Solution Review: Calculating the Size of a Tree

This lesson contains the solution review for the challenge of calculating the size of a binary tree.

We'll cover the following

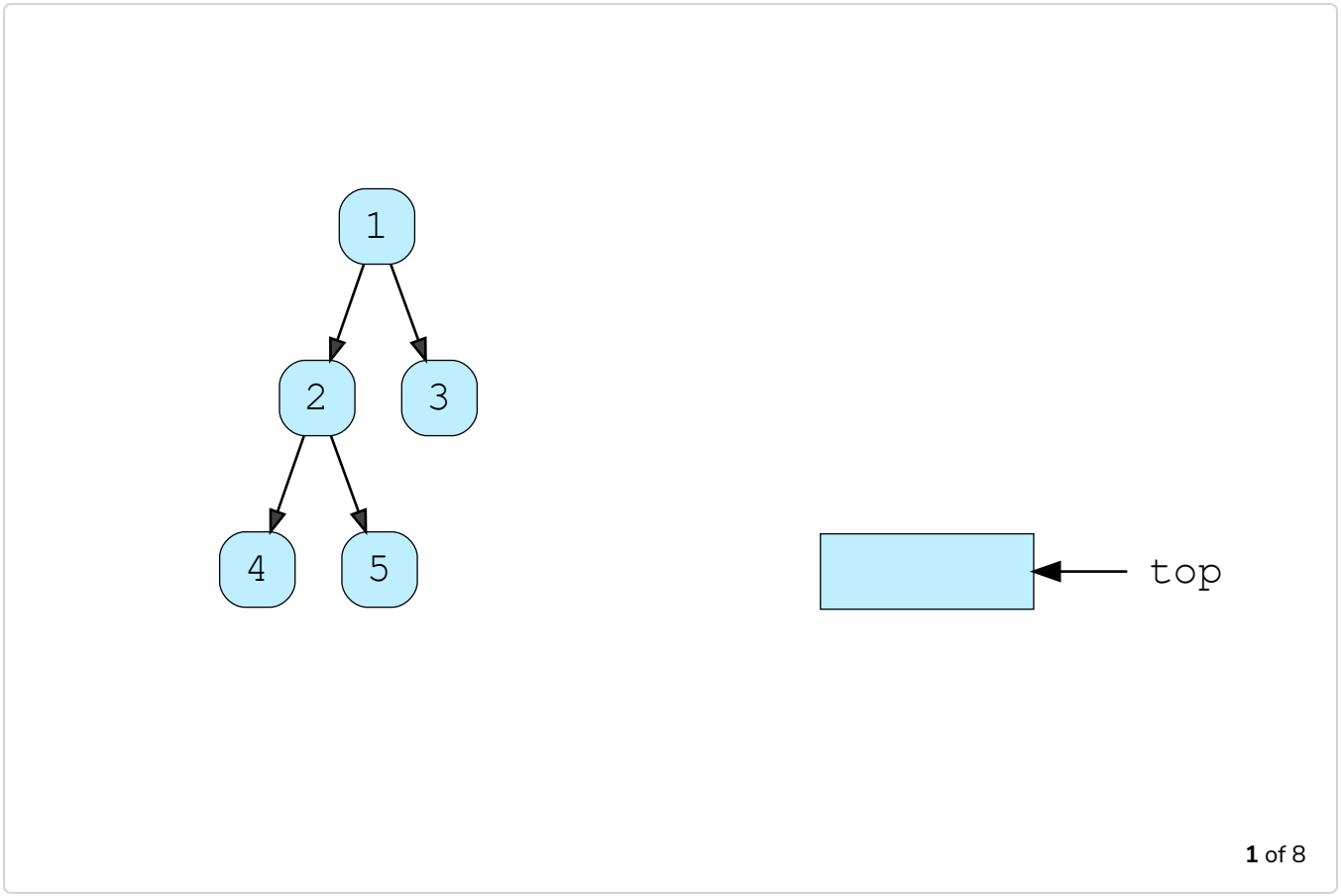
In this lesson, we will review a solution to the problem of determining the size of a binary tree.

The “size” of a binary tree is the total number of nodes present in the binary tree. We will explicitly define this quantity in greater detail and cover a strategy for how one may calculate this quantity in the binary tree data structure we have been building in this chapter.

We will discuss an iterative and a recursive approach for solving this challenge.

Iterative Approach

In the iterative solution, we’ll make use of a stack on which we can push the starting node and increment size by `1`. Next, we’ll pop elements and push their children on to the stack if they have any. For every push, we’ll increment size by `1`. When the stack becomes empty, the count for the size will also be final. Have a look at the slides below to check out the algorithm:



Here is the implementation of the algorithm illustrated above in Python:

```
def size(self):
  if self.root is None:
      return 0

  stack = Stack()
  stack.push(self.root)
  size = 1
  while stack:
    node = stack.pop()
    if node.left:
      size += 1
      stack.push(node.left)
    if node.right:
      size += 1
      stack.push(node.right)
  return size
```

Lines 2-3 contain the edge case which checks for an empty tree and returns `0` in that case. On **line 5**, we declare `stack` to a `Stack` object and push the root node on to the stack on **line 6**. After the push, we initialize `size` to `1` as we have a node present in the stack.

