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 – Solutions and Grading

Primitive Binary Trees

Date Due: Wednesday, August 4, 11:59pm **Total Marks: 82**

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.

Solution:

```
def nodes_at_level(tnode, level):
    """
    Purpose:
        Counts the number of nodes in the given primitive tree are
        at the given level.
    Preconditions:
        tnode: a treenode object
        level: a non-negative integer
    Return:
        The count of nodes at the level.
    """
    assert level >= 0, "Called with negative level value"

    if tnode is None:
        # empty tree
        return 0
    elif level == 0:
        # reached the desired level
        return 1
    else:
        # look deeper
        return  (nodes_at_level(tnode.left,  level-1)
               + nodes_at_level(tnode.right, level-1))
```

The key here is to count the number of nodes at the level in both subtrees, and then add the two results together. The only trick is to realize that to count nodes at level n, the two recursive calls have to count nodes at level $n-1$.

Another way to say this is that if r is a root node, and l is it's child, then a node at level n starting at the root r is at level $n-1$ starting from l.

This insight leads to 2 base cases: the case where $n = 0$, in which case we don't have to look further down the tree, and the base case where the tree is empty.

```
def largest_leaf_value(tnode):
    """
    Purpose:
        searches the given primitive tree and returns the largest
        data value stored at any leaf node.
    Precondition:
        tnode: a treenode object
    Return:
        The largest leaf value.
    """
    if tnode is None:
        # no tree at all
        return None
    elif tnode.left is None and tnode.right is None:
        # leaf node
        return tnode.data
    elif tnode.left is None:
        # only one branch, on the right
        return largest_leaf_value(tnode.right)
    elif tnode.right is None:
        # only one branch, on the left
        return largest_leaf_value(tnode.left)
    else:
        # two branches
        lmax = largest_leaf_value(tnode.left)
        rmax = largest_leaf_value(tnode.right)
        return max(lmax, rmax)
```

The key here is to look for the largest leaf node on the right, and the largest on the left, and then compare to see which is bigger. The trick here is that some tree-nodes will have 0, 1, or 2 children. So if you know there is no left subtree, don't look down that branch. If a tree-node has no left and no right subtree, then it is a leaf node, and it's the largest leaf node.

The model solution shown is not the only way to implement the function, but it is quite explicit about the possible structures.

```
def closest(tnode, target):
    """
    Purpose:
        search the given primitive binary tree for the data value
        that is closest to the given target.
    Precondition:
        tnode: a treenode object
        target: a number
    Return:
        The data value closest to target.
    """
    if tnode is None:
        # no tree at all
        return None
    elif tnode.left is None and tnode.right is None:
        # leaf node
        return tnode.data
    elif tnode.left is None:
        # one child, the left branch; two values to compare
        rclosest = closest(tnode.right, target)

        if abs(tnode.data - target) < abs(rclosest - target):
            return tnode.data
        else:
            return rclosest

    elif tnode.right is None:
        # one child, the left branch; two values to compare
        lclosest = closest(tnode.left, target)

        if abs(tnode.data - target) < abs(lclosest - target):
            return tnode.data
        else:
            return lclosest

    else:
        # look down both branches; 3 values to compare
        rclosest = closest(tnode.right, target)
        lclosest = closest(tnode.left,  target)

        if (abs(tnode.data - target) < abs(lclosest - target)
            and abs(tnode.data - target) < abs(rclosest - target)):
            return tnode.data
        elif abs(lclosest - target) < abs(rclosest - target):
            return lclosest
        else:
            return rclosest
```

This function reflects the three possible structures: a tree-node can have 0, 1, or 2 subtrees.

As always, we look for the closest value on the left (if it exists), and on the right (if it exists). However, we have to compare these two values with the value at the root, which requires looking at up to three values.

The solution code has 2 base cases, though only one is absolutely necessary: the case for the leaf-node. If an empty tree is given to the function, there is no right answer, and in this case, the function returns None.

