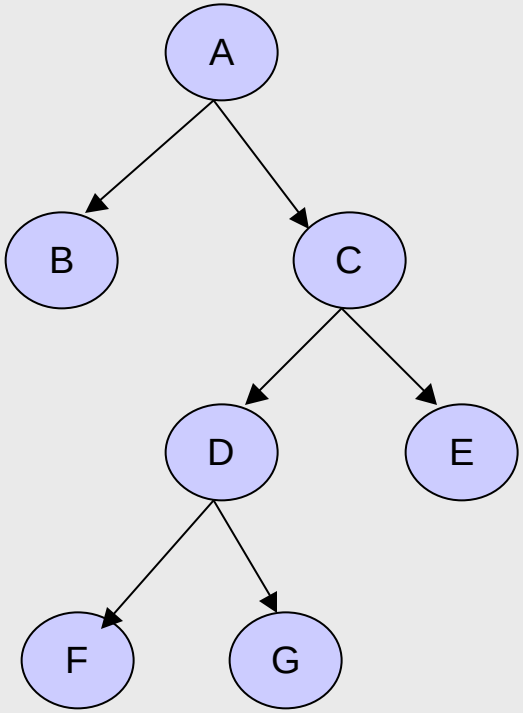
COP 3502: Computer Science I
Spring 2004

– Note Set 18 –
Binary Trees – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

A Closer Look at the Inorder Traversal



- The inorder traversal of the tree on the left is:

  B, A, F, D, G, C, E

- Notice that before "visiting" the root of any subtree we've proceeded as far down the left subtree as possible.
- This is called a depth-first traversal.
- The preorder and postorder traversals are also depth-first traversals, in that one subtree is explored to its logical end before any nodes in the other subtree are ever visited.

A Closer Look at the Inorder Traversal (cont.)

- For binary trees, the most common form of traversal is one of the depth-first traversals.
- Depending on the application, the preorder, inorder, and postorder traversals are equally useful.
- It is also possible to traverse a binary tree using a level-order traversal. In a level-order traversal, all of the nodes on a given level of the tree are visited before any node on the next deeper (closer to the leaves) level is visited.
- A level-order traversal of the tree shown on page 2 would be: A, B, C, D, E, F, G. We'll examine level-order traversals in more detail later.

Traversal Applications

- Preorder traversals are useful in tree cloning operations, because you encounter the root node of a subtree prior to traversing the children, so the structure of the tree is easy to recreate.
- Inorder traversals are most useful in binary search tree applications. For example, given a binary search tree (we'll see the definition of such a tree shortly), an inorder traversal of the search tree will print the values in the nodes in ascending order.
- Postorder traversals are most common with expression trees. A postorder traversal of an expression tree produces the postfix form of an infix expression. Postorder traversals are also used for expression evaluation. We'll look at this application next.

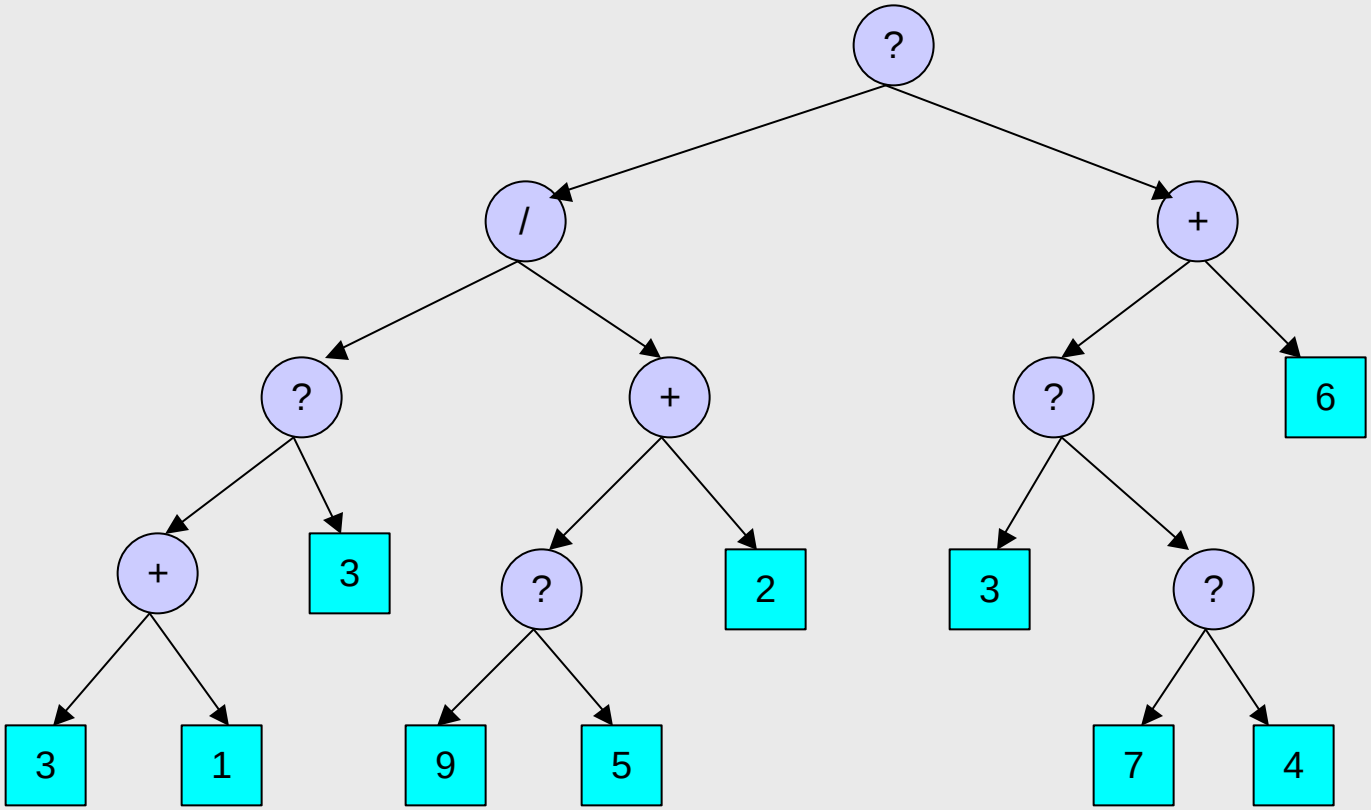
Applications for Binary Trees: Expression Trees

- When we dealt with stacks, we saw an algorithm that converted an infix expression into its postfix representation using a stack. We saw that the postfix form of the expression was easier to evaluate than its infix form since every operation was immediately preceded by its operands.
- An arithmetic expression is often represented by a binary tree. Such a binary tree is known as an expression tree.
- An expression tree is a binary tree representing an arithmetic expression where the external nodes (leaf nodes) of the tree represent variables or constants (the operands) and the internal nodes (non-leaf nodes) represent the operations.

