Carleton University
Department of Systems and Computer Engineering
SYSC 2100 - Algorithms and Data Structures - Winter 2023

Lab 8 - Binary Trees

Getting Started

1. Review *Important Considerations When Submitting Files to Brightspace*, which can be found in the course outline.

2. Launch Wing 101 and configure Wing's code reformatting feature. Instructions can be found in Appendix A of document, *SYSC 2100 - Style Guide for Python Code*.

   All code you submit for grading must be formatted. If you decide to disable automatic reformatting, make sure you manually reformat your code (Appendix A.2). At a minimum, we recommend that you reformat your file as you finish each exercise.

3. Download lab8_binarytree.py and lab8_test_binarytree.py from the *Lab Materials* module in Brightspace..

4. Open lab8_binarytree.py in Wing 101. Locate these assignment statements at the top of the file:

   ```
   __author__ = ''
   __student_number__ = ''
   ```

   Replace the empty strings with your name and student number. (Don't modify the variable names.)

5. Your solutions to the exercises must conform to the conventions described in *SYSC 2100 - Style Guide for Python Code.*

6. **Important**: if you decide to write a script in lab8_binarytree.py, it **must** be placed in an `if __name__ == '__main__':` block, like this:

   ```
   class BinaryTree:
     __class _Node:
           # _Node methods not shown
                ...

       # BinaryTree methods not shown

   if __name__ == '__main__':
       tree = BinaryTree()
           ...
   ```

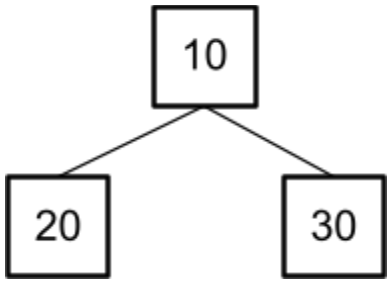
   Doing this ensures that the script will only be executed when you run lab8_binarytree.py as the "main" module, and won't be executed when

lab8_binarytree.py is imported into another module; for example, the testing/grading program provided to the TAs.

Class `BinaryTree` in lab8_binarytree.py is a partial implementation of a basic binary tree data structure. Nested class `_Node` implements the tree's nodes.

Read the `__init__` methods in `BinaryTree` and `_Node`. An instance of `BinaryTree` has one attribute, named `_root`, which stores a reference to the binary tree's root node. A new `BinaryTree` object is an empty tree; that is, it has no nodes.

Exercise 1: Suppose we want to build this binary tree, which has three nodes:

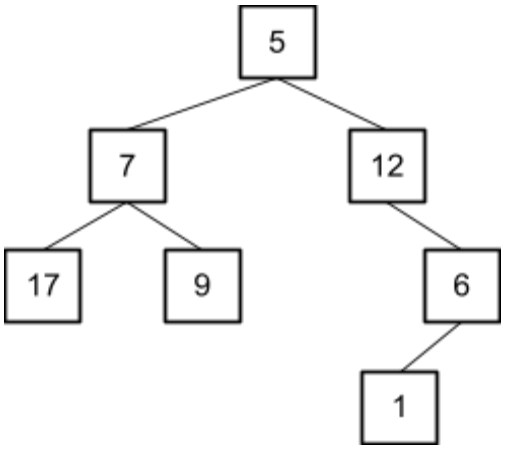


The `BinaryTree` and `_Node` classes don't provide methods to insert nodes in a binary tree. Building a basic binary tree requires that we directly access the attributes of the `BinaryTree` and `_Node` objects.

Function `build_10_20_30` in lab8_binarytree.py creates the tree shown above. We first create an empty binary tree, then install a new `_Node` containing 10 as the tree's root node. Next, we create a `_Node` containing 20 and install it as the root node's left child. Finally, we create a `_Node` containing 30 and install it as the root node's right child..

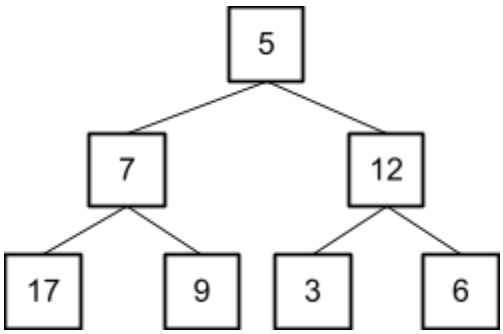
In lab8_test_binarytree.py, class `Build_10_20_30_TestCase` tests this function. Read `build_10_20_30` and its test class. Run the test program and review the output in the shell window.

Exercise 2: `build_binary_tree` (towards the end of lab8_binarytree.py) is an incomplete implementation of a function that creates the binary tree shown in the following diagram. Read the docstring, delete the `raise` statement and complete the function definition.



