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

Binary trees

Date Due: March 23, 2018, 10pm **Total Marks: 45**

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. This is a tool that you need to be required to use, even when you don't need it, so that when you do need it, you are already familiar with it. 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.
   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A8 folder, and run `git --no-pager log` there. If you did all your work in this folder, git will be able to see it even if you did your work on Windows. Git's information is out of sight, but in your folder.
   Note: If you are working at home on Windows, Google for how to make git available on your command-line window. You basically have to tell the command-line app where the git app is.

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.

Solution: The submitted log might in the form of a text file, or a couple of screenshots or camera grabs of the PyCharm interface. Ideally, the log file was submitted, but there were enough problems with Windows and setting the path properly that we have to be a little generous here.

Notes to markers:

- There are two things to look for: the number of commits and the quality of the commit messages.
- The meaningful commit messages criterion: a message has to be about the work. Implemented a function, fixed a bug, started a question, added testing, those kinds of things.
- Give 3 marks for good commit messages; 1 mark for anything else.
- Students were told at least 2 commits per question.
- I don't want you counting commits very carefully. More than 10 commits is worth 2 marks, and 1 marks if there are not even 10 commits.

Question 1 (16 points):

Purpose: To practice recursion on binary trees.

Degree of Difficulty: Easy to Moderate to Tricky

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

1. `subst(tnode, t, r)` Purpose: To substitute a target value t with a replacement value r wherever it appears in the given tree. Returns `None`, but modifies the given tree.
2. `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.
3. `collect_data_inorder(tnode)` Purpose: To collect all the data values in the given tree. Returns a list of all the data values, and the data values appear in the list according to the in-order sequence. Hint: Base this function on the `in_order()` traversal, which you can find in `traversals.py`.
4. `count_smaller(tnode, target)` Purpose: Count the number of data values in the given tree that are less than the given target value. Assume that the given tree has data values that are number only.

On Moodle you will find a file called `exampletrees.py` which has a few example trees that you can use for demonstration and testing. This file also recommends a demonstration strategy for each function.

You'll also find some useful functions in the file `treefunctions.py`.

Note: Test your functions carefully. You may start with the trees in the Python file provided, but those are demonstrations. Your testing could and probably should be a bit more deliberate.

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:

- 1 mark: Your function has a good doc-string.
- 1 mark: Your function has a correct base case.
- 1 mark: Your function's recursive case is correct.
- 1 mark: You tested your function adequately.

Solution:

`subst(tnode, t, r)`
A model solution is as follows:

```
def subst(tnode, t, r):
    """
    Purpose:
        Replace every occurrence of data value t with r in the tree
    Pre-conditions:
        :param tnode: a treenode
        :param: t: a target value
        :param: r: a replacement value
    Post-conditions:
        Every occurrence of the target value is replaced
    Return
        :return: None
    """
    if tnode == None:
        return
    else:
        if tn.get_data(tnode) == t:
            tn.set_data(tnode, r)
        subst(tn.get_left(tnode), t, r)
        subst(tn.get_right(tnode), t, r)
```

Notes for markers:

- A function this simple should only need one base case. Deduct a mark if there is more than one base case.
- The recursive calls do not need to be ordered in any particular way.
- Testing should include the following equivalence classes:
  - The empty tree. `subst()` has no effect.
  - A non-empty tree, with the target appearing multiple places.
  - A non-empty tree with the target not appearing at all.
  - More testing is permitted, but without these 3, it's inadequate.

`copy(tnode)`
A model solution is as follows:

```
def copy(tnode):
    """
    Purpose:
        Make a copy of the tree rooted at the given tnode.
    Pre-conditions:
        :param tnode: A treenode
    Return:
        :return: A copy of the tree.
    """
    if tnode is None:
        return None
    else:
        return tn.create(tn.get_data(tnode),
                         copy(tn.get_left(tnode)),
                         copy(tn.get_right(tnode)))
```

Notes for markers:

- A function this simple should only need one base case. Deduct a mark if there is more than one base case.
- Several variations are allowable. Students may use assignment statements to capture the return value from the recursion, and do the work in several steps.
- Testing should include the following equivalence classes:
  - The empty tree.
  - A non-empty tree of at least 2 levels, with some subtrees empty.
  - More testing is permitted, but without these 2, it's inadequate.
  - A good test would check if the tree and its copy are equal (==) but different (`is not`).