Expression Trees

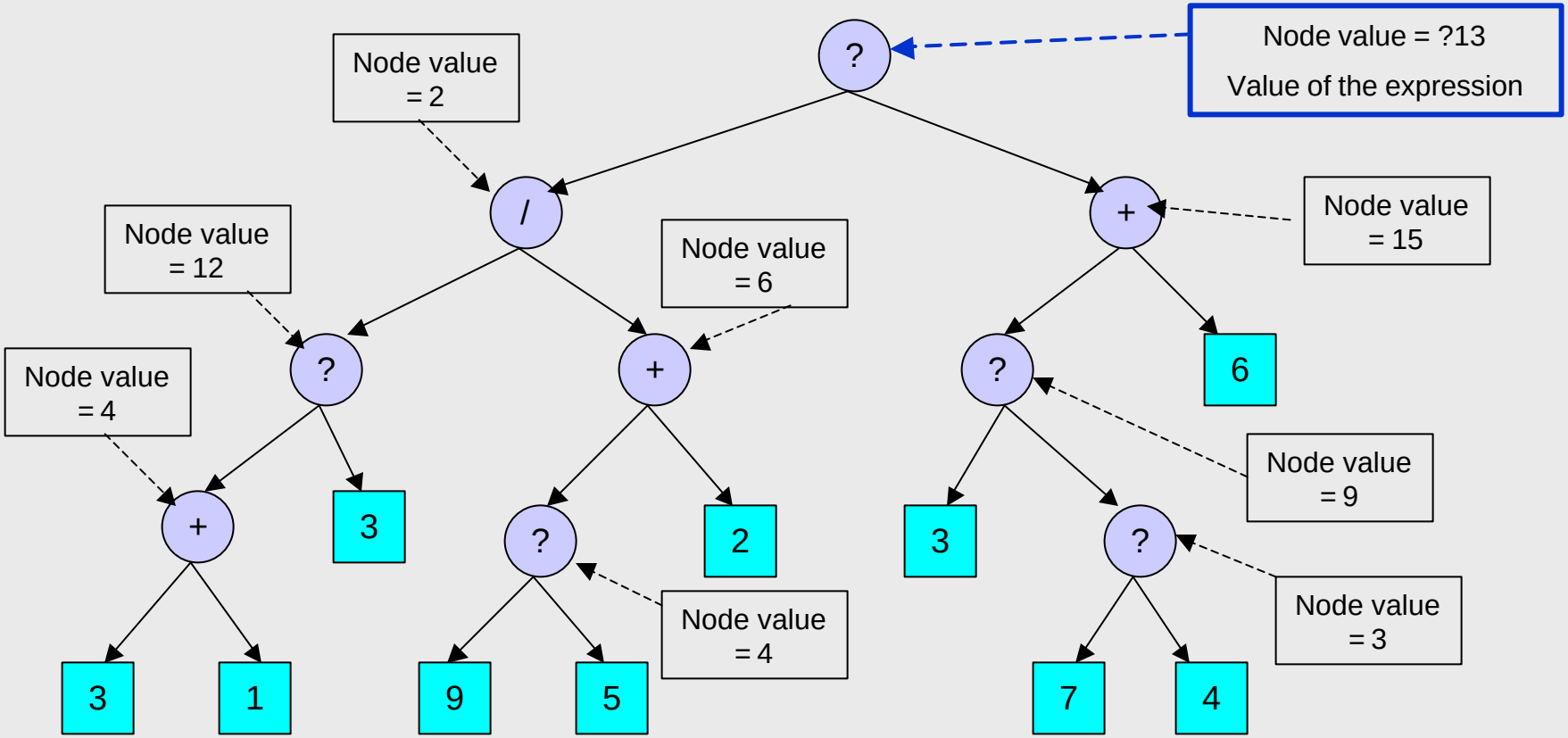
- Each node in an expression tree has a value associated with it:
  - If the node is a leaf node (external node), then its value is that of its corresponding variable or constant.
  - If the node is an internal node (non-leaf node), then its value is defined by applying its corresponding operation to the values of its children.
- In a binary tree, if every node in the tree has either 0 or 2 children the tree is a proper binary tree, otherwise it is considered an improper binary tree. For expressions which involve only binary operations, the expression tree will be proper. However, if unary plus/minus operations appear in the expression, the corresponding expression tree will be improper.

Expression Trees (cont.)



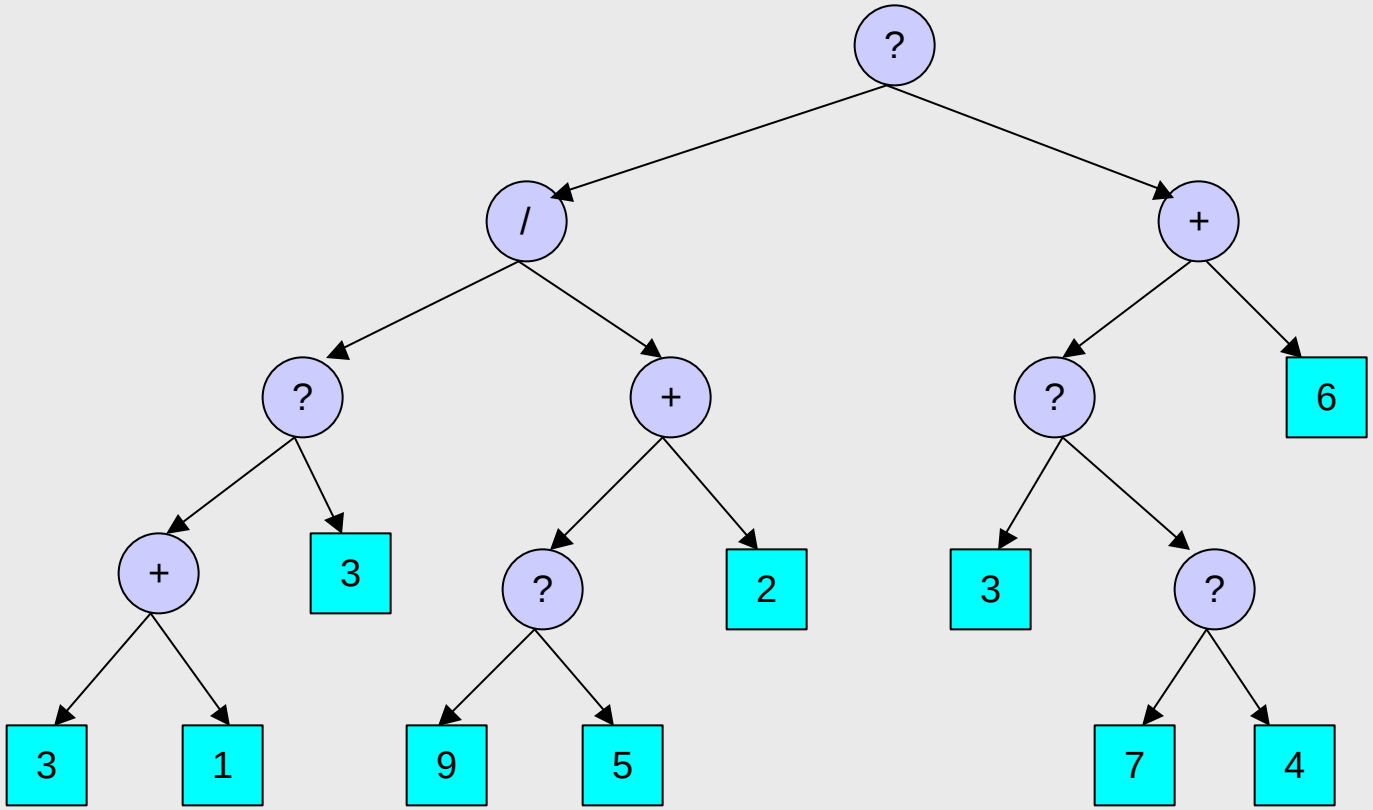
Expression tree for: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

Expression Trees (cont.)



Expression tree for: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

Expression Trees (cont.)