In lab8_test_binarytree.py, define one or more methods in class `Build_Binary_Tree_TestCase` to thoroughly test `build_binary_tree`. Run the test script and review the output in the shell window. If necessary, edit `build_binary_tree` and rerun the test script until all tests pass.

Exercise 3: `build_perfect_binary_tree` (towards the end of lab8_binarytree.py) is an incomplete implementation of a function that creates the perfect binary tree shown in the following diagram. Read the docstring, delete the `raise` statement and complete the function definition.



In lab8_test_binarytree.py, define one or more methods in class `Build_Perfect_Binary_Tree_TestCase` to thoroughly test `build_perfect_binary_tree`. Run the test script and review the output in the shell window. If necessary, edit `build_perfect_binary_tree` and rerun the test script until all tests pass.

Exercise 4: Suppose we want to print the payload stored in each node in a binary tree. One way to do this is to perform a *pre-order* traversal of the binary tree. As each node is visited, the node's payload is printed before the node's children are visited.

Method `preorder_print` in class `BinaryTree` is a *wrapper* method that calls recursive

method `_preorder_print`, which performs the pre-order traversal. Trace `preorder_print`, step-by-step, and predict what will be printed when the method is called on the trees created by `build_binary_tree` and `build_perfect_binary_tree`; e.g.,

```
>>> tree = build_binary_tree()
>>> tree.preorder_print()
>>> tree = build_perfect_binary_tree()
>>> tree.preorder_print()
```

Execute this code in the shell. Were your predictions correct?

Change `_preorder_print` from a recursive *method* to a recursive *function* that is nested in method `preorder_print`. Edit `preorder_print` to call the function. (See methods `size` and `height` and functions `_size` and `_height` for examples of how to do this.)

Use the shell to pre-order print the trees built by `build_binary_tree` and `build_perfect_binary_tree`.Verify that the output produced by the revised `preorder_print` method is correct.

Exercise 5: Another way to print a binary tree is to perform an *in-order* traversal of the tree. Each node *u is* printed after all the nodes in *u*'s left subtree have been printed but before any of the nodes in *u*'s right subtree have been printed.

Read the docstring for the wrapper method `inorder_print`. Delete the `raise` statement and complete the method definition. In `inorder_prin`t, define a recursive nested function named `_inorder_print`. The function header and docstring are:

```
def _inorder_print(node: 'BinaryTree._Node') -> None:
    """Print the binary tree rooted at node using an inorder
    traversal."""
```

Remember, `inorder_print` must call `_inorder_print`, and `_inorder_print` cannot contain any loops.

Predict what would be printed by an in-order traversal of the trees created by `build_binary_tree` and `build_perfect_binary_tree`. Now use the shell to call `inorder_print` on these trees. Were your predictions correct?

Exercise 6: A third way to print a binary tree is to perform a *post-order* traversal of the tree. Each node *u is* printed after all the nodes in *u*'s left and right subtrees have been printed.

Read the docstring for the wrapper method `postorder_print`. Delete the `raise` statement and complete the method definition. In `postorder_print`, define a recursive function named `_postorder_print`. The function header and docstring are:

```
def _postorder_print(node: 'BinaryTree._Node') -> None:
    """Print the binary tree rooted at node using a postorder
```

```
    traversal."""
```

Remember, `postorder_print` must call `_postorder_print`, and `_postorder_print` cannot contain any loops.

Predict what would be printed by a post-order traversal of the trees created by `build_binary_tree` and `build_perfect_binary_tree`. Now use the shell to call `postorder_print` on these trees. Were your predictions correct?

Exercise 7: Read the docstring for method `count`, delete the `raise` statement and complete the method definition. Use the same approach as `size`, `height` and the printing methods: `count` should be a wrapper method that calls a nested recursive function (not a method) that visits all the nodes in the tree.

In lab8_test_binarytree.py, define one or more methods in class `CountTestCase` to thoroughly test `count`. Run the test script and review the output in the shell window. If necessary, edit `count` and rerun the test script until all tests pass.

Wrap Up

- Before submitting lab8_binarytree.py, review your code. Does it conform to the coding conventions, as specified in the *Getting Started* section? Has it been formatted? Did you edit the `__author__` and `__student_number__` variables?
- Submit lab8_binarytree.py to Brightspace. Make sure you submit the file that contains your solutions, not the unmodified file you downloaded from Brightspace! You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the submission due date. Only the most recent submission will be saved by Brightspace.
- Don't submit lab8_test_binarytree.py.
- Solutions that are emailed to your instructor or a TA will not be graded, even if they are emailed before the submission due date.