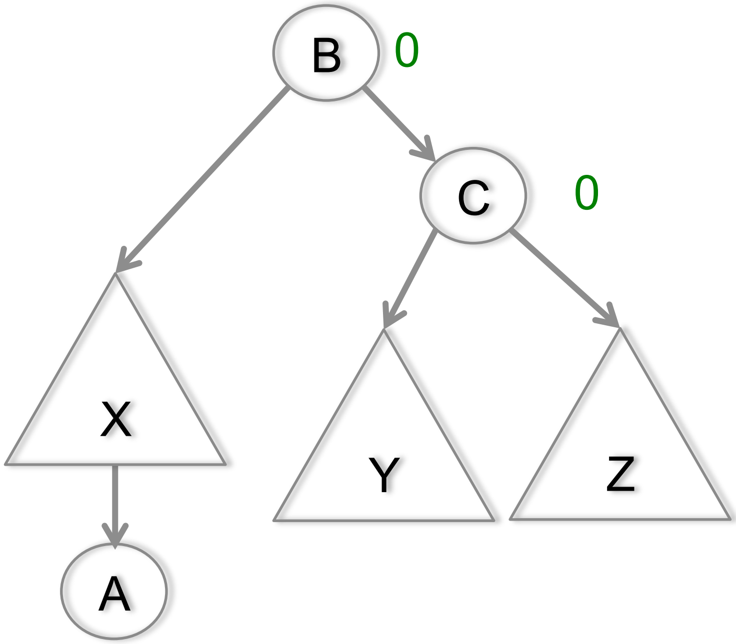
AVL "ROTATION"

- **General case**
  - Suppose this tree is balanced, {X,Y,Z} all have the same height
  - Adding A, puts C out of balance
  - Rotate B up and pass the Y subtree to C



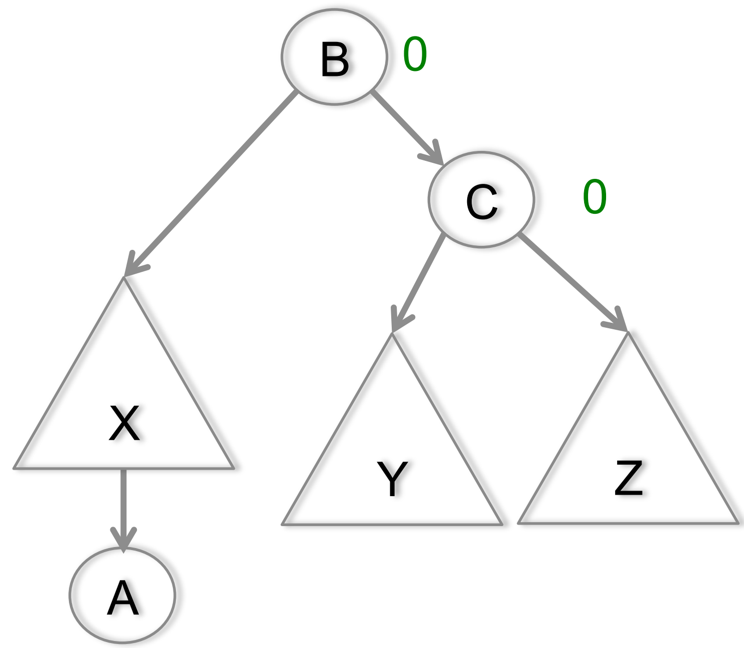
AVL "ROTATION"

- **General case**
  - Suppose this tree is balanced, {X,Y,Z} all have the same height
  - Adding A, puts C out of balance
  - Rotate B up and pass the Y subtree to C