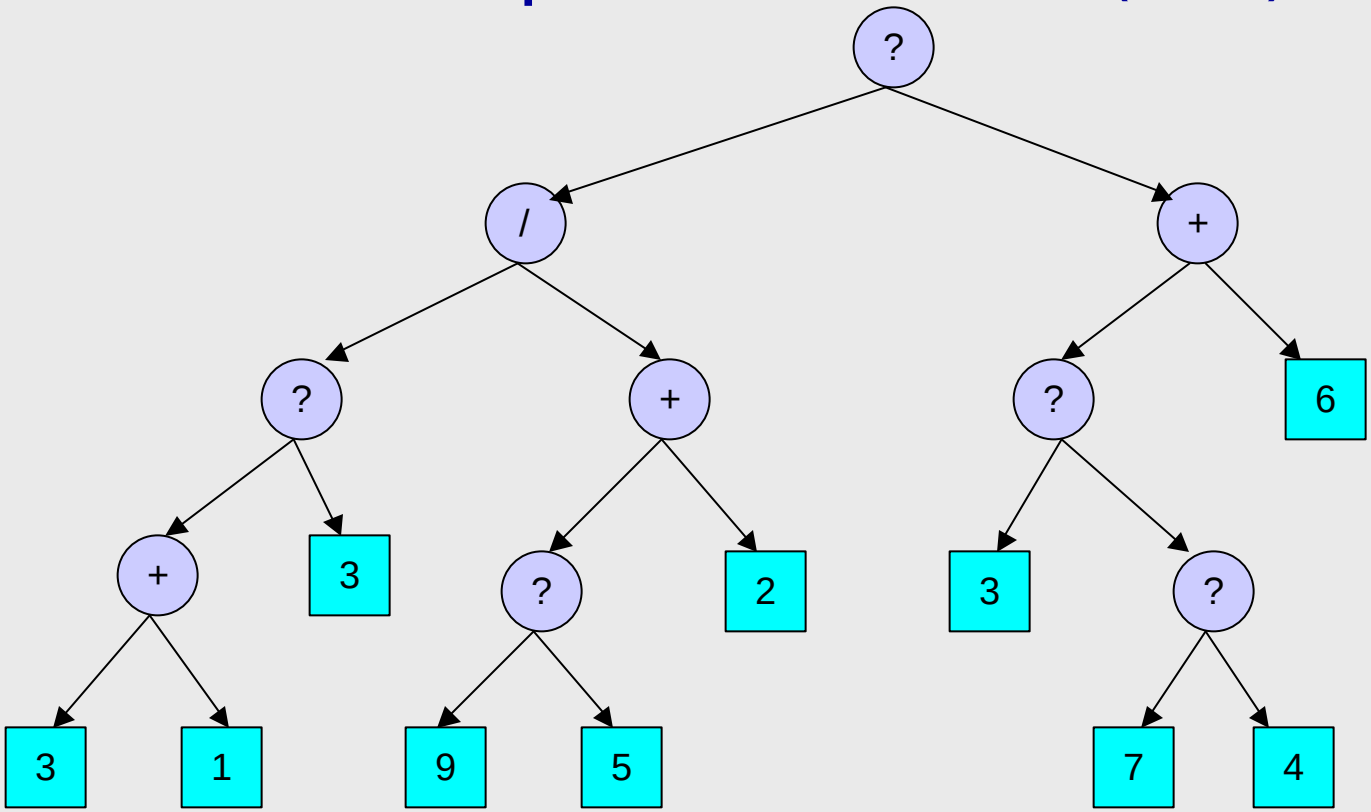


An inorder traversal of this tree give:

3 + 1 ? 3 / 9 – 5 + 2 – 3 ? 7 – 4 + 6, which except for the parenthesis is equal to:

((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

Expression Trees (cont.)



The expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

converted to postfix notation is: 3 1 + 3 ? 9 5 – 2 + / 3 7 4 – ? 6 + –

A postorder traversal of the tree also produces: 3 1 + 3 ? 9 5 – 2 + / 3 7 4 – ? 6 + –

Expression Trees (cont.)

- It is easy using a binary tree to convert an infix expression into its postfix form by simply performing a postorder traversal of the tree corresponding to the infix expression.
- Now what we need is an algorithm for converting an infix expression into an expression tree.
- It turns out that this is not a hard algorithm to develop. Consider the infix expression that we have been using:

  ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

  The tree is built from the bottom-up by "parsing" the fully parenthesized infix expression from left-to-right. Basically, the algorithm maintains a forest of trees via a stack. Although it is not crucial to understand how an expression tree is constructed (at least at this point in time – eventually you will learn the complete algorithm), I wanted to show you the basics of the construction process. This is shown on the next page.

Expression Trees (cont.)

- A tree node is created for every operand and operator encountered in the infix expression. A pointer is maintained to each of these nodes.

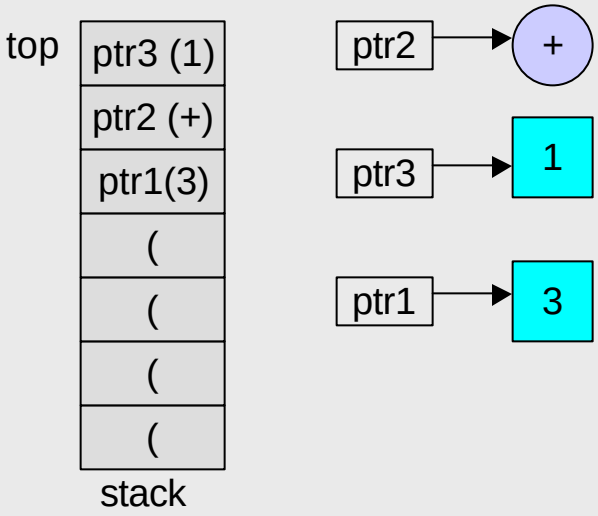
1. Push all open parentheses onto the stack.

2. Each time and operand or operator is encountered, create a node and push its pointer onto the stack.

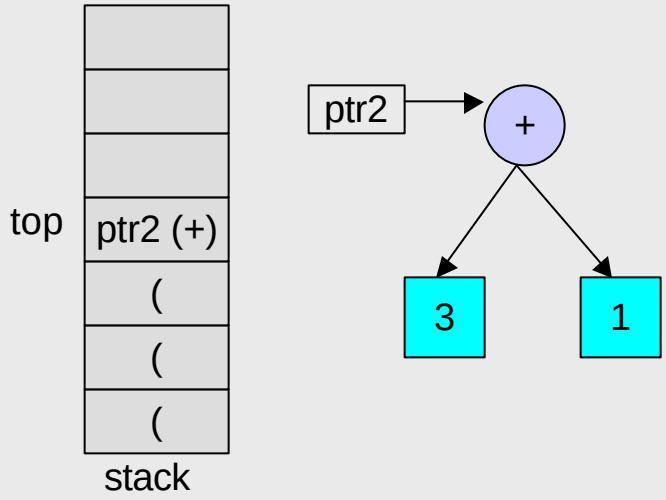
3. For each right parenthesis that is encountered do the following: pop the pointer to the right operand, pop the pointer to the operator, pop the pointer to the left operand, and pop the open parenthesis which is now on the top of the stack. Set the left child of the operator node to the left operand and the right child of the operator node to the right operand and push the pointer to the operator node onto the stack.

4. Repeat steps 1 through 3 until the entire expression has been scanned.

Expression Trees (cont.)



Stack and pointers just before encountering first close parenthesis



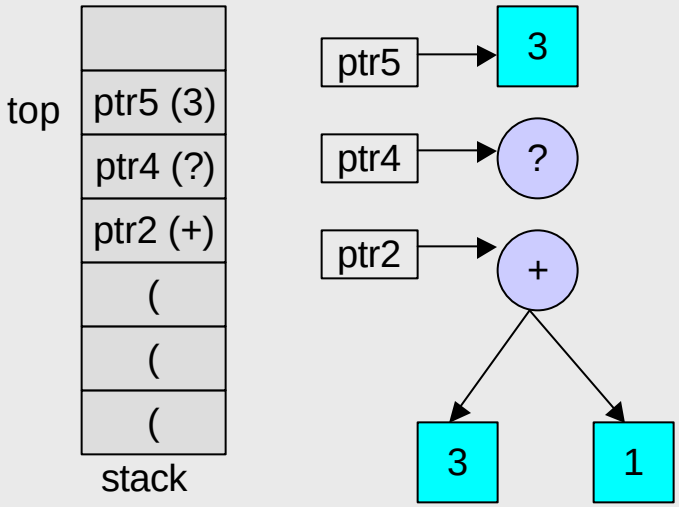
Stack and pointers just after encountering first close parenthesis

Here

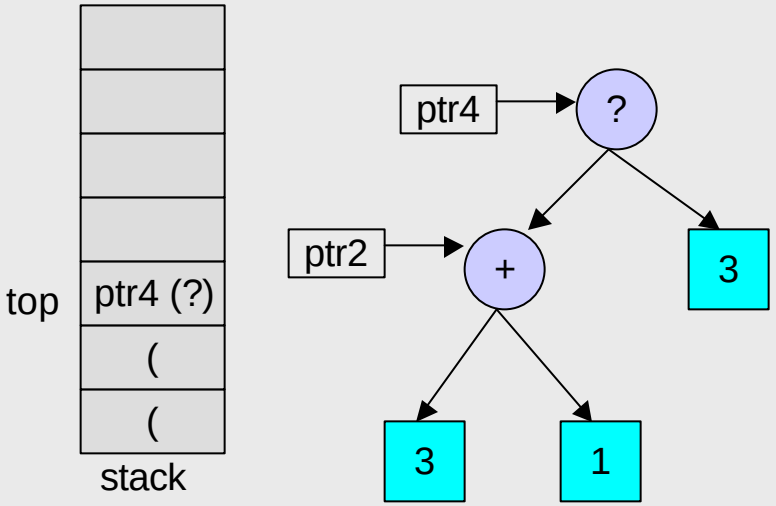
Here

The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

Expression Trees (cont.)