`collect_data_inorder(tnode)`
A model solution is as follows:

```
def collect_data_inorder(tnode):
    """
    Purpose:
        Return a list of data values with inorder sequence.
    Pre-conditions:
        :param tnode: a treenode
    Return
        :return: A list of data values, with inorder sequence
    """
    if tnode is None:
        return []
    else:
        return (collect_data_inorder(tn.get_left(tnode))
                + [tn.get_data(tnode)]
                + collect_data_inorder(tn.get_right(tnode)))
```

Notes for markers:

- A function this simple should only need one base case. Deduct a mark if there is more than one base case.
- Several variations are allowable. Students may use assignment statements to capture the return value from the recursion, and do the work in several steps.
- A common error is to use `append()` incorrectly, putting lists inside lists. It is possible to use `append()` correctly, or `extend()`. But list concatenation is the simplest.
- Testing should include the following equivalence classes:
  - The empty tree.
  - A non-empty tree of at least 2 levels, with some subtrees empty.
  - More testing is permitted, but without these 2, it's inadequate.

`count_smaller(tnode, target)`
A model solution is as follows:

```
def count_smaller(tnode, target):
    """
    Purpose:
        Count the number of times a target value appears in the tree
        rooted at tnode.
    Pre-conditions:
        :param tnode: a treenode
        :param target: a value
    Return:
        The number of times a value appears in the tree.
    """
    if tnode is None:
        return 0
    else:
        total = count_smaller(tn.get_left(tnode), target) \
              + count_smaller(tn.get_right(tnode), target)
        if tn.get_data(tnode) < target:
            return 1 + total
        else:
            return total
```

Notes for markers:

- A function this simple should only need one base case. Deduct a mark if there is more than one base case.
- Several variations are allowable. Students may break the recursive case into 2 recursive cases depending on the value stored in the data filed of the node. This is not a problem.
- Testing should include the following equivalence classes:
  - The empty tree.
  - A non-empty tree of at least 2 levels, with a variety of values.
  - More testing is permitted, but without these 2, it's inadequate.

Question 2 (8 points):

Purpose: To do more thinking about 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.

Write a function `mirrored(t1, t2)` that returns `True` if the two given trees satisfy the mirror property, and `False` otherwise.

What to Hand In

- A file called `a8q2.py`, containing your function definition.
- A file called `a8q2_testing.py`, containing your testing.

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

Evaluation

- 4 marks: Your function works.
- 4 marks: Your testing is adequate.

Solution: A model solution is as follows:

```
def mirrored(t1, t2):
    """
    Purpose:
        Determine if t1 and t2 are mirrored.
    Pre-conditions:
        :param t1: a treenode
        :param t2: a treenode
    Return:
        :return: True if they are mirrored, False otherwise
    """
    if t1 is None and t2 is None:
        return True
    elif t1 is None or t2 is None:
        return False
    else:
        return (tn.get_data(t1) == tn.get_data(t2)
                and mirrored(tn.get_left(t1), tn.get_right(t2))
                and mirrored(tn.get_right(t1), tn.get_left(t2)))
```

Notes for markers:

- There are 2 base cases in the model solution, but students may break the `False` case into 2 separate cases (one is empty and the other is not).
- Students may use if-statements instead of `and` for the recursion. That's fine, but a little clumsy.
- Testing should include the following equivalence classes:
  - Two empty trees (one case)
  - One empty tree, the other not (2 cases)
  - Two non-empty trees, at least 2 levels, with some subtrees empty, and not mirrored.
  - Two non-empty trees, at least 2 levels, with some subtrees empty, and mirrored.
  - More testing is permitted, but without these 5 cases, it's inadequate.