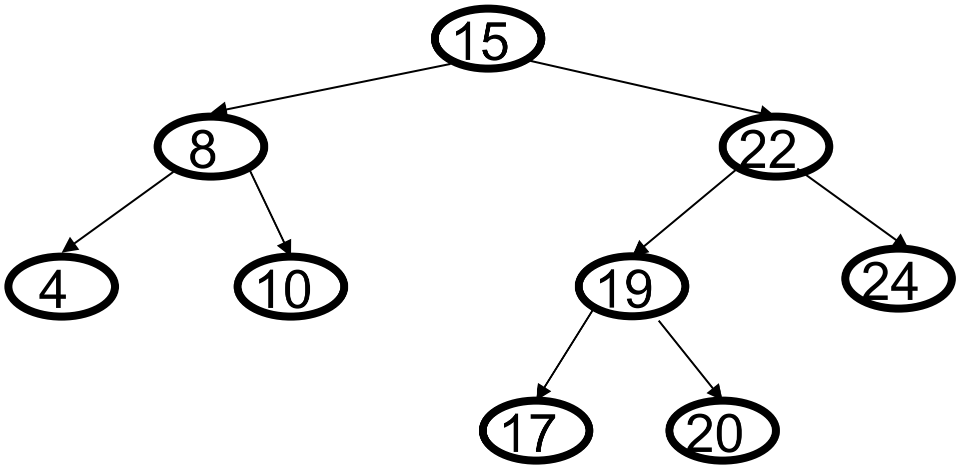


AVL "ROTATION"

- **General case**
  - Suppose this tree is balanced, {X,Y,Z} all have the same height
  - Adding A, puts C out of balance
  - Rotate B up and pass the Y subtree to C
  - **Perform this rotation at the lowest point of imbalance**