Notes to markers: For each of the three functions:

- 3 marks: Execute the score script given.
  - It will try to import `a8q1`; if it cannot, a zero will be reported.
- Check the code for appropriate use of recursion.
  - The function must be recursive, and must examine the tree directly.
  - The function must not use Python list in any way.
  - The function cannot use a global variable to accumulate or remember information about the task.
  - If any of the above are violated, deduct 3 marks from the correctness score reported from the script (without going negative).
- 2 marks: Check that the function interface documentation is appropriate.
  - Check that each recursive call uses arguments that match the Pre-conditions.
  - Check that each return statement matches the Return.
  - Deduct one mark for each aspect of the doc-string that does not match the implementation.

For the testing:

- 5 marks for testing of all three methods. Requirements:
  - Base case tests.
  - Recursive case tests.

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.

Solution:

```
def collect_inorder(tnode):
    """
    Purpose:
        Collect the data values in a list, in in-order sequence.
    Precondition:
        tnode: a treenode object
    Return:
        A list with all the data values.
    """
    if tnode is None:
        return []
    else:
        ll = collect_inorder(tnode.left)
        ll.append(tnode.data)
        rl = collect_inorder(tnode.right)
        ll.extend(rl)
        return ll
```

The values in the tree have to be put into a list. The model solution shows one way to do this. It's probably more common to use the Python method + here, which is fine.

The only requirement is that the data at the root has to be put into the list in the correct position. It's possible to make the recursive calls in any order, as long as the data is returned in the correct order.

```
def collect_preorder(tnode):
    """
    Purpose:
        Collect the data values in a list, in pre-order sequence.
    Precondition:
        tnode: a treenode object
    Return:
        A list with all the data values.
    """
    if tnode is None:
        return []
    else:
        result = [tnode.data]
        ll = collect_preorder(tnode.left)
        rl = collect_preorder(tnode.right)
        result.extend(ll)
        result.extend(rl)
        return result
```

Nothing too interesting here.

```
def collect_postorder(tnode):
    """
    Purpose:
        Collect the data values in a list, in post-order sequence.
    Precondition:
        tnode: a treenode object
    Return:
        A list with all the data values.
    """
    if tnode is None:
        return []
    else:
        ll = collect_postorder(tnode.left)
        rl = collect_postorder(tnode.right)
        ll.extend(rl)
        ll.append(tnode.data)
        return ll
```

Or here.

Notes to markers: For each of the three functions:

- 3 marks: Execute the score script given.
  - It will try to import `a8q2`; if it cannot, a zero will be reported.
- Check the code for appropriate use of recursion.
  - The function must be recursive, and must examine the tree directly.
  - The function must not use Python list in any way.
  - The function cannot use a global variable to accumulate or remember information about the task.
  - If any of the above are violated, deduct 3 marks from the correctness score reported from the script (without going negative).
- 2 marks: Check that the function interface documentation is appropriate.
  - Check that each recursive call uses arguments that match the Pre-conditions.
  - Check that each return statement matches the Return.
  - Deduct one mark for each aspect of the doc-string that does not match the implementation.

For the testing:

- 5 marks for testing of all three methods. Requirements:
  - Base case tests.
  - Recursive case tests.

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.

Solution:

```
def check_trees(tnode1, tnode2):
    """
    Purpose:
        Check if the two trees have same data in the same place
    Precondition:
        tnode1: a treenode object
        tnode2: a treenode object
    Return:
        True if the structure and data is the same; False otherwise
    """
    if tnode1 is tnode2:
        return True
    elif tnode1 is None or tnode2 is None:
        return False
    else:
        return tnode1.data == tnode2.data \
               and check_trees(tnode1.left, tnode2.left) \
               and check_trees(tnode1.right, tnode2.right)
```

In this function, we check first if the two treenodes are exactly the same object. This is not essential for a correct solution, but it can be used to get an answer without checking the whole tree: if the roots are the same, then everything about the tree is the same. This particular base case also succeeds if both trees are empty `None`.

