Assignment 8

Binary Trees

Date Due: 20 March February 2020, 8pm **Total Marks: 71**

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

- **17/03/2020**: Q3: Removed the requirement that all functions be recursive.
- **16/03/2020**:Q1.4: Clarified that `min_level()` returns the height.
- **14/03/2020**: released to students

Primitive Binary Trees

In this assignment question, 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 tree functions are provided in the document `treefunctions.py`.
  - `is_leaf(tnode)` It's very common to want to check if a node is a leaf node, so this function gets the job done without having to type a long condition.
  - `to_string(tnode)`, which takes a single primitive tree and returns a string that you can print to see the structure of the tree more easily. Easier than printing out the nested dictionary directly!
- Some functions to create binary trees are found in `treebuilding.py`
  - `build_lecture_example()` returns a tree with the structure of the example found in the lecture slides
  - `build_turtle()`, `build_tuttle()` each return a smallish tree, and these two trees are almost identical, except for one node.
  - `build_xtree_me()` returns a slightly larger tree
  - `treeify(alist)` builds a tree using the values in the given list. The left and right subtrees are close to equal in height and in the number of nodes.
  - `build_complete(height)` builds a complete tree with the given height. The data values are numbers.
  - `build_fibtree(n)` builds a tree whose data values are fibonacci numbers.

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.~~ If you are working on a Windows computer and you haven't installed Git, take a few minutes to install it. You may need to configure PyCharm's Preferences to be aware of the installation.

What to Hand In

After completing and submitting your work for Questions 1-3, 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-gitlog.txt`.

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 function that you submit for grading. The markers will not be counting rigorously, but it will be obvious if you are using Git inappropriately.

Question 1 (24 points):

Purpose: To practice recursion on binary trees.

Degree of Difficulty: Easy to Moderate.

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.

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 as a data value in the given tree. Returns `None`, but modifies the given tree.
2. `count_target(tnode, target)` Purpose: Counts the number of times the given target appears in the given tree. If `target` does not appear, a zero count is returned.
3. `sum_leaf_values(tnode)` Purpose: returns the sum of all the leaf values in the given tree. If the given `treenode` is empty, this function should return 0. This function should only be applied to trees containing numeric data values.
4. `min_level(tnode)` Purpose: Returns the height of the leaf node with the smallest level. If the primitive binary tree is empty, the function should return `0`.
   Note: `min_level(tnode)` returns the *height*, and so its name is imprecise.

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:

- 3 marks: Your function is recursive and correct.
  - You will get 0 marks if your function does not have a doc-string to document the function interface according to CMPT 145 standards.
- 3 marks: Your testing for the function is adequate.
  - You should provide test cases that are designed from blackbox and whitebox perspectives
  - You should attempt to identify test case equivalence classes
  - You may use `pytest`, but this is not required.

Question 2 (24 points):

Purpose: To explore the application of binary trees to the representation of arithmetic expressions.

Degree of Difficulty: Moderate

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.

In Chapter 9, we learned how to use Stacks to evaluate an arithmetic expression written using *post-fix* notation. You might recall the example from the lecture:

The following example evaluates to 42:

$$3\,4\,\times\,5\,6\,\times\,+$$

In Assignment 4, we learned how to use Stacks to evaluate an arithmetic expression written using a more normal kind of notation, but with lots of brackets:

```
example1 = '( 1 + 2 )'                # should evaluate to 3
example2 = '( ( 11 + 12 ) * 13 )'     # should evaluate to 299
```

In this question, we will use trees to represent an arithmetic expression. We will call these expression trees, and we will build them using our Tree Node ADT. An expression tree is defined as follows.

- Either a single tree node, whose data is a string representing a numeric quantity, and whose left and right subtrees are both `None`.
- Or a tree consisting of a root node, whose data is one of the following strings: `'+'`, `'-'`, `'*'`, `'/'`, and whose left and right subtrees are both expression trees.

Note that we exclude empty expression trees from the definition.

For example, the following are some simple expression trees that correspond to the examples above:

```
import treenode as TN

# a single node is an expression tree
example0 = TN.create('3.5')

# if the data value is an operation, there are always 2 children
example1 = TN.create('+', TN.create('1'), TN.create('2'))

# expression trees can be nested
example2 = TN.create('*', TN.create('+', TN.create('11'), TN.create('12')),
                          TN.create('13'))

# all data values are strings
example3 = TN.create('+', TN.create('*', TN.create('3'), TN.create('4')),
                          TN.create('*', TN.create('5'), TN.create('6')))
```

You should draw these expression trees by hand to understand their structure. In our work here, all treenodes will have strings as data values. In a leaf node, the string will look like a number; all other nodes have strings as data values, but these will look like operators. Keeping everything as a string in the tree means there is never any question about what data type it is.