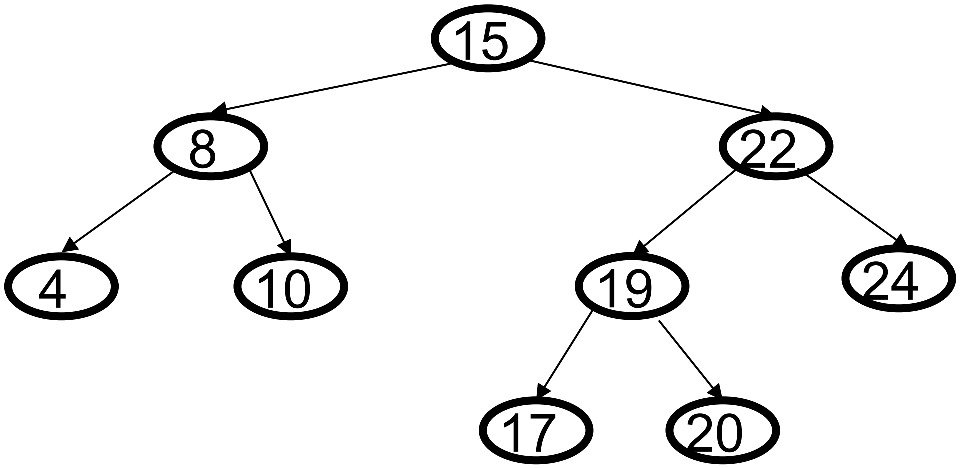


SINGLE ROTATION EXAMPLE



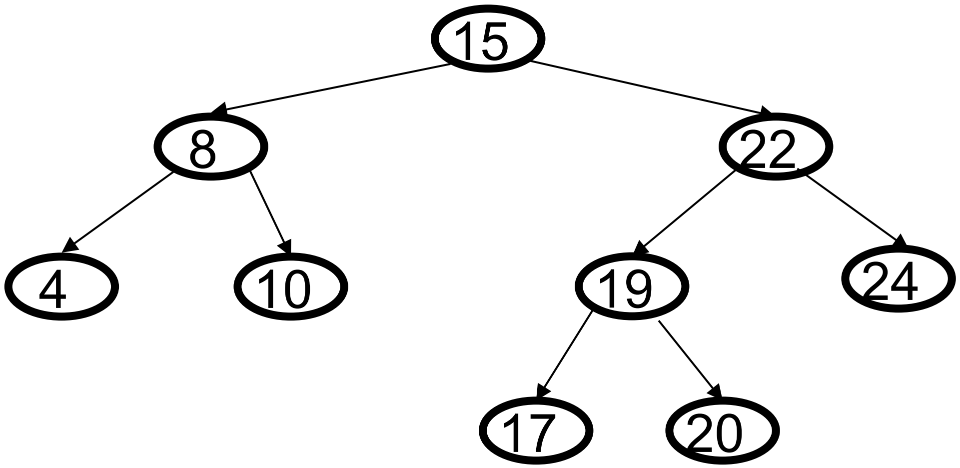
- **Consider the above tree**

SINGLE ROTATION EXAMPLE



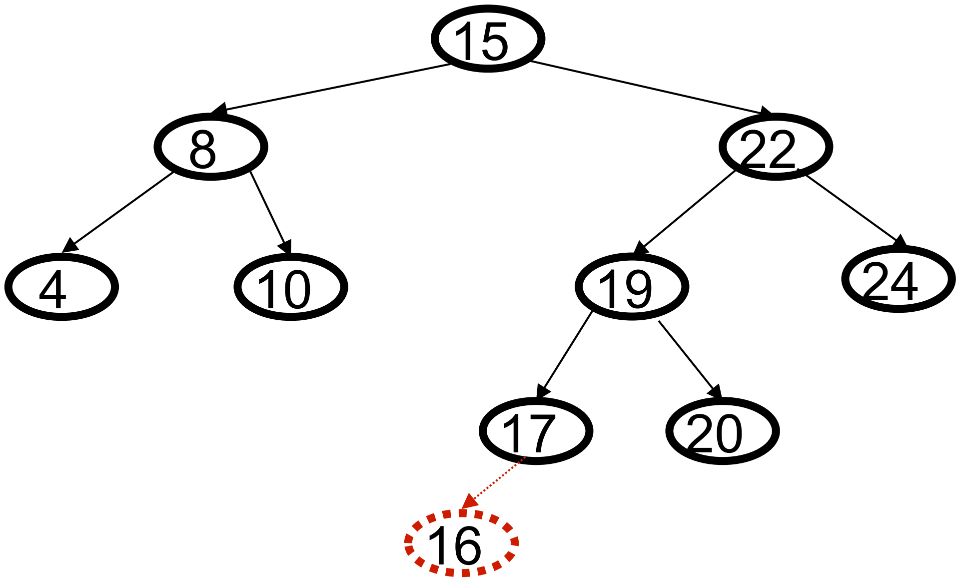
- **Consider the above tree**
  - Is it an AVL tree?

SINGLE ROTATION EXAMPLE



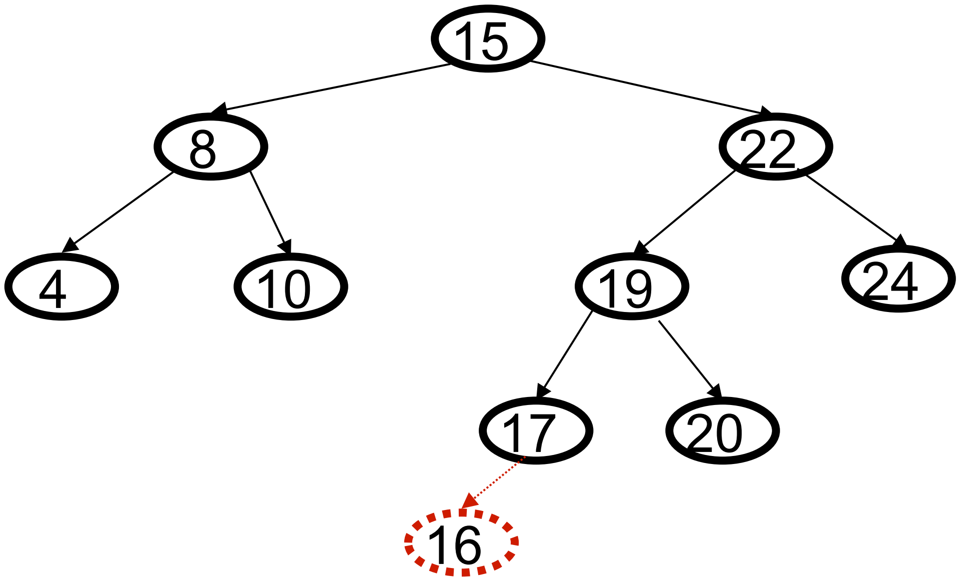
- **Consider the above tree**
  - Is it an AVL tree? *Yes*

