UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 8

Primitive Binary Trees

Date Due: July 26, 2019, 11pm **Total Marks: 72**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

Version History

- **07/19/2019**: released to students

Primitive Binary Trees

In this assignment, the objective is to master the treenode ADT (Readings Chapter 18), which is very similar to the node ADT (Readings Chapter 11). In Chapter 11, we used the term `node-chain` to refer to sequences of nodes, we will use the phrase `primitive binary trees` to refer to the kinds of structures that we can build with treenodes, as described in Chapters 18 and 19. The adjective `primitive` refers to the limitations of the operations. As we will see in Chapters 22-24, we can make use of primitive binary trees as the basis for more capable data structures, just as we did for node-chains in Chapters 14 and 15.

The first 2 questions on this assignment consist of a collection of relatively short exercises, some of which are somewhat artificial or contrived. The lesson here is to get as much practice with treenodes as possible. Questions 3 and 4 are a little more interesting, and ask you to think a bit deeper about efficiency and trees.

There are a bunch of files available to you for your use in these exercises.

- The treenode ADT is found in `treenode.py`
- Some functions to create binary trees are found in `treebuilding.py`
- Since primitive binary trees are essentially nested dictionaries, it can be difficult to test functions that return trees. As in Assignment 5, we've provided functions that can help you visualize trees called `to_string_for_printing(tnode)`, which takes a single primitive tree and returns a string that you can print to see the structure of the tree more easily. This function is found in `treetesting.py`, which also contains some other functions that you may find useful for some of your testing work, in the same way that Assignment 5's `to_string()` could have been used.

Question 0 (5 points):

Purpose: To force the use of Version Control in Assignment 8

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 8. Do the following steps.

1. Create a new PyCharm project for Assignment 8.
2. Use `Enable Version Control Integration...` to initialize Git for your project.
3. Download the Python and text files provided for you with the Assignment, and add them to your project.
4. Before you do any coding or start any other questions, make an initial commit.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open the terminal in your Assignment 8 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm, locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.

You may need to work in the lab for this; Git is installed there.

What to Hand In

After completing and submitting your work for Questions 1-4, open a command-line window in your Assignment 8 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/-paste this into a text file named `a8-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the way to use git, but it is the way students work on assignments.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: The log file shows that you used Git as part of your work for Assignment 8. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per programming question for a total of at least 8 commits.

Question 1 (25 points):

Purpose: To practice recursion on binary trees.

Degree of Difficulty: Easy.

You can find the `treenode` ADT on the assignment page. Using this ADT, implement the following functions:

1. `count_node_types(tnode)` Purpose: Returns a tuple containing the number of leaf nodes in the tree, and the number of non-leaf nodes in the tree. A leaf node is a node without any children. The `is_leaf()` function provided in `treefunctions.py` can be used to check if a node is a leaf node. Remember, you can use circle brackets to create a tuple:

```
a_tuple = ("a", "b")
print( a_tuple[0] )
# Prints out "a"
```

2. `subst(tnode, t, r)` Purpose: To substitute a target value t with a replacement value r wherever it appears as a data value in the given tree. Returns `None`, but modifies the given tree.
3. `copy(tnode)` Purpose: To create an exact copy of the given tree, with completely new treenodes, but exactly the same data values, in exactly the same places. If `tnode` is `None`, return `None`. If `tnode` is not `None`, return a reference to the new tree.
4. `nodes_at_level(tnode, level)` Purpose: Counts the number of nodes in the given primitive tree at the given level, and returns the count. If `level` is too big or too small, a zero count is returned.

What to Hand In

- A file `a8q1.py` containing your 4 functions.
- A file `a8q1_testing.py` containing your testing for the 4 functions.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- Each function will be graded as follows:
  - 2 marks: Your function has a good doc-string.
  - 3 marks: Your function is recursive and correct.
- 5 marks: You've tested your functions.

Question 2 (20 points):

Purpose: To do some deeper reflection on primitive binary trees.