This topic is important to computer science because programming language compilers and interpreters use expression trees to represent expressions in programs and scripts. Python, for example, will read a script, and will turn an expression written in Python syntax into an expression tree, and then Python will evaluate the expression tree. In C or Java, a compiler will read a program, convert arithmetic expressions to expression trees, and then convert the expression tree into machine language statements. Our work in this assignment question is to look at a very small part of that application.

Tasks

1. Write and test a function `evaluate(etree)` that calculates the result of the arithmetic expression represented by an expression tree. For example:

```
>>> print(A8.evaluate(example3))
42.0
```

   The variable `example3` refers to the expression tree examples on the previous page.

2. Write and test a function `to_string_infix(etree)` that returns a string representation of a given expression tree. The result of this function should look like the kinds of expressions we used in Assignment 4. For example:

```
>>> print(A8.to_string_infix(example3))
( ( 3 * 4 ) + ( 5 * 6 ) )
```

3. Write and test a function `to_string_postfix(etree)` that returns a string representation of a given expression tree. The result of this function should look like the kinds of expressions we used in Chapter 9. For example:

```
>>> print(A8.to_string_postfix(example3))
3 4 * 5 6 * +
```

4. Write and test a function `to_string_prefix(etree)` that returns a string representation of a given expression tree. Prefix notation is a lot like postfix notation, except the operator is placed `before` the two operands. For example:

```
>>> print(A8.to_string_prefix(example3))
+ * 3 4 * 5 6
```

What to Hand In

- A file called `a8q2.py`, containing the definitions for the function described above.
- A file `a8q2_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

Each function will be graded as follows:

- 3 marks: Your function is recursive and correct.
  - You will get 0 marks if your function does not have a doc-string to document the function interface according to CMPT 145 standards.
- 3 marks: Your testing for the function is adequate.
  - You should provide test cases that are designed from blackbox and whitebox perspectives
  - You should attempt to identify test case equivalence classes
  - You may use `pytest`, but this is not required.

Question 3 (18 points):

Purpose: To continue working with expressions as applications of binary trees.

Degree of Difficulty: Tricky

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.

In the previous question in this homework assignment, you wrote some simple recursive functions for expression trees. You were given some trees to work with, and the programs used them as inputs. In this question, your job is to build an expression trees from strings.

Tasks

1. Write and test a function `parse_infix(estring)` that returns an expression tree given a string representation of a an expression using infix notation. For example:

```
>>> etree = A8.parse_infix('( ( 3 * 4 ) + ( 5 * 6 ) )')
>>> print(A8.evaluate(etree))
42.0
```

   The etree was not displayed in the above example because it would be a mess to look at!
   Hint: The program from Assignment 4 for evaluating an infix expression can be adapted to build trees instead of evaluation. One of the stacks can be used to store expression trees, rather than numeric values.

2. Write and test a function `parse_postfix(estring)` that returns an expression tree given a string representation of a an expression using postfix notation. For example:

```
>>> etree = A8.parse_postfix('3 4 * 5 6 * +')
>>> print(A8.evaluate(etree))
42.0
```

   Hint: The program from Chapter 9 for evaluating a postfix expression can be adapted to build trees instead of evaluation. The stack can be used to store expression trees, rather than numeric values.

3. Write and test a function `parse_prefix(estring)` that returns an expression tree given a string representation of a an expression using prefix notation. For example:

```
>>> etree = A8.parse_prefix('+ * 3 4 * 5 6')
>>> print(A8.evaluate(etree))
42.0
```

   Hint: The easy way to solve this is to use `parse_postfix(estring)`. However, there is a very interesting recursive solution to this problem. It requires a main function that sets up the inputs to a recursive helper function.

What to Hand In

- The file `a8q3.py` containing the three function definitions as described above.
- 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

Each function will be graded as follows:

- 3 marks: Your function is ~~recursive and~~ correct.
  - You will get 0 marks if your function does not have a doc-string to document the function interface according to CMPT 145 standards.
- 3 marks: Your testing for the function is adequate.
  - You should provide test cases that are designed from blackbox and whitebox perspectives
  - You should attempt to identify test case equivalence classes
  - You may use `pytest`, but this is not required.