Note the recursive case and the use of the Boolean expression. It's perfectly okay to use a longer if statement to check the various conditions. This is an example of one of the principles mentioned in the chapter on style.

```
def replace(tnode, t, r):
    """
    Purpose:
        Replace each occurence of data value t with r
    Precondition:
        tnode: a treenode object
        t: any value
        r: any value
    Post-Condition:
        The data in the tree is modified
    Return:
        None
    """
    if tnode is None:
        return
    else:
        if tnode.data == t:
            tnode.data = r
        replace(tnode.left,  t, r)
        replace(tnode.right, t, r)
```

The only interesting part is that the function doesn't return anything. It's called for its effect, not its return value. Any function that builds a new tree here is incorrect, and should have 3 marks deducted.

```
def copy(tnode):
    """
    Purpose:
        Make a new tree with the same structure and data
    Precondition:
        tnode: a treenode object
    Return:
        The copied tree
    """
    if tnode is None:
        return None
    else:
        return TN.treenode(tnode.data,
                           copy(tnode.left),
                           copy(tnode.right))
```

Almost nothing of interest here. Keep building copies recursively until you hit an empty tree.

An alternative implementation is acceptable. It's possible to avoid trying to copy an empty tree. It makes the function a bit more complicated, but it's not incorrect.

Notes to markers: For each of the three functions:

- 3 marks: Execute the score script given.
  - It will try to import `a8q3`; if it cannot, a zero will be reported.
- Check the code for appropriate use of recursion.
  - The function must be recursive, and must examine the tree directly.
  - The function must not use Python list in any way.
  - The function cannot use a global variable to accumulate or remember information about the task.
  - If any of the above are violated, deduct 3 marks from the correctness score reported from the script (without going negative).
- 2 marks: Check that the function interface documentation is appropriate.
  - Check that each recursive call uses arguments that match the Pre-conditions.
  - Check that each return statement matches the Return.
  - Deduct one mark for each aspect of the doc-string that does not match the implementation.

For the testing:

- 5 marks for testing of all three methods. Requirements:
  - Base case tests.
  - Recursive case tests.

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.

Solution:

```
def complete(tnode):
    """
    Purpose:
        A tree is complete if all nodes have 2 non-empty
        subtrees, except for the nodes at maximum level,
        which are all leaf nodes.
    Pre-conditions:
        :param tnode: a treenode
    Return
        A tuple (True, height) if the tree is complete,
        A tuple (False, None) otherwise.
    """
    if tnode is None:
        return True, 0
    else:
        lflag, ldepth = complete(tnode.left)
        if not lflag:
            return False, None
        rflag, rdepth = complete(tnode.right)
        if not rflag:
            return False, None
        if ldepth == rdepth:
            return True, rdepth+1
        else:
            return False, None
```

The key idea here is that to be complete, the leaf nodes on both subtrees have to be at the same height. This is expressed in the following test case:

```
def test_4_08():
    test_item = "complete()"
    reason = "largish incomplete tree, with complete subtrees"
    ltree = TB.build_complete(4)
    rtree = TB.build_complete(2)
    atree = TN.treenode(1, ltree, rtree)
    result, height = complete(atree)
    expected = False
    assert result == expected, assert_report.format(test_item, reason, result, expect
```

The test case creates 2 complete trees of different heights (`ltree` and `rtree`), and hangs them from a common treenode (`atree`). This construction does not meet the definition of complete, because the heights are different.

There is a bit of trickiness in catching the two values in the tuple, and I have shown the best way to do this. It's acceptable (but less readable) to use a single variable, and use indexing to access the two different parts of the tuple.

Notice that the right tree is not explored at all if the left tree is not complete. This can potentially save a lot of time!

Notes to markers:

- 5 marks: Execute the score script given.
  - It will try to import `a8q4`; if it cannot, a zero will be reported.
  - The score script will provide a grade out of 8 that combines the 3 points for correctness, and the 5 points for testing.
- Check the code for appropriate use of recursion.
  - The function must be recursive, and must examine the tree directly.
  - The function cannot use a global variable to accumulate or remember information about the task.
  - If any of the above are violated, deduct 3 marks from the correctness score reported from the script (without going negative).
- 5 marks: efficiency.
  - The second subtree (presumably the right tree, but it could be the left) should only be checked if the first subtree is complete. If not, the answer on the second subtree doesn't matter, because the answer won't change.
  - If both subtrees are checked all the time, then deduct all 5 marks.
- 2 marks for testing. Requirements:
  - Base case tests.
  - Recursive case tests.