Degree of Difficulty: Moderate

We say that two binary trees t_1 and t_2 satisfy *the mirror property* if all of the following conditions are true:

1. The data value stored in the root of t_1 is equal to the data value stored in the root of t_2.
2. The left subtree of t_1 and the right subtree of t_2 satisfy the mirror property.
3. The right subtree of t_1 and the left subtree of t_2 satisfy the mirror property.

If both t_1 and t_2 are empty, we say the mirror property is satisfied, but if one is empty but the other is not, the mirror property is not satisfied.

Using the TreeNode ADT found on the assignment page:

- Write a function `mirrored(t1, t2)` that returns `True` if the two given trees satisfy the mirror property, and `False` otherwise.
- Design and implement a recursive function called `reflect(tnode)` that swaps every left and right subtree in the given primitive tree. This function returns None, but modifies the given tree.
- Design and implement a recursive function called `reflection(tnode)` that creates a copy of the given tree, with left and right subtrees swapped. In other words, the returned tree has to satisfy the mirrored property with the original tree, but the original tree cannot be affected at all.
- Explain why `mirrored(atree, reflect(atree))` returns `False`, whereas `mirrored(atree, reflection(atree))` returns `True`.

What to Hand In

- A file called `a8q2.py`, containing your three function definitions.
- A file `a8q2_testing.py` containing your testing for the three functions.
- A file called `a8q2.txt`, containing your explanation.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of the file.

Evaluation

- For each function:
  - 5 marks: Your function works.
- 2 marks: Your explanation is correct.
- 3 marks: Your testing is adequate.

Question 3 (12 points):

Purpose: To do more thinking about binary trees; to practice the art of tupling.

Degree of Difficulty: Moderate

In class we defined a complete binary tree as follows:

> A *complete* binary tree is a binary tree that has exactly two children for every node, except for leaf nodes which have no children, and all leaf nodes appear at the same depth.

Visually, complete binary trees are easy to detect. But a computer can't read diagrams as well as humans do, so a program needs to be written that explores the tree by walking through it.

Consider the function below.

```
1  import treenode as tn
2
3  def bad_complete(tnode):
4      """
5      Purpose:
6          Determine if the given tree is complete.
7      Pre-conditions:
8          :param tnode: a primitive binary tree
9      Post-conditions:
10         The tree is unaffected.
11     Return
12         :return: the height of the tree if it is complete
13                  -1 if the tree is not complete
14     """
15     if tnode is None:
16         return 0
17     else:
18         ldepth = bad_complete(tn.get_left(tnode))
19         rdepth = bad_complete(tn.get_right(tnode))
20         if ldepth == rdepth:
21             return rdepth+1
22         else:
23             return -1
```

The function `bad_complete()` is designed to return an integer. If the integer is positive, then `bad_complete()` is indicating that the tree is complete because the left subtree is complete, and the right subtree is complete, and furthermore, the two sub trees have the same height. However, if either subtree is not complete, or if the two subtrees have different height, the whole tree cannot be complete. If a tree is not complete, `bad_complete()` returns the value `-1`.

Using integers this way to provide two different kinds of messages is very common, but can lead to problems with correctness and robustness. That's one reason why the function has been named `bad_complete()`.

The other reason that the function above is named `bad_complete()` is that it is massively inefficient. To understand the problem with `bad_complete()`, we need a case analysis.

- If the given tree is complete, the function has to explore the whole tree. This is the worst case, and if the tree is complete, there is no way to avoid exploring the whole tree. This is expensive, but necessary.
- Suppose that we have a tree whose left subtree is not complete, but whose right subtree is complete. In this case, we have to explore the left subtree to find out that it is not complete, but once we know that, the fact that the right sub-tree is complete makes no difference. A tree whose left subtree is not complete cannot be complete, no matter what the right subtree is. Exploring the right subtree when the left subtree is not complete is a complete waste of effort.

- Suppose that we have a tree whose left subtree is complete, and has height 4, and the right subtree is complete with height 1000. Exploring the whole right subtree to its full depth is not necessary; we can conclude that the whole tree is not complete as soon as we discover that the right subtree's height is different from the height of the left subtree.

