Assignment 10

Binary Search Trees, and Huffman Coding

Date Due: 9 August 2019, 11pm **Total Marks: 32**

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

- **02/08/2019**: released to students

Question 1 (21 points):

Purpose: To adapt some working code to a slightly modified purpose.

Degree of Difficulty: Moderate

In class we discussed binary search trees, and basic operations on them. The code for these operations can be found in the file `bstprim.py` on the Assignment 10 Moodle page. To get you to study the code, and to understand what it is doing, we have introduced a few bugs into the code. The bugs are not devious, but are errors that you should be able to fix by understanding the code.

As we also discussed in class, the `KVTreeNode` class is variant of the `treenode` class. The key-value treenode allows us to organize the data according to a key, and store a data value associated with it. You can find the implementation in file `KVTreeNode.py`.

Make a copy of `bstprim.py`, and call it `a10q1.py`. Adapt the functions from `bstprim.py` to use the `KVTreeNode` class. The `KVTreeNode`s in the tree should have the binary search tree property on the keys, but not the values. The functions you need to adapt are as follows:

member_prim(t, k) Returns the tuple `True, v`, if the key `k` appears in the tree, with associated value `v`. If the key `k` does not appear in the tree, return the tuple `False, None`.

insert_prim(t, k, v) Stores the value `v` with the key `k` in the Table `t`.

- If the key `k` is already in the tree, the value `v` replaces the value currently associated with it. In this case, it returns the tuple `(False, t)` even though `t` did not change structure in this case.
- If the key is not already in the tree, the key and value are added to the tree. In this case the function returns the tuple `(True, t2)`. Here, `t2` is the same tree as `t`, but with the new key, value added to it.

delete_prim(t,k) If the key `k` is in the tree `t`, delete the node containing `k` (and its value) from the tree and return the pair `(True, t2)` where `t2` is the tree after deleting the key-value pair; return the pair `(False, t)` if the key `k` is not in the given tree `t` (`t` is unchanged).

List of files on Moodle for this question

- `a10q1.py` — partially completed
- `bstprim.py` — the BST operations, using `TreeNode`.
- `TreeNode` — The simpler BST treenode class. Used by `bstprim.py`; provided in case you want to use bstprim.
- `KVTreeNode` — The key-value BST treenode class. For use with `a10q1.py`. It's already in final form.
- `test_a10q1.py` — A script that will help you check your progress on `a10q1.py`.

Testing

A test script `test_a10q1.py` is available on the Assignment 10 Moodle page for your use to check your progress.

What to Hand In

Your implementation of the primitive BST functions adapted to use KVTreeNodes, in a file named `a10q1.py`.

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

Evaluation

- 7 marks: `member_prim` is correctly adapted.
- 7 marks: `insert_prim` is correctly adapted.
- 7 marks: `delete_prim` is correctly adapted.

Question 2 (11 points):

Purpose: To implement the Table ADT, as described in class.

Degree of Difficulty: Should be Easy

In this question, you'll implement the full Table class, as we discussed in class. The Table class has the following methods:

size() Returns the number of key-value pairs in the Table.

is_empty() Returns `True` if the Table `t` is empty, `False` otherwise.

insert(k,v) Stores the value `v` with the key `k` in the Table. If the key `k` is already in the Table, the value `v` replaces any value already stored for that key.

retrieve(k) If the key `k` is in the Table, return a tuple `True, value`, where `value` is the value stored with the given key; otherwise return the tuple `False, None`.

delete(k) If the key `k` is in the Table, delete the key-value pair from the table and return `True`; return `False` if the key `k` is not in the Table.

A starter file has been provided for you, which you can find on Moodle named `a10q2.py`. The `__init__` method is already implemented for you, and all of the others have an interface and a trivial do-nothing definition. In addition, we've provided a scoring script for you to check your progress.

To complete this question, you may use `import a10q1` to import your solution to Question 1 into the Table ADT. Your Table methods can call the primitive BST functions. Your Table methods may have to do a few house-keeping items (such as change the size attribute when inserting or deleting), but most of the work is done by your solution to Question 1.

List of files on Moodle for this question

- `a10q2.py` — partially completed
- `test_a10q2.py` — A script that will help you check your progress on `a10q2.py`.

What to Hand In

Your implementation of the Table ADT in a file named `a10q2.py`.

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

Evaluation

- 1 mark: `size`: Your implementation correctly returns the number of key-value pairs stored in the table.
- 1 mark: `is_empty`: Your implementation correctly returns the number of key-value pairs stored in the table.
- 3 marks: `retrieve`: Your implementation correctly searches for a key in the table, and returns the value if found.
- 3 marks: `insert`: Your implementation correctly adds a key-value pair to the table.
- 3 marks: `delete`: Your implementation correctly removes a key-value pair from the table.

Question 3 (15 points):

THIS IS A BONUS QUESTION! It means that you can get over 100% in this assignment.

Purpose: To practice the skill of reading and modifying someone else's code. To recognize that efficiency of memory use is sometimes as important as computational efficiency.

Degree of Difficulty: Moderate

On the Assignment 10 Moodle page, you'll find the program `Decoder.py`. It is a fully functional program that is able to decode files based on a expected file format.

The program follows a simple design, and is well-documented. It contains 4 functions, as follows:

main() : The main program.

1. Reads the entire file.
2. Extracts the code.
3. Extracts the encoded message
4. Builds the codec
5. Decodes the message.

build_decoder(code_lines) Build the dictionary for decoding starting with the given list of code-lines.

decode_message(coded_message, codec) : Decode the message using the decoder.

read_file(fname) : Reads every line in the named file, putting all the lines into a single list.

However, there is a serious efficiency problem with this program. The design requires that the entire file be read at once, and that all the data get passed from one function to another. For example, the function `read_file()` reads the entire file's contents, and stores it all in a single list of strings. From there, the lines of encoded text are decoded one at a time, but all the encoded text is stored in a list.

For small files, this is okay, since modern computers are fairly big. However, for very large files, bigger than what we're working with, it is very inefficient to store the whole file's worth of data in memory all at once. It is better, when it is possible, avoid this situation.

Your job is to take the given code, and rewrite parts of it, so that the program does not store all the data in the file all at once. This task will require some study of the current code, and some redesign. You do not need to write new algorithms. You just need to arrange it so that the program is not storing all file's data at the same time.

For this question, Version Control may be especially useful, as you will want to save your program at various stages of development while you rewrite it. Whether you actually use the history of your development or not, ensuring that you could is a professional attitude.

A small collection of example files are also provided to you to test your program. Phoenix has included some his thoughts that have been encoded.

What to Hand In

- A file named `a10q3.py` containing the revised program.
- A file named `a10q3-mystery.txt` containing the decoded output from the example files (`encoded-story1.txt`, `encoded-story2.txt` and `encoded-story3.txt`).

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

Evaluation

- 8 marks: Your program correctly decodes encoded files, and does not store the entire contents of the encoded file in memory at the same time.
- 5 marks: Your program is well-designed, documented, and shows evidence of defensive programming.
- 2 marks: Phoenix's thoughts in `a10q3-mystery.txt`.