COSC122 2023 Semester 2
INTRODUCTION TO COMPUTER SCIENCE

Assignment 2: Diagnosing Genetic Diseases

6% of overall grade
Due Fri 29 Sep 2023, 11:55pm

1 Overview

Note: this section has the same information as Section 1 in Assignment 1, except for the due date; the new material starts in Section 2.

1.1 Introduction

Modern medicine is increasingly moving towards personalised diagnosis and treatments. A large part of this is based on DNA sequencing, which has seen significant growth over the past two decades. What was once a prohibitively expensive task, sequencing the entirety of your genome is now available as an affordable online service.

In these assignments we will be using algorithms to learn about, and help treat, patients based on their genes. While short DNA sequences such as that of a fly are so small that almost any algorithm will be sufficient on a modern machine, humans are orders of magnitude more complicated. Thus the algorithms we use need to be able to scale to look at hundreds of millions of base pairs, and as computer science students you will be the ones designing the tools for the next generation of medical researchers.

Throughout these assignments, when we represent DNA, we only ever store one half of the strand—the first half always unambiguously implies the other. Indeed, this is exactly what a cell does when it is replicating. For our purposes, we will consider a gene to be a sequence of 32 bases/letters—this is about a thousand times shorter than genes usually are, but still gives us $4^{32} \approx 1,844$ quintillion different genes to work with.

1.2 Due date

Assignment 2 has one due date: Fri 29 Sep 2023, 11:55pm.

Note: You can submit it up to one week after the due date, but you will incur a 15% absolute penalty (in other words, your mark will be the raw mark less 15 marks (given the assignment is marked out of 100). Assignments *cannot* be submitted more than one week after the

deadline, unless you've organised an extension with the course supervisor because of illness etc.

1.3 Submission

Submit via the quiz server. The submission page will be open closer to the due date.

1.4 Implementation

All the files you need for this assignment are on the quiz server. Do not import any additional libraries unless explicitly given permission within the task outline. For each section there is a file with a function for you to fill in. You need to complete these functions, but you can also write other functions that these functions call if you wish. All submitted code needs to pass PYLINT program checking.

1.5 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Moodle, as this enables everyone to see responses to common issues, but **never** post your own program code to Learn. Remember that the main point of this assignment is for you to exercise what you have learnt from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

2 Diagnosing patients via their genome

A person's DNA can tell us a lot about what is happening inside their body, and more importantly what may one day happen. In Assignment 2 we will be working with people that do not yet show symptoms of any disease, using their genomes to provide early detection. This assignment revolves around hash tables & binary search trees (BST), and how they can be used to efficiently search large amounts of data. The result will be a system that can read in a file containing genetic disorder data, and then be queried to find out if a patient is suffering from a genetic disorder recorded in the table. Note, the approach used in this assignment is an *oversimplification* of real world practice and is not medical advice!

For each of the first two tasks of this assignment, you will be completing the definitions of hash table classes that store `(Gene, str)` pairs so we can match genes to diseases. To complete the definition of a hash table, you must fill in the `__getitem__`, and `insert` methods.

For the third task, you will store `(Gene, str)` in a BST class with the same methods, and some extra BST utility functions.

For each hash table class you should maintain two counters as instance attributes: `hashes`, which counts the number of times a `Gene` object was hashed; and `comparisons`, which counts how many times two `Gene` objects are compared against one another. The BST class only requires a `comparisons` instance variable.

You will need to use specific data structures (e.g. `Gene`), located in `classes2.py`. You do not have to implement these data structures, you will just have to interact with them. Details of these data structures are in the Provided Classes section below.

You are expected to test your code using **both** your own tests and the provided tests. More information about testing your code is in the Testing your Code section later.

2.1 Provided classes

The `classes2.py` module contains three useful classes for you to work with: one from the first assignment (`Gene`), and four new classes ...

`GeneLinkedList` : a purposefully minimal implementation to ensure you understand the internal mechanism of linked lists. It forms the chains in a `ChainingGeneHashTable`. Each link in the chain is made of a `GeneLink`, starting at the `head` (if the list is empty, `head` is `None`).

`GeneLink` : a single node in a `GeneLinkedList`. Each has some `data` (probably a `(gene, disease)` pair) and a pointer to the `next` node. If there is no next link, the value of `next` is `None`.

Example: simplified image of Python tutor demo.

Example: interactive Python Tutor demo.

`GeneBstNode` : a single node in a `