Next, we have a `while` loop which runs as long as `stack` is not empty. On **line 9**, we pop from the stack and store the popped element in `node`. On **lines 10-15**, we check if `node` has a left or right child and push the child on to the stack while also incrementing the `size` by `1`. Finally, when `stack` is empty, the `while` loop terminates, and `size` is returned on **line 16**.

Recursive Approach

We will recursively traverse the nodes and keep track of the count of the nodes visited. Check out the implementation below:

```
def size_(self, node):
  if node is None:
      return 0
  return 1 + self.size_(node.left) + self.size_(node.right)
```

Recursively speaking, the size of the tree is the size of the left subtree of the root node + the size of the right subtree of the root node + 1 (for the root node).

The base case is that an empty binary tree has a size of 0 so when `node` becomes `None`, we return `0` as a count. Otherwise, we return `1` plus the count from the recursive call on the left and the right subtree.

Now, this was pretty straightforward. In the code widget below, you can play around with the entire implementation of `BinaryTree` that we have covered so far in this chapter.

```
class Stack(object):
    def __init__(self):
        self.items = []

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
```

```
    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def __str__(self):
        s = ""
        for i in range(len(self.items)):
            s += str(self.items[i].value) + "-"
        return s


class Queue(object):
    def __init__(self):
        self.items = []

    def __len__(self):
        return self.size()

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value


class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def search(self, find_val, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_search(tree.root, find_val)
        elif traversal_type == "inorder":
            return self.inorder_search(tree.root, find_val)
        elif traversal_type == "postorder":
            return self.postorder_search(tree.root, find_val)
        else:
            print("Traversal type " + str(traversal_type) + " not recognized.")
            return False
```

```
    def print_tree(self, traversal_type):
        # Recursive traversals
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")

        # Iterative traversals
        elif traversal_type == "levelorder":
            return self.levelorder_print(tree.root)
        elif traversal_type == "inorder_iterative":
            return self.inorder_iterative(tree.root)
        elif traversal_type == "preorder_iterative":
            return self.preorder_iterative(tree.root)
        elif traversal_type == "postorder_iterative":
            return self.postorder_iterative(tree.root)
        else:
            print("Traversal type " + str(traversal_type) + " not recognized.")
            return False

    def levelorder_print(self, start):
        if start is None:
            return
        queue = Queue()
        queue.enqueue(start)

        traversal = ""
        while len(queue) > 0:
            traversal += str(queue.peek()) + "-"
            node = queue.dequeue()

            if node.left:
                queue.enqueue(node.left)
            if node.right:
                queue.enqueue(node.right)

        return traversal

    def preorder_search(self, start, find_val):
        if start:
            if start.value == find_val:
                return True
            else:
                return self.preorder_search(start.left, find_val) or \
                       self.preorder_search(start.right, find_val)
        return False

    def preorder_print(self, start, traversal):
        """Root->Left-Right"""
        if start:
            traversal += (str(start.value) + "-")
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)
        return traversal

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
```

```
            traversal = self.inorder_print(start.right, traversal)
        return traversal

    def postorder_print(self, start, traversal):
        """Left->Right->Root"""
        if start:
            traversal = self.postorder_print(start.left, traversal)
            traversal = self.postorder_print(start.right, traversal)
            traversal += (str(start.value) + "-")
        return traversal

    def preorder_iterative(self, start):
        stack = Stack()

        cur = start
        is_done = False

        traversal = ""
        while not is_done:
            if cur is not None:
                traversal += str(cur.value) + "-"
                stack.push(cur)
                cur = cur.left
            else:
                if len(stack) > 0:
                    cur = stack.pop()
                    cur = cur.right
                else:
                    is_done = True
        return traversal

    def inorder_iterative(self, start):
        s = Stack()

        cur = start
        is_done = False

        traversal = ""
        while not is_done:
            if cur is not None:
                s.push(cur)
                cur = cur.left
            else:
                if len(s) > 0:
                    cur = s.pop()
                    traversal += str(cur.value) + "-"
                    cur = cur.right
                else:
                    is_done = True
        return traversal

    def postorder_iterative(self, start):
        s = Stack()

        cur = start
        is_done = False

        traversal = ""
        while not is_done:

            if cur is not None:
                s.push(cur)
```

```
                cur = cur.left
            else:
                if len(s) > 0:
                    cur = s.pop()
                    traversal += str(cur.value) + "-"
                    cur = cur.right
                else:
                    is_done = True

        return traversal

    def height(self, node):
        if node is None:
            return -1
        left_height = self.height(node.left)
        right_height = self.height(node.right)

        return 1 + max(left_height, right_height)

    def size_(self, node):
        if node is None:
            return 0
        return 1 + self.size_(node.left) + self.size_(node.right)

    def size(self):
        if self.root is None:
            return 0

        stack = Stack()
        stack.push(self.root)
        size = 1
        while stack:
            node = stack.pop()
            if node.left:
                size += 1
                stack.push(node.left)
            if node.right:
                size += 1
                stack.push(node.right)
        return size

# Calculate size of binary tree:
#     1
#    / \
#   2  3
#  / \
# 4  5
#
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.size())
print(tree.size_(tree.root))
```

I hope you had fun learning about Binary Trees. In the next chapter, we have a different type of binary tree, i.e., the binary search tree. Stay tuned to find out more!