Question 3 (8 points):

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

Degree of Difficulty: Moderate

Write a function `complete(tnode)` that returns `True` is the given tree is a complete binary tree, and `False` otherwise.

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 nodes at the maximum level, where the nodes are barren (i.e., leaf nodes).

Visually, complete binary trees are easy to detect. But if you're just given a root node, it will be harder.

Consider the function below:

```
1  def bad_complete(tnode):
2      def cmplt(tnode):
3          if tnode is None:
4              return 0
5          else:
6              ldepth = cmplt(tn.get_left(tnode))
7              rdepth = cmplt(tn.get_right(tnode))
8              if ldepth == rdepth:
9                  return rdepth+1
10             else:
11                 return -1
12     result = cmplt(tnode)
13     return result > 0
```

Notice the internal definition of `cmplt()`. This function is almost identical to our `height` function except for lines 8-11. The function uses the integer value -1 to report an incomplete tree. Whether this use of the return value is good or bad is a matter of opinion, but without an alternative, we make a virtue of necessity, which is not always good!

Rewrite the internal definition to 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." You'll also have to change the second-to-last line of the function, line 12, to account for the change to the internal function.

Hint: Your internal function 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
```

What to Hand In

, with:

- The file `a8q3.py` containing the function definition, using tupling, and an internal function.
- The file `a8q3_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

- 2 marks: you used an internal function to do most of the work
- 3 marks: your internal function correctly uses tupling
- 1 mark: your function calls the internal function correctly, and returns a Boolean value only.
- 2 marks: your testing is adequate.

Solution: A model solution is as follows:

```
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
        :return: True if the tree is complete, False otherwise.
    """
    def cmplt(tnode):
        """
        :return: (True, height) if tnode is complete
                 (False, None) is tnode is not complete
        """
        if tnode is None:
            return True, 0
        else:
            lflag, ldepth = cmplt(tn.get_left(tnode))
            if not lflag:
                return False, None
            rflag, rdepth = cmplt(tn.get_right(tnode))
            if not rflag:
                return False, None
            if ldepth == rdepth:
                return True, rdepth+1
            else:
                return False, None
    result, _ = cmplt(tnode)
    return result
```

There are two important things going on here. First is tupling. With tuples, and tuple assignment, there is no reason to play silly games with return values like the original. The second is more subtle. The original function always checked the right subtree, even if the left subtree was not complete. There is no need to check in that situation.

Notes for markers:

- The internal function should return a tuple, as in the model solution.
- Some students will prefer to use indexing of tuples instead of tuple assignment. I don't get it, because it's ugly, but it's acceptable.
- A common error will be that the tuple is not examined as separate elements.
- The question did not say anything about efficiency, so the unnecessary work should not be penalized.
- Testing should include the following equivalence classes:
  - An empty tree
  - A non-empty tree, at least 2 levels, not complete.

- A non-empty tree, at least 2 levels, complete.
- More testing is permitted, but without these 3 cases, it's inadequate.

Question 4 (8 points):

Purpose: To do more thinking about binary trees.

Degree of Difficulty: Tricky Tricky Tricky

We say that a binary tree is *ordered* if all of the following conditions are true:

1. The left subtree is ordered.
2. The right subtree is ordered.
3. If the left subtree is not empty, all of the data values in the left subtree are less than the data value at the root.
4. If the right subtree is not empty, all of the data values in the right subtree are greater than the data value at the root.

The empty tree is *ordered* by definition.

Write a function `ordered(tnode)` that returns `True` if the given tree is *ordered*, and `False` otherwise.

What to Hand In

- A file called `a8q4.py`, containing your function definition.
- A file called `a8q4_testing.py`, containing your testing.

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

Evaluation

- 4 marks: Your function works.
- 4 marks: Your testing is adequate.

Solution: A model solution is as follows:

```
def ordered(tnode):
    """
    Purpose:
        Determine if the data values in the tree are ordered.
    Preconditions:
        :param tnode: a treenode
    Return
        :return: True if the data are ordered, False otherwise
    """
    def check(tree):
        """
        :param tree: a non-empty treenode
        :return: True, min, max, if ordered
                 False, None, None, if not ordered
        """
        # convenience!
        data = tn.get_data(tree)
        left = tn.get_left(tree)
        right = tn.get_right(tree)

        if left is None and right is None:
            return True, data, data
        elif left is None:
            # just look right
            flag, rmn, rmx = check(right)
            if flag and data < rmn:
                return True, data, rmx
            else:
                return False, None, None
        elif right is None:
            # just look left
            flag, lmn, lmx = check(left)
            if flag and data > lmx:
                return True, lmn, data
            else:
                return False, None, None
        else:
            # look right first
            rflag, rmn, rmx = check(right)
            if not rflag or data > rmn:
                return False, None, None
            # only look left if the right checks out
            lflag, lmn, lmx = check(left)
            if not lflag or data < lmx:
                return False, None, None
            # ordered!
            return True, lmn, rmx

    # body of ordered()
    if tnode is None:
        return True
    else:
        flag, _, _ = check(tnode)
        return flag
```

This is the kind of recursive function on trees that makes us appreciate simple recursion.

The function's correctness comes from the fact that the data value at a node has to be greater than every node in the left subtree, and less than every value in the right subtree (we're ignoring equality for simplicity). So to answer the question, we need to know the min and max values in both subtrees.

We also need to know if the subtrees themselves are ordered. This combination of information looks like a nice job for tupling!

The only real problem is that empty subtrees don't have a well-defined min and max data value. So we have to look at 4 cases:

1. The node has no children. The min and max values are just the data itself.
2. The node has a left child, but not a right. So only look left.
3. The node has a right child, but not a left. So only look right.
4. The node has both children.

The time complexity is $O(N)$: you have to look at each node exactly once, and do a bit of work at each node to figure out the answer. We're not building any lists, so we are using as little memory as possible. The call stack grows, but with a balanced tree, no more than $O(\log N)$ stack frames, and with a degenerate tree, the worst case is $O(N)$ stack frames.

There are many other algorithms to solve this problem, but many of them are terrible! For example:

- Just use `collect_data_inorder(tnode)` from Q1 exactly once, and check to see if it's equal to the sorted version of itself. This is terrible because it does too much work (collecting, sorting, and comparing), and takes too much memory (the data values are all stored in 3 different lists!). This method is also not recursive.
- Use `collect_data_inorder(tnode)` at every node in the tree. This is bad for all the reasons above, but worse because it happens repeatedly! The time complexity is roughly $O(N^2 \log N)$
- Writing functions `treemin()` and `treemax()`, which examine whole trees for the min and max data values. If these are called at every node, a lot of work gets repeated over and over again, and that will multiply the effort by a factor of $O(N)$. The tupling technique avoids this repeated calculation.

Notes for markers:

- This is a very tricky function to write, and will be tricky to grade.
- The evaluation did not mention efficiency, so that cannot be a factor in grading. This was my error, but it's fair to grade according to the stated rubric.
- The description did not require a recursive function, so that cannot be a factor in grading. This was my error, but it's fair to grade according to the stated rubric.
- If you can spot an error by looking at the code briefly, deduct marks. If not, try running the test script, and see.
- Testing should include the following equivalence classes:
  - An empty tree.
  - A singleton tree.
  - Examples of 2 level trees with 1 or 2 children, that are ordered.
  - Examples of 2 level trees with 1 or 2 children, that are not ordered.

- Examples of trees with more than 2 levels, complete and incomplete, that are ordered.
- Examples of trees with more than 2 levels, complete and incomplete, that are not ordered.
- More testing is permitted, but without these 6 cases, it's inadequate.