SINGLE ROTATION EXAMPLE



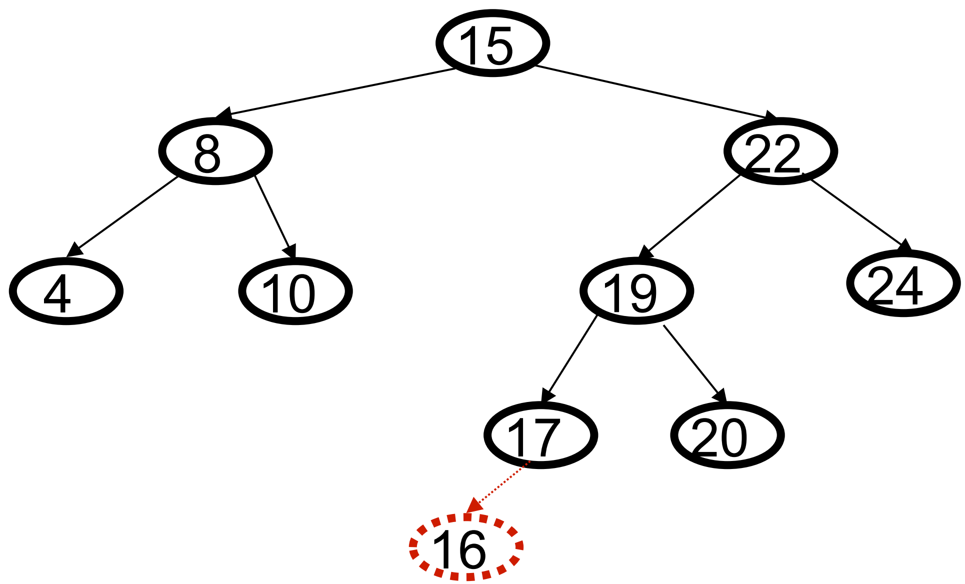
- **Add 16 to the tree**

SINGLE ROTATION EXAMPLE



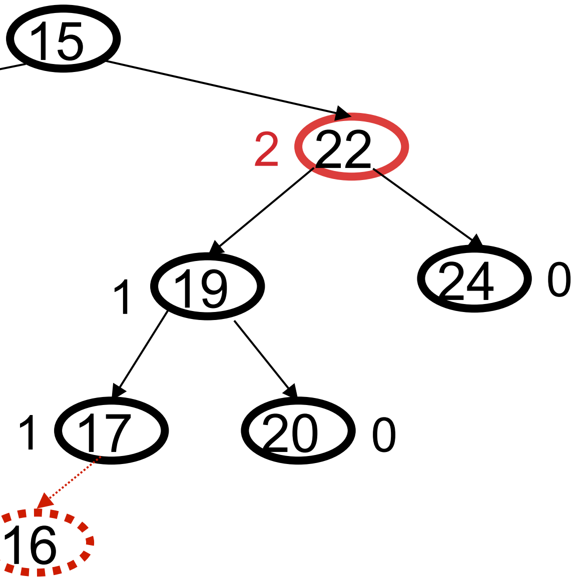
- **Add 16 to the tree**
  - Is it unbalanced now?

SINGLE ROTATION EXAMPLE



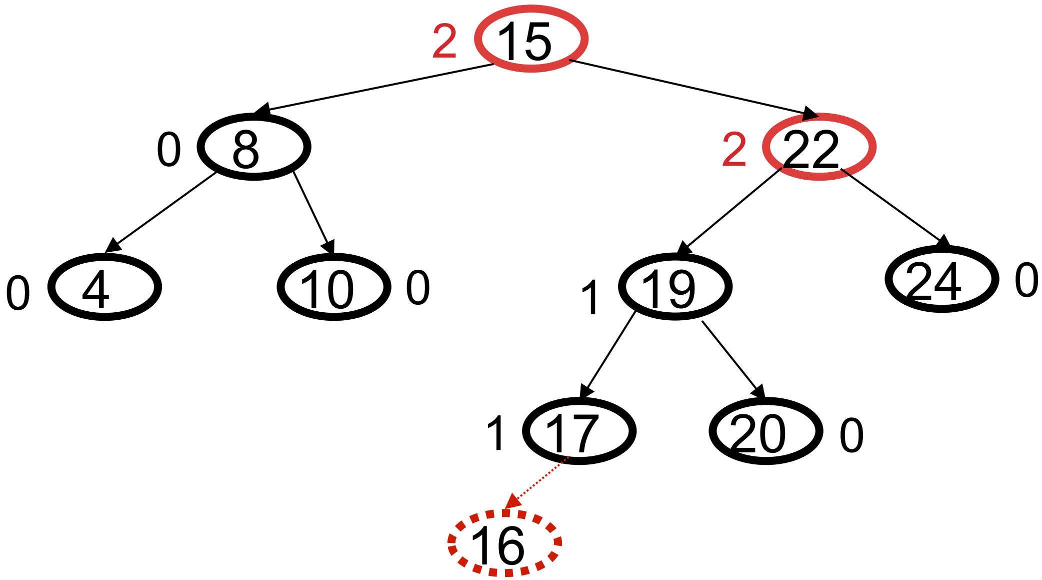
- **Add 16 to the tree**
  - Is it unbalanced now? Where?