Stack and pointers just before encountering second close parenthesis

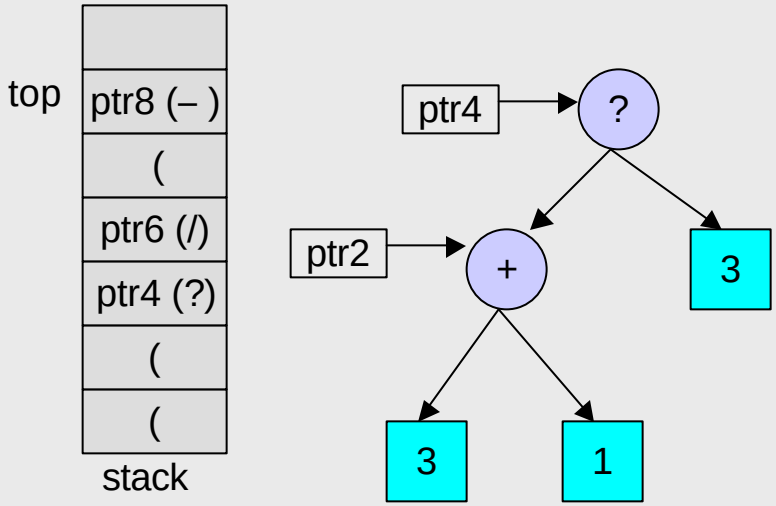


Stack and pointers just after encountering second close parenthesis

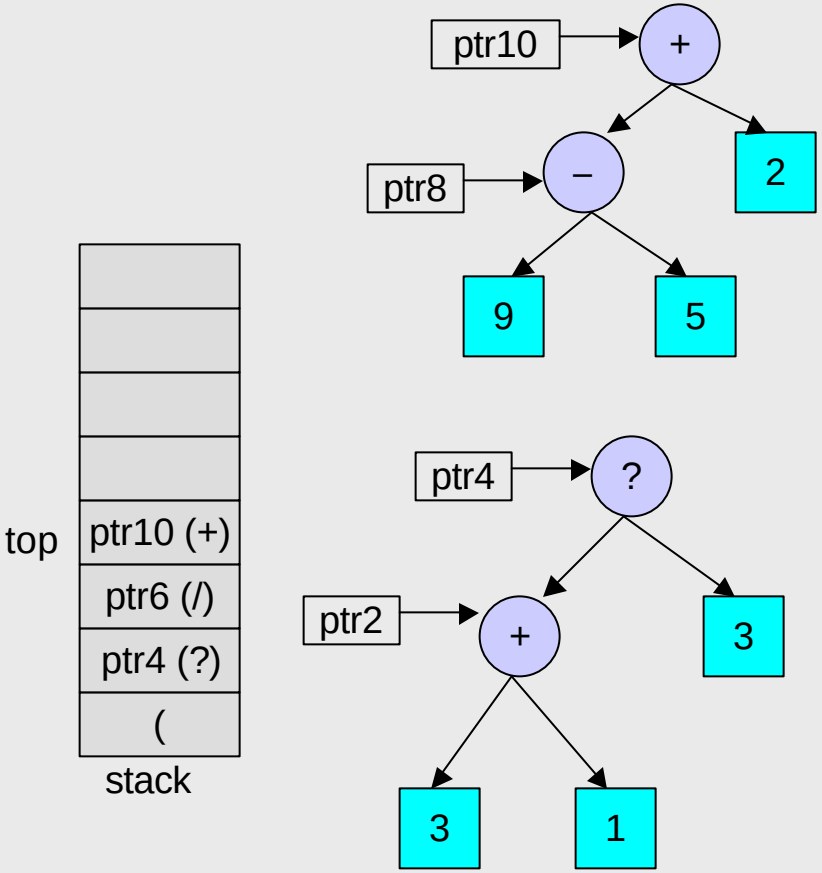
Here

Here

The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just after encountering third close parenthesis