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.

Solution:

```
def path_to(tnode, value):
    """
    Purpose:
        Return a Python list of all the data values on the path
        from the root to the given value. The returned list is
        ordered with the value first, and the root last.
    Preconditions
        :param tnode: a primitive tree
        :param value: a data value that may or may not be in the tree
    Post-conditions:
        The tree is unaffected.
    Return:
        :return: a tuple (True, alist) if the value is in the tree
                 a tuple (False, None) if the value is not in the tree
    """
    if tnode is None:
        return False, None
    elif tnode.data == value:
        # found it, stop looking!
        return True, [value]
    else:
        # try left first
        lflag, lpath = path_to(tnode.left, value)
        if lflag:
            # got it!
            lpath.append(tnode.data)
            return True, lpath
        # not left, so try right
        rflag, rpath = path_to(tnode.right, value)
        if rflag:
            # got it!
            rpath.append(tnode.data)
            return True, rpath
        else:
            # no path from here, left or right
            return False, None
```

There are 5 things we have to check.

1. There is no path, because the tree is empty.
2. We find a trivial path, with the value at the root.
3. The path goes left; add the root data value to the path.
4. The path goes right; add the root data value to the path.
5. No path right or left.

```
def find_path(tnode, here, there):
    """
    Purpose:
        Find a path between data value here, and data value there
        in a tree. If either of these given values are not in the
        tree, no path is returned.
    Preconditions
        :param tnode: a primitive tree
        :param here: a value
        :param there: a value
    Return:
        :return: A list of node values if the value is in the tree
                 None otherwise
    """

    # look for the two values
    flagh, to_here  = path_to(tnode, here)
    flagt, to_there = path_to(tnode, there)

    if not flagh or not flagt:
        # couldnt find atleast one of the values
        return None

    # got a path from the root to both values.
    # Need to merge the two paths.
    # - easiest to work with the reverse path
    to_here.reverse()
    to_there.reverse()

    # There's always some part of the two paths that are the same
    # the last value that's the same is the Least Common Ancestor
    # The LCA is always on the path, but nothing above it is
    lca = 0
    while (lca < len(to_here) and lca < len(to_there)
           and to_here[lca] == to_there[lca]):
        lca += 1

    # compose the path by stripping everything off before the lca
    first_bit = to_here[lca-1:]
    second_bit = to_there[lca:]

    # to start the path, we'll undo the reverse of the first few nodes
    first_bit.reverse()

    # now return the joined path
    return first_bit + second_bit
```

This algorithm an be best understood by drawing a diagram. If a path exists between two nodes, then the path "goes" up the tree from one of the nodes, and reaches a node that is an ancestor of both, then "goes" down the tree to the other value.

We can construct this path by finding 2 paths from the root, one to each node. The paths will start the same, because they are in the same tree. But at some point the two paths will converge. The node at which the two paths diverge is called the lowest common ancestor.

The function finds this point, and keeps only the parts that are different.

Notes to markers:

- 4 marks: `path_to()`
  - Execute the score script given. It will try to import `a8q5`; if it cannot, a zero will be reported.
  - The score script will provide a grade out of 5 that combines the 4 points for correctness, and 1 of the points for testing.
  - The function must be recursive, and must examine the tree directly.
  - The function cannot use a global variable to accumulate or remember information about the task.
  - If any of the above are violated, deduct 3 marks from the correctness score reported from the script (without going negative).
- 3 marks: `find_path()`
  - Execute the score script given. It will try to import `a8q5`; if it cannot, a zero will be reported.
  - The score script will provide a grade out of 5 that combines the 4 points for correctness, and 1 of the points for testing.
  - This function is not recursive.
  - The function needs to call `path_to()` twice, and use the information to construct the path.
- 3 marks for testing both methods. Requirements:
  - Base case tests for `path_to()`
  - Recursive case tests for `path_to()`
  - Test cases for `find_path()` should include paths that go through the root, and some that don't.