SINGLE ROTATION EXAMPLE



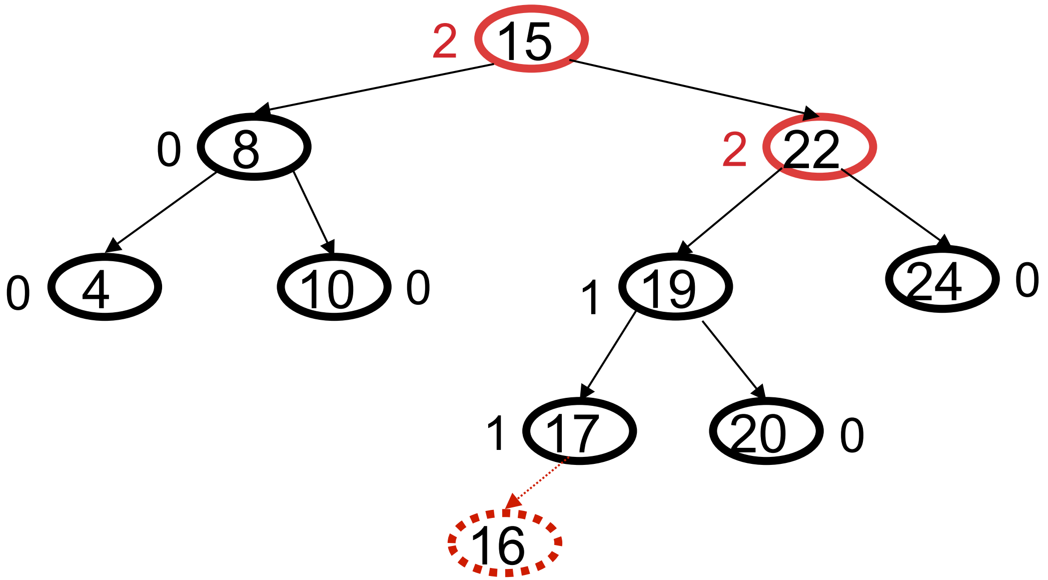
- **Add 16 to the tree**
  - Is it unbalanced now? Where? **22**

SINGLE ROTATION EXAMPLE



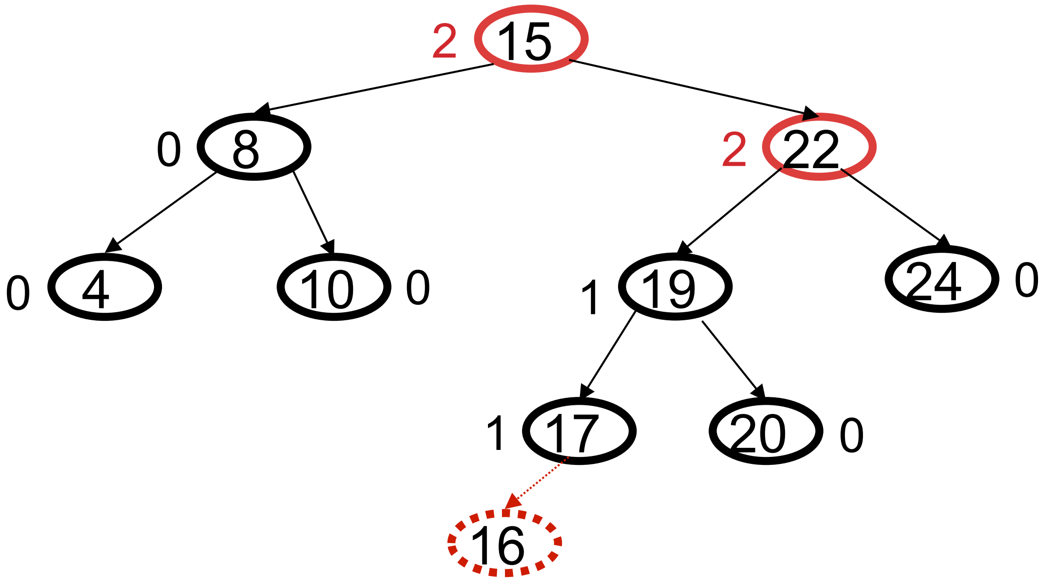
- **Add 16 to the tree**
  - Is it unbalanced now? Where? **22**
  - Also at 15, but we choose the lowest point