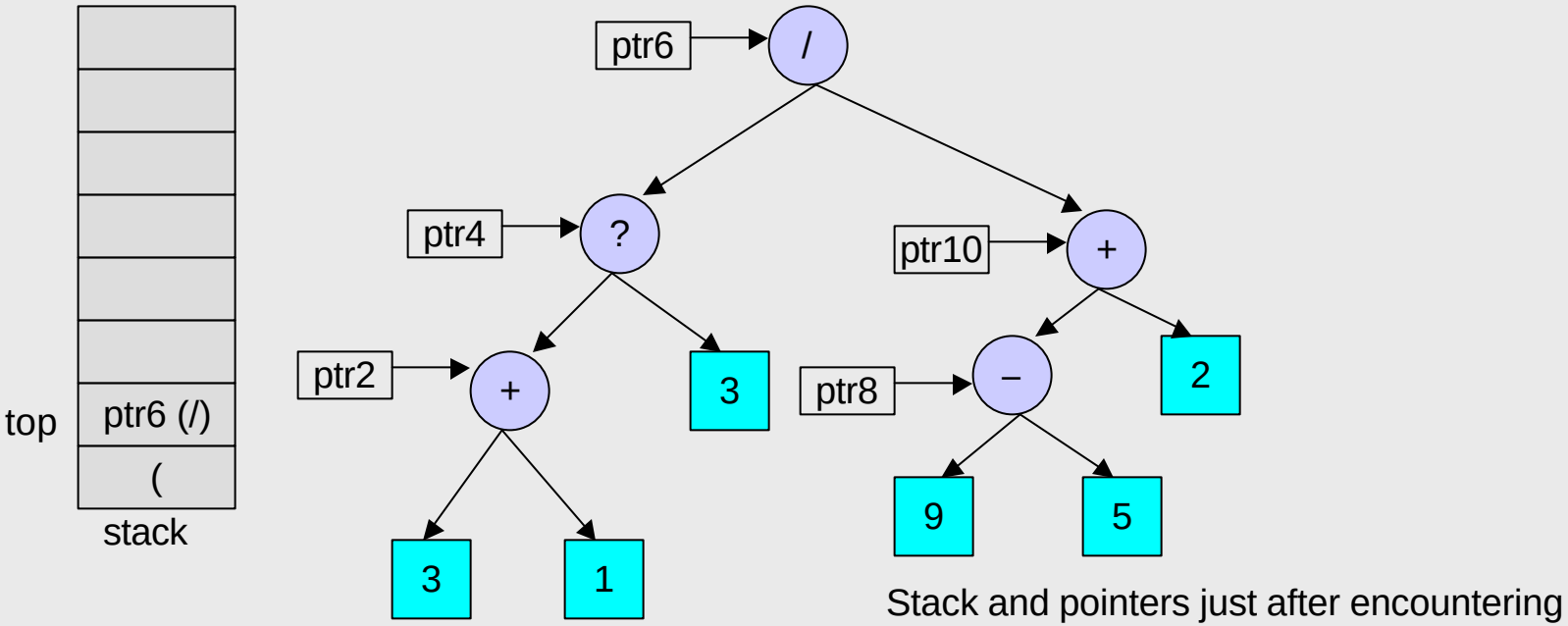


Stack and pointers just after encountering fourth close parenthesis

Here

Here

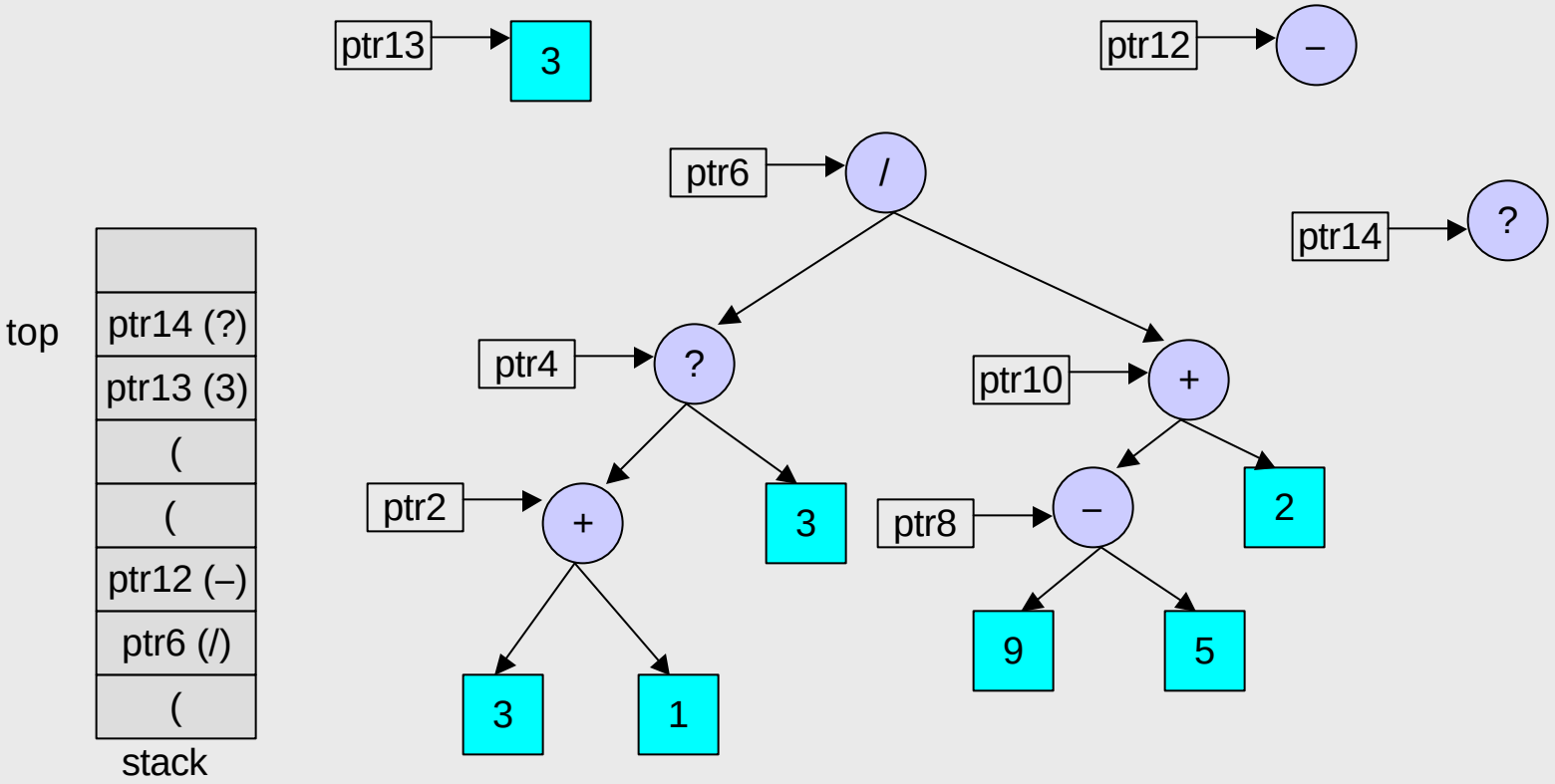
The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just after encountering fifth close parenthesis

Here

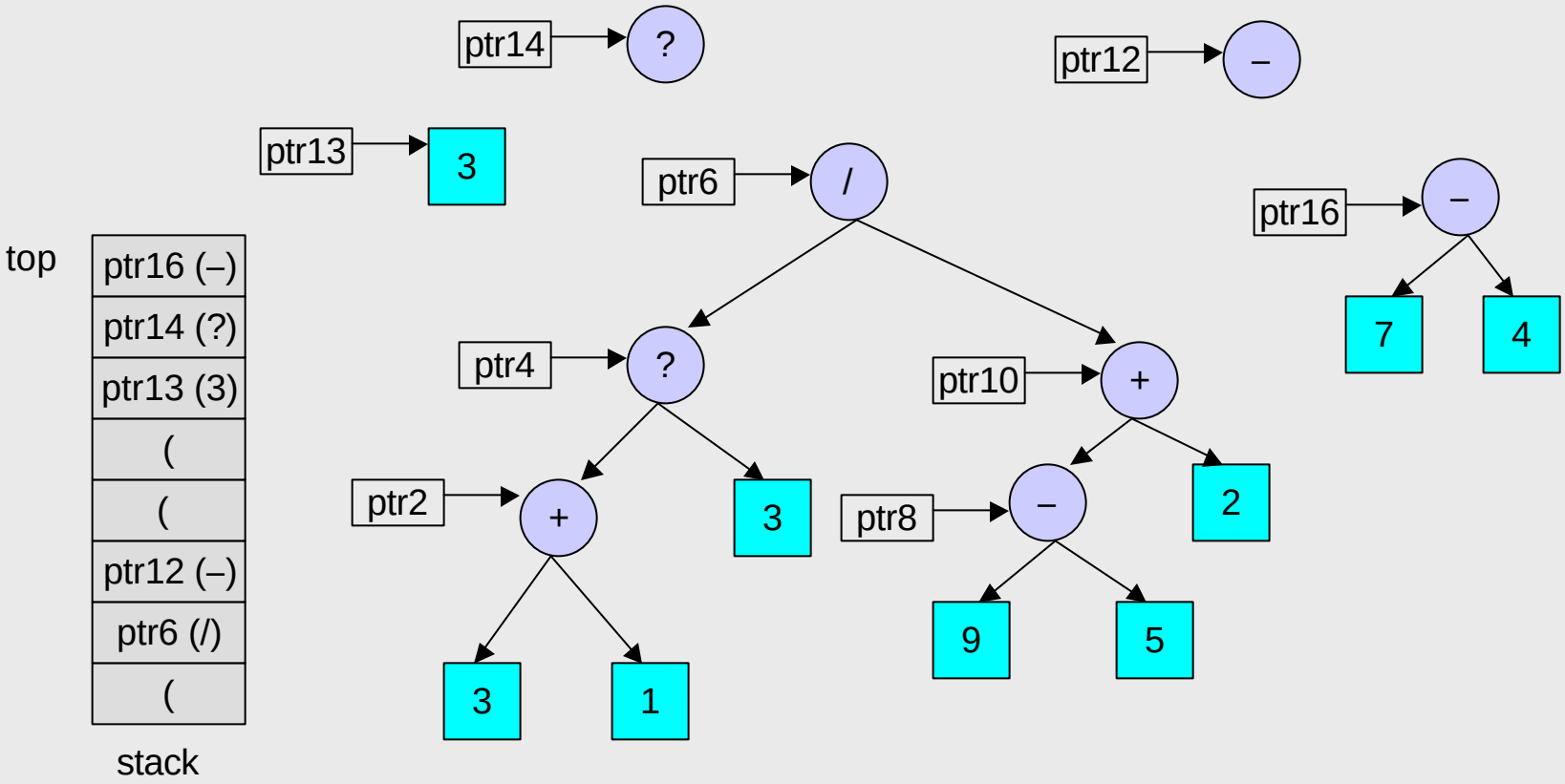
The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just before encountering ninth open parenthesis

Here

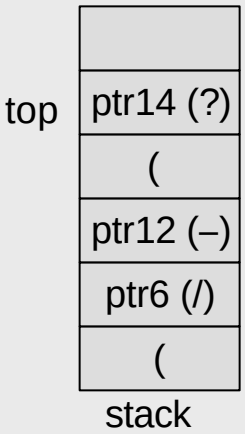
The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

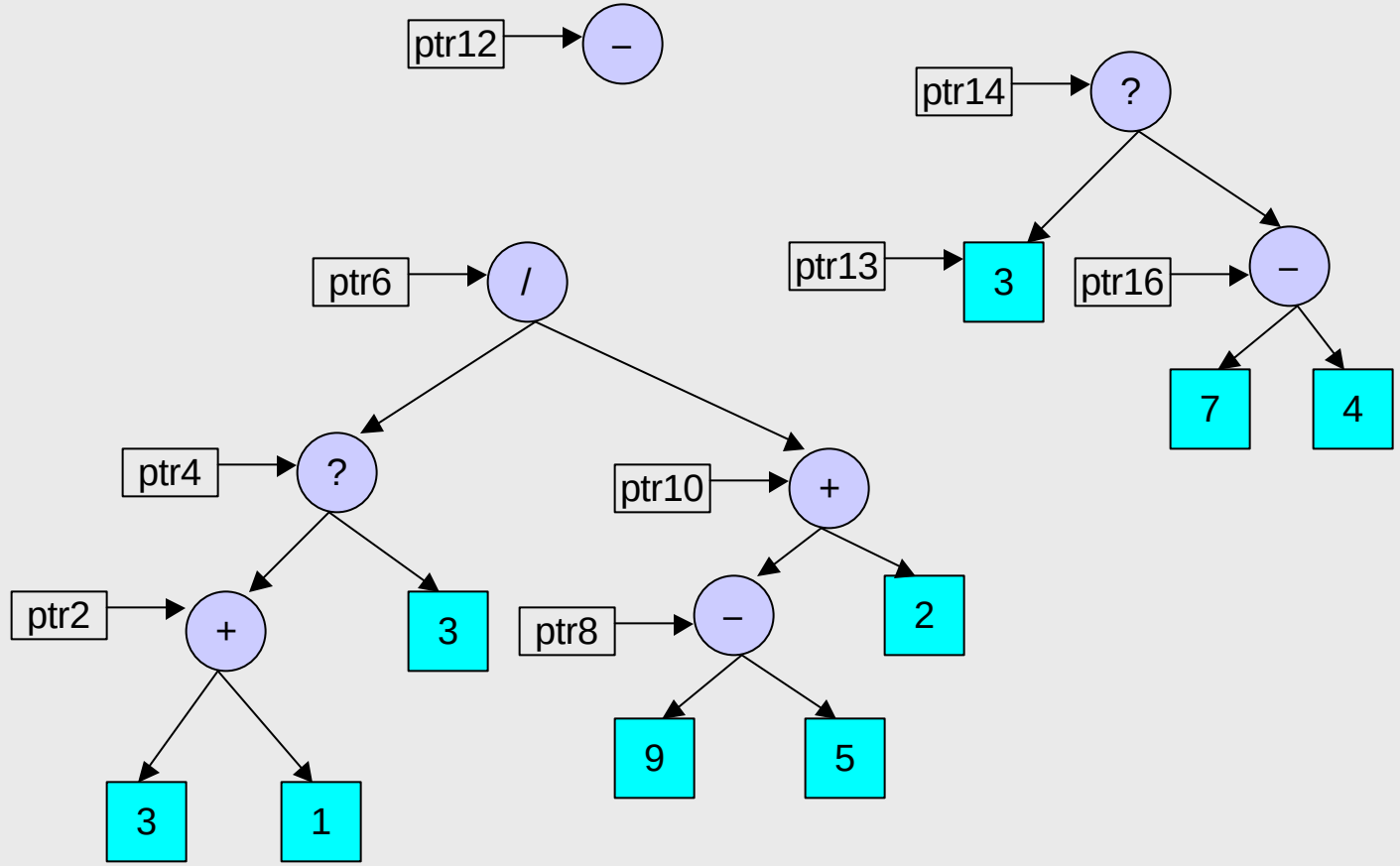


Stack and pointers just before encountering seventh close parenthesis

Here

The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

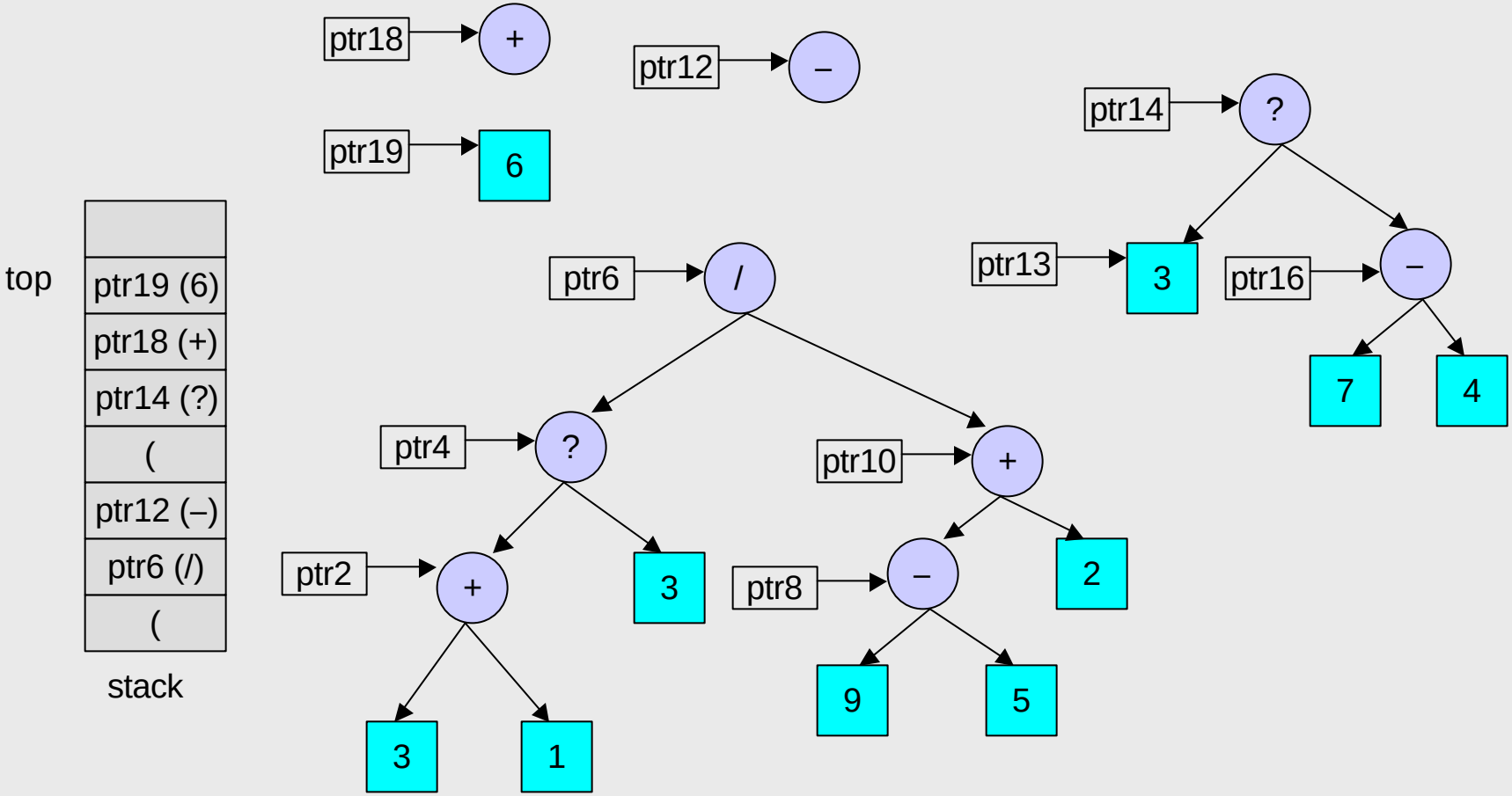




Stack and pointers just after encountering seventh close parenthesis

Here

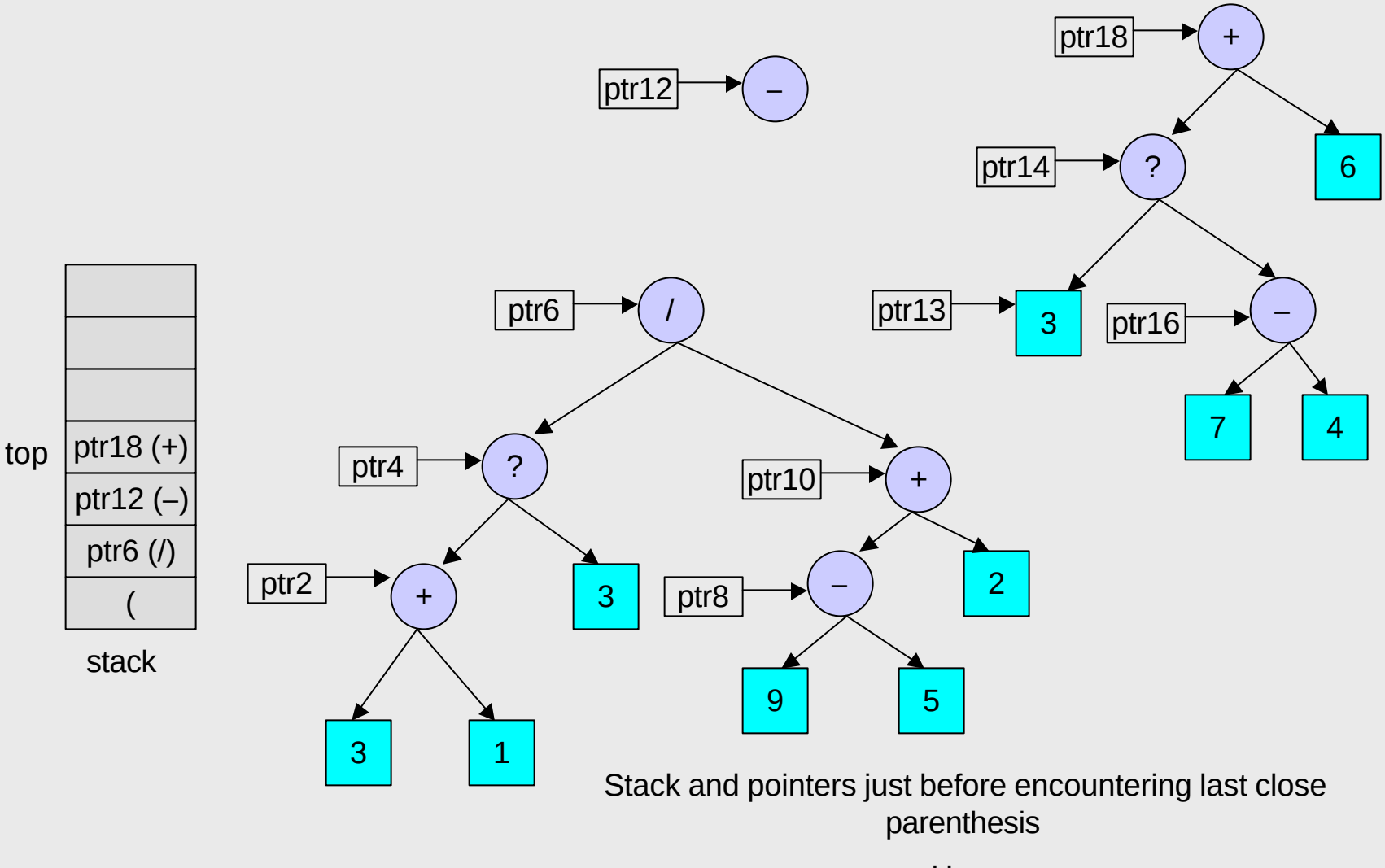
The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just before encountering next to last close parenthesis