Task

Write a function named `complete(tnode)` using the same approach as `bad_complete()`, but instead of returning a single integer, `complete(tnode)` should return a tuple of 2 values, `(flag, height)`, where:

- `flag` is True if the subtree is complete, False otherwise
- `height` is the height of the subtree, if `flag` is True, `None` otherwise.

This technique is called "tupling."

Your function should avoid unnecessary work when a tree is not complete. To be clear, you cannot avoid exploring the whole tree when the tree is complete. But we could save some time when we discover an incomplete tree.

Here's a description of the function interface:

```
def complete(tnode):
    """
    Purpose:
        Determine if the given tree is complete.
    Pre-conditions:
        :param tnode: a primitive binary tree
    Return
        :return: A tuple (True, height) if the tree is complete,
                 A tuple (False, None) otherwise.
    """
```

Hints:

- Your recursive calls will return a tuple with 2 values. Python allows tuple assignment, i.e., an assignment statement where tuples of the same length appear on both sides of the =. For example:

```
# tuple assignment
a,b = 3,5

# tuple assignment
a,b = b,a
```

- Use your algorithm analysis skills in your design phase. Also, use the skills that you gained in Labs 6 and 7 for timing your programs to show that your function is relatively fast on incomplete trees. To help you do this, take note of the following functions:
  - The function `build_complete()` in the file `treebuilding.py` builds a complete tree of a given height.
  - The function `build_fibtree()` in the file `treebuilding.py` builds a large tree that is not complete.
  - You can build a tricky tree as follows:

```
import treenode as TN
import treebuilding as TB

tricky_tree = TN.create(0, TB.build_fibtree(5), TB.build_complete(10))
```

What to Hand In

- A file called `a8q3.py`, containing the function definition for the function `complete(tnode)`.
- A file `a8q3_testing.py` containing your testing.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: your function correctly uses tupling.
- 5 marks your function does not do more work than it has to.
- 2 marks: your testing is adequate.

Question 4 (10 points):

Purpose: To solve an algorithm problem for trees that is slightly more interesting.

Degree of Difficulty: Tricky

We're interested in finding a path from the root of a tree to a node containing a given value. If the given value is in the tree, we want to know the data values on the path from the root to the value, as a Python list. The list should be ordered with the given value first, and the root last (like a stack of the data of nodes visited if you walked the path starting from the root).

- Design and implement a recursive function called `path_to(tnode, value)` that returns the tuple `(True, alist)` if the given value appears in the tree, where `alist` is a Python list with the data values found on the path, ordered as described above. If the value does not appear in the tree at all, return the tuple `(False, None)`.
- In a tree, there is at most one path between any two nodes. Design a function `find_path(tnode, val1, val2)` that returns the path between any the node containing `val1` and the node containing `val2`. This function is not recursive, but should call `path_to(tnode, value)`, and work with the resulting lists. If either of the two given values do not appear in the tree, return `None`.

  Hint: Assume that the two values are in the tree. There are three ways the two values could appear. One value could be in the left subtree, and the other could be in the right subtree. In this case, the path passes through the root of the tree. The lists returned by `path_to(tnode, value)` would have exactly one element in common, namely the root. You can combine these two lists, but you only need the root to appear once.

  On the other hand, both `val1` and `val2` could be on the left subtree, or both on the right subtree. In both of these two cases, `path_to(tnode, value)` will have some values in common, which are not on the path between `val1` and `val2`. The path between them can be constructed by using the results from `path_to(tnode, value)`, and removing some but not all of the elements the two lists have in common.

Assumptions

- Assume that tree values are not repeated anywhere in the tree. As a result, you cannot really use `treebuilding.build_fibtree()` for testing, but `treebuilding.build_complete()` would work.

What to Hand In

- The file `a8q4.py` containing the two function definitions:
  - `path_to(tnode, value)`
  - `find_path(tnode, val1, val2)`
- The file `a8q4_testing.py` containing testing for your function.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 4 marks: `path_to(tnode, value)` is correct.
- 4 marks: `find_path(tnode, val1, val2)` is correct.
- 2 marks: your testing is adequate.