SINGLE ROTATION EXAMPLE



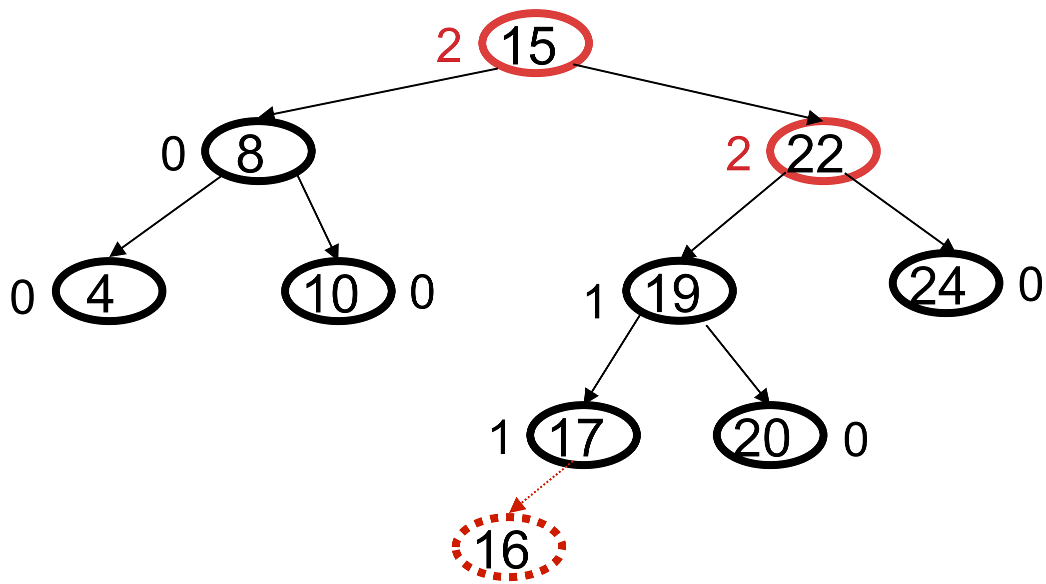
- **Perform the rotation around 22**

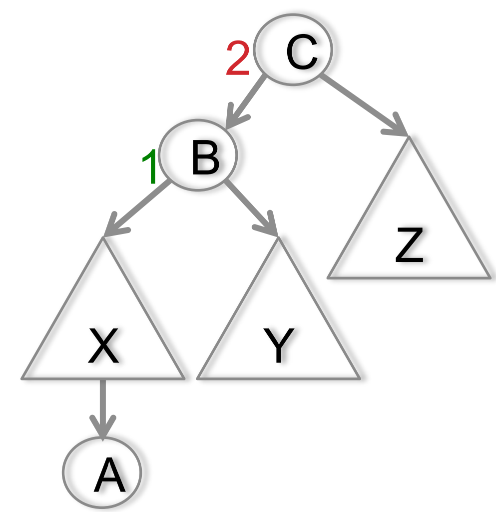
SINGLE ROTATION EXAMPLE



- **Perform the rotation around 22**
  - What rotation takes place?