Here →

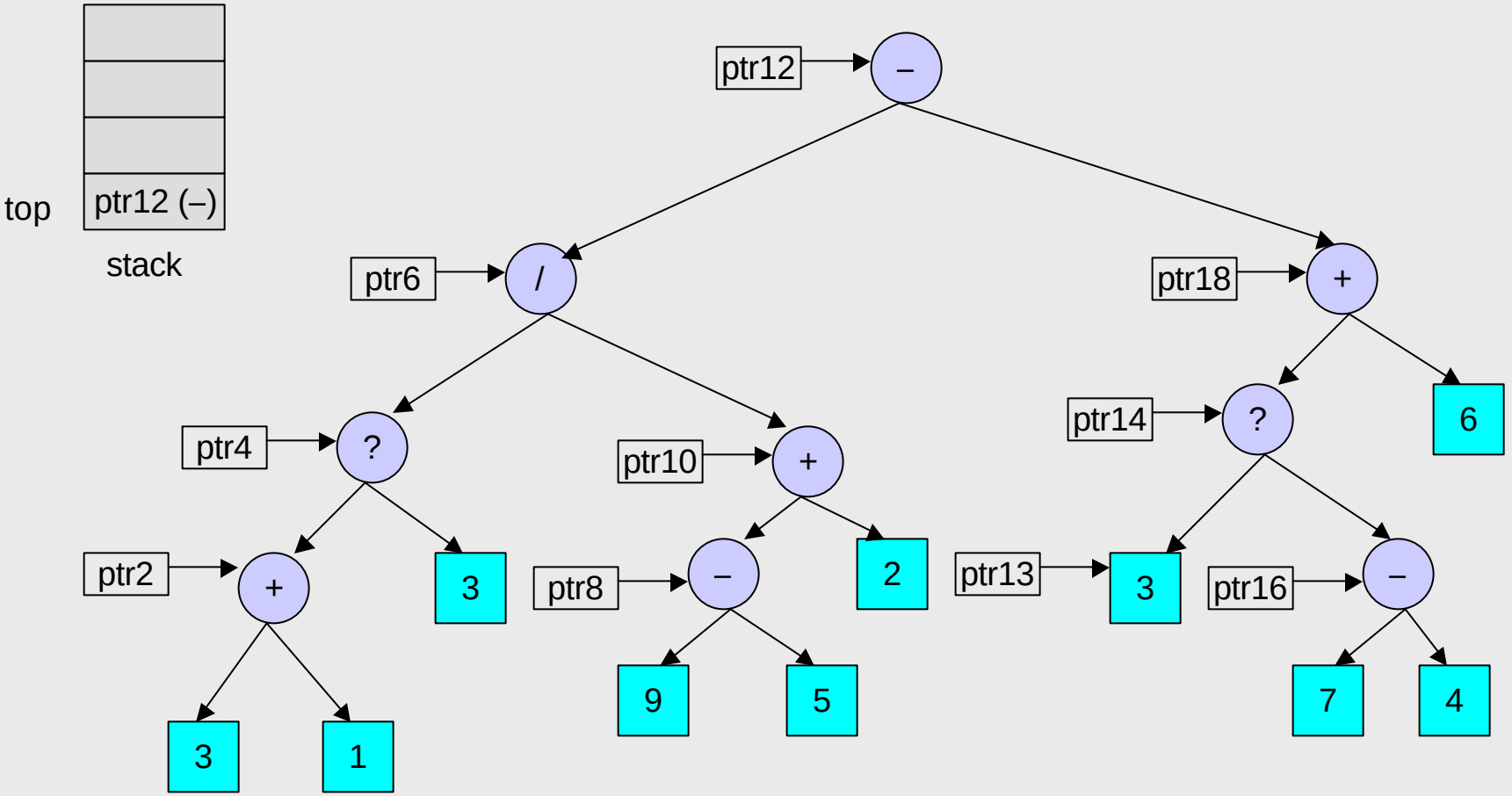
The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just before encountering last close parenthesis

Here →

The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))



Stack and pointers just after encountering last close parenthesis

Here

The infix expression: ((((3 + 1) ? 3) / ((9 – 5) + 2)) – ((3 ? (7 – 4)) + 6))

Binary Search Trees

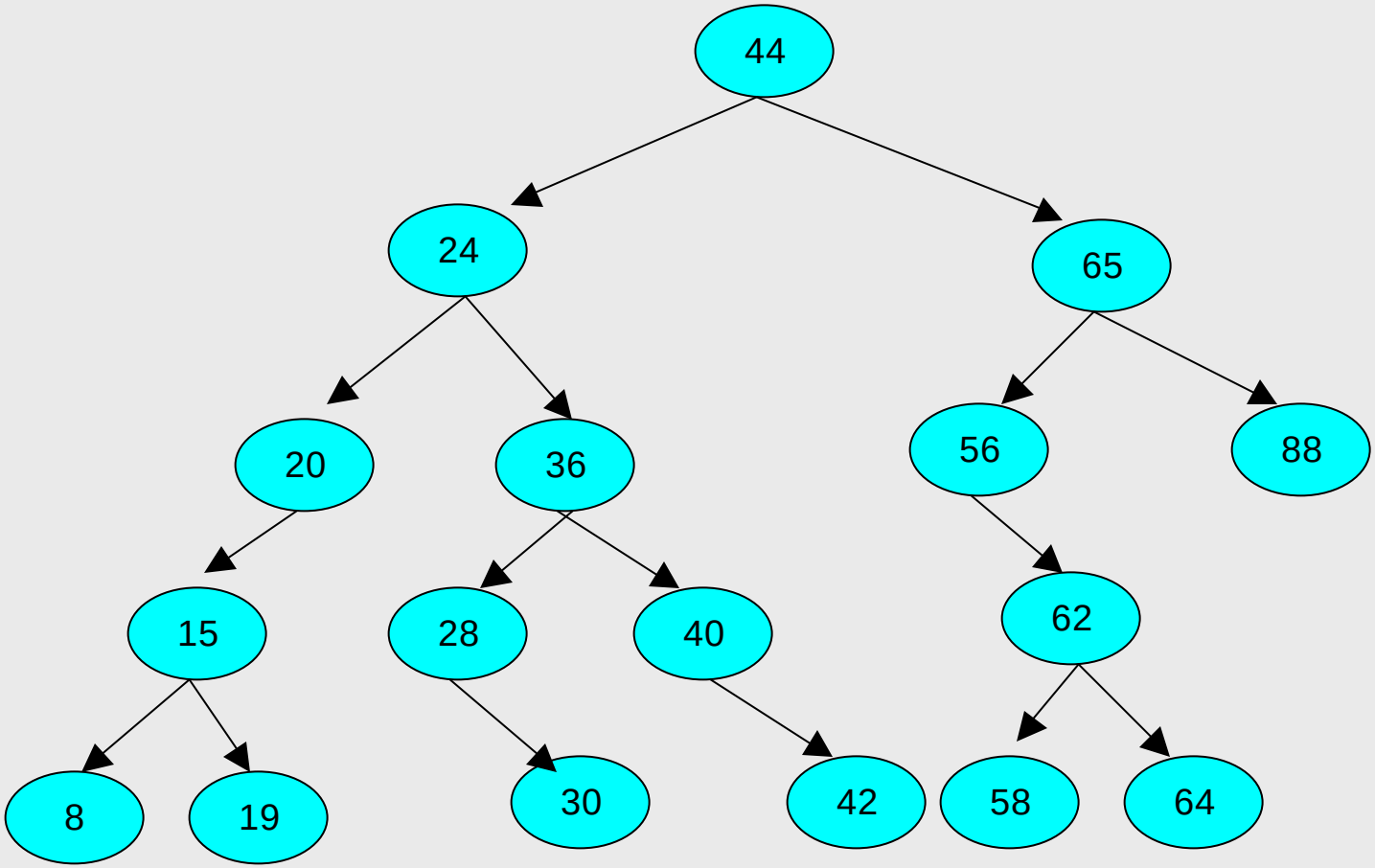
- A binary search tree is a binary tree that is either empty or each node of the tree contains a data value which satisfies the following:

1. All of the data values in the left subtree of each node are smaller than the data value in the node (root of the subtree) itself. (Stated another way, the value of the node itself is larger than the value of every node in its left subtree.)

2. All of the data values in the right subtree of each node are larger than the data value in the node (root of the subtree) itself. (Stated another way, the value of the node itself is smaller than the value of every node in its right subtree.)

3. Both the left and right subtrees of the node are themselves binary search trees.

Binary Search Trees (cont.)



A Binary Search Tree

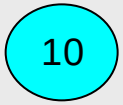
Binary Search Trees (cont.)

- A binary search tree, commonly referred to as a BST, is extremely useful for efficient searching. Basically, a BST amounts to embedding the binary search into the data structure itself.
- Notice how the root of every subtree in the BST on the previous page is the root of a BST.
- Clearly, the search tree ordering property means that insertions into a BST are not placed at some arbitrary point in the tree.
- Before we look at an algorithm to insert into a BST, let's see what steps need to occur to handle an insertion.

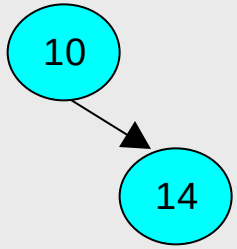
Insertion Into A Binary Search Tree

- Let's suppose we insert the data values, 10, 14, 6, 2, 5, 15, and 17 in their order of appearance, into an initially empty BST.

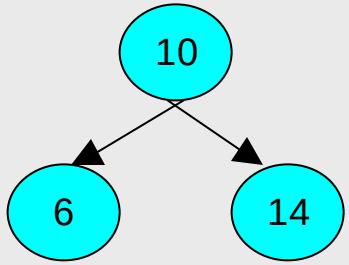
Step 1: Create new node with value 10.



Step 2: Create new node with value 14. This new node belongs in the right subtree of node 10 since 14 > 10. The right subtree of node 10 is empty so, node 14 becomes a right child of node 10.

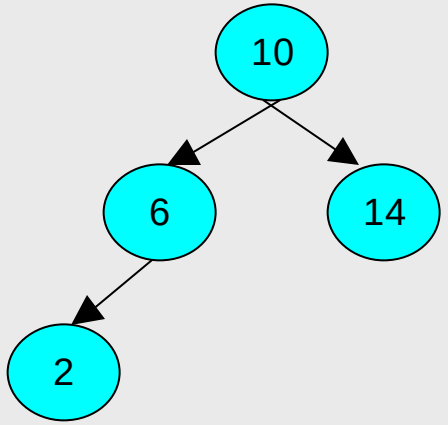


Step 3: Create new node with value 6. This new node belongs in the left subtree of node 10 since 6 < 10. The left subtree of node 10 is empty so, node 6 becomes the left child of node 10.

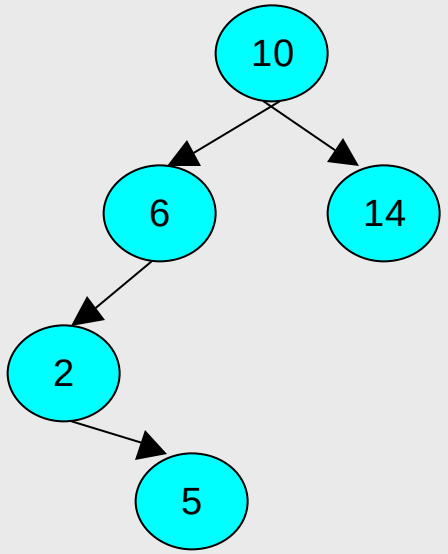


Insertion Into A Binary Search Tree (cont.)

Step 4: Create new node with value 2. This new node belongs in the left subtree of node 10 since 2 < 10. The root of the left subtree of node 10 has value 6. The new node belongs in the left subtree of node 6 since 2 < 6. The left subtree of node 6 is empty so, node 2 becomes the left child of node 6.

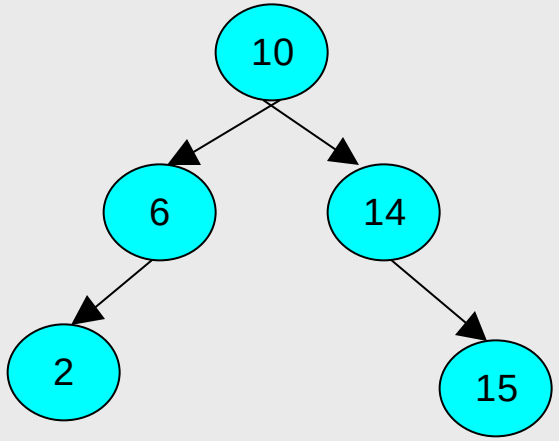


Step 5: Create new node with value 5. This new node belongs in the left subtree of node 10 since 10 > 5. The root of the left subtree of node 10 has value 6, so the new node belongs in the left subtree of node 6. The root of the left subtree of 6 has value 2. The new node belongs in the right subtree of 2 which is empty, so the new node with value 5 becomes the right child of node 2.

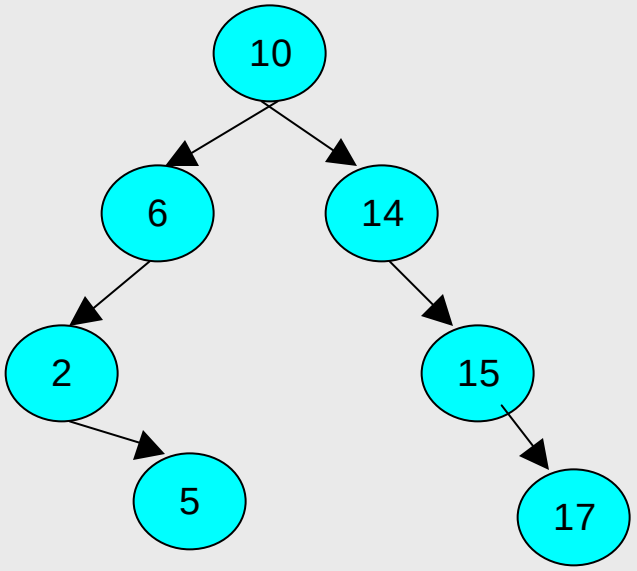


Insertion Into A Binary Search Tree (cont.)

Step 6: Create new node with value 15. This new node belongs in the right subtree of node 10 since 15 > 10. The root of the right subtree of node 10 has value 14. The new node belongs in the rightt subtree of node 14 since 14 < 15. The right subtree of node 14 is empty so, node 15 becomes the right child of node 14.



Step 7: Create new node with value 17. This new node belongs in the right subtree of node 10 since 10 < 17. The root of the right subtree of node 10 has value 14, so the new node belongs in the right subtree of node 14. The root of the right subtree of 14 has value 15. The new node belongs in the right subtree of 15 which is empty, so the new node with value 17 becomes the right child of node 15.



Binary Search Trees (cont.)

- This ordering of the nodes is an ordering property and not a structure property of the tree. This means that depending on how the BST is constructed it is possible for the tree to become skewed to either the right or left. In other words, either the right or left subtree of the root node is considerably deeper than the other side.