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: Tuesday, August 2, 11:59pm **Total Marks: 82**

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.
- This assignment is homework assigned to students and will be graded. This assignment shall not be distributed, in whole or in part, to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this assignment to a student registered in the course. **Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.**
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to com-pare every pair of student submissions. Plagiarism can include: copying answers from a web page, or from a classmate, or from solutions published in previous semesters. Basically, if you cannot delete your whole assignment and do it again yourself (given adequate time), it's not your work, so don't try to claim credit for it. Your success in this course depends on what you can do, not on what you can hand in.
- 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.
- Read the purpose of each question. Read the Evaluation section of each question.
- 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 Canvas.** If you are not sure, talk to a Lab TA about how to do this.
- **Canvas 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. **Do not send late assignment submissions to your instructors, lab TAs, or markers. If you require an extension, request it in advance using email.**

Version History

- **07/29/2021**: Question 3.1 Corrected the description for `check_trees()`.
- **07/29/2021**: released to students

Primitive Binary Trees

In this assignment, the objective is to master the treenode ADT (Readings Chapter 20), which is very similar to the node ADT (Readings Chapter 13). In Chapter 13, 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 20 and 21. 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 16 and 17.

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`. You can use these functions for your formal or informal testing.
- Since primitive binary trees are simple classes, it can be difficult to test functions that return trees. As in Assignment 5, we've provided a 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 1 (20 points):

Purpose: Students will practice the following skills:

- Designing, implementing and testing functions that do calculations on trees.

Degree of Difficulty: Easy if you did A7. This is why we did all that recursion!

References: You may wish to review the following:

- Chapter 20, 21

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

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

1. `nodes_at_level(tnode, level)` Purpose: Counts the number of nodes in the given primitive tree are at the given level, and returns the count. If `level` is too big or too small, a zero count is returned. Use an assertion to ensure that the value `level` is non-negative; if the given value is negative, your function should fail the assertion, and cause a run-time error. This behaviour does not need to be formally tested in your test script.
2. `largest_leaf_value(tnode)` Purpose: searches the given primitive tree and returns the largest data value stored at any leaf node. If the given `treenode` is empty, this function should return `None`. This function should only be applied to trees containing numeric data values. Don't try to prevent or check if the function is used on other kinds of trees.
3. `closest(tnode, target)` Purpose: searches the given primitive binary tree, and returns the data value that is closest to the given `target`. In this question, we'll define closeness by the value of `abs(target - x)`, where `x` is a data value stored in a treenode. If the primitive binary tree is empty, the function should return `None`. This function should only be applied to trees containing numeric data values. Don't try to prevent or check if the function is used on other kinds of trees.

What to Hand In

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

Be sure to include your name, NSID, student number, and course number 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: Students will practice the following skills:

- Working directly with the three recursive tree traversal techniques.

Degree of Difficulty: Easy. This really is just to reinforce the three traversal orderings.

References: You may wish to review the following:

- Chapter 20, 21

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

In this question, you will design three similar functions whose purpose is to gather the data values in a given tree into a list. The difference between the three functions is the order that the values appear in the list.

- `collect_inorder(tnode)`. Purpose: Returns a list of data values from the given primitive binary tree consistent with an in-order traversal of the tree.
- `collect_preorder(tnode)`. Purpose: Returns a list of data values from the given primitive binary tree consistent with a pre-order traversal of the tree.
- `collect_postorder(tnode)`. Purpose: Returns a list of data values from the given primitive binary tree consistent with a post-order traversal of the tree.

What to Hand In

- A file `a8q2.py` containing your functions.
- A file `a8q2_testing.py` containing your testing for the functions.

Be sure to include your name, NSID, student number, and course number 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 3 (20 points):

Purpose: Students will practice the following skills:

- Designing, implementing, and testing functions that build or manipulate the structure of trees.

Degree of Difficulty: Easy. These three functions should be familiar by now. They're just a bit different from the examples in Assignment 7.

References: You may wish to review the following:

- Chapter 20, 21

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Task

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

1. `check_trees(t1, t2)` Purpose: To check if tree `t1` has exactly the same data values in the same places as tree `t2`. ~~If `tnode` is `None`, return `None`. If `tnode` is not `None`, return a reference to the new tree.~~ If the two trees have the same data values in the same locations, this function returns True. If the two trees have different data values, the function returns False. For testing, use some of the trees in the module `treebuilding.py`.
2. `replace(tnode, t, r)` Purpose: To replace 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. For testing, you may use the previous function, `check_trees(t1, t2)`.
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. For testing, you may use the function `check_trees(t1, t2)`.

What to Hand In

- A file `a8q3.py` containing your functions.
- A file `a8q3_testing.py` containing your testing for the functions.

Be sure to include your name, NSID, student number, and course number 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 4 (12 points):

Purpose: Students will practice the following skills:

- Redesigning a bad function to improve robustness.
- Redesigning a bad function to improve its practical efficiency.

Degree of Difficulty: Easy

References: You may wish to review the following:

- Chapter 20, 21

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

Background

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, adapted from some website that pretends to "teach" about Python and trees.

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

Seriously, this is the kind of terrible code you find if you Google for help for CMPT courses. Your instructor added the doc-string, so that the function's purpose is clarified.

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.
- 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.
- This is an example that motivates design for efficiencies that are not reflected in our asymptotic analysis. Preventing useless search does not change the worst case, so the algorithm's worst case time complexity is not improved by preventing this work. However, practically speaking, this is the right way to design a function like this, and can make the difference between a program being practically useful, and practically useless.

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." We've used it before in previous assignments.

Note: 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 the function interface documentation:

```
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:
We've used this technique before, but here is a review.

- 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
```

- To help you test your function, 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.treenode(0, TB.build_fibtree(5), TB.build_complete(10))
```

What to Hand In

- A file called `a8q4.py`, containing the function definition for the function `complete(tnode)`.
- A file `a8q4_testing.py` containing your testing.

Be sure to include your name, NSID, student number, and course number 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 5 (10 points):

Purpose: Students will practice the following skills:

- Working with interesting tree algorithms. This is the kind of algorithm that comes up in practical applications of trees in computer science, including graph theory and artificial intelligence.

Degree of Difficulty: Tricky

References: You may wish to review the following:

- Chapter 20, 21

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

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). While this may seem arbitrary, this is an ordering that naturally arises when you design a function for this purpose.

- 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 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.
  - If 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.
  - If either or both of the values are not in the tree, then `path_to(tnode, value)` will return `(False, None)`, and your function can return None. Remember the lesson from the previous question!

What to Hand In

- The file `a8q5.py` containing the two function definitions:
  - `path_to(tnode, value)`
  - `find_path(tnode, val1, val2)`
- The file `a8q5_testing.py` containing testing for your function.

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

Evaluation

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