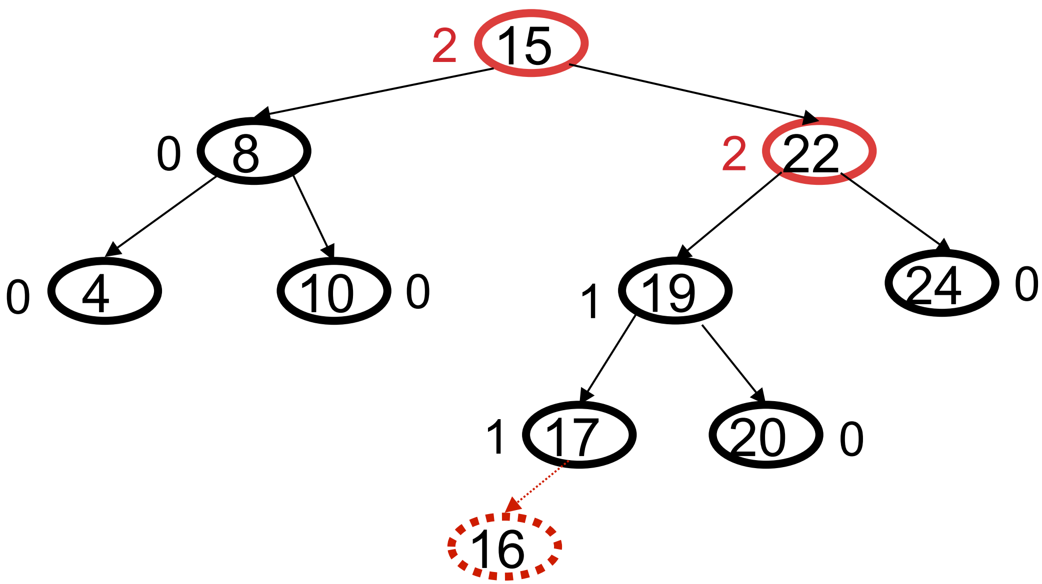
SINGLE ROTATION EXAMPLE





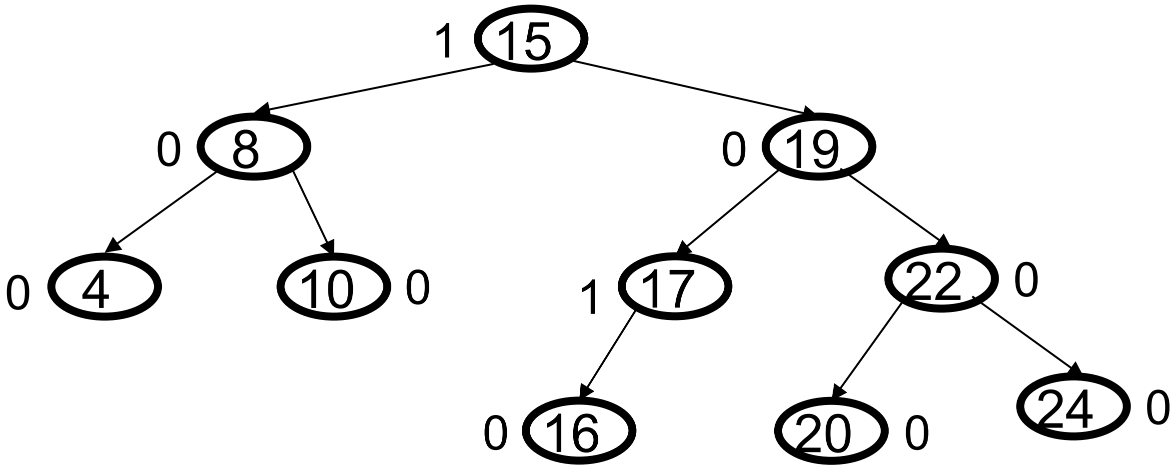
- **Perform the rotation around 22**
  - What rotation takes place?

SINGLE ROTATION EXAMPLE



- **Perform the rotation around 22**
  - What rotation takes place?
  - What is the resulting tree?

SINGLE ROTATION EXAMPLE



- **19 must move up to where 22 was**
  - 20 changes parents
  - Balances are recomputed throughout the tree

AVL "ROTATION"

- These two rotations (right-right and left-left) are symmetric and can be solved the same way

AVL “ROTATION”

- **These two rotations (right-right and left-left) are symmetric and can be solved the same way**
  - Named by the location of the added node relative to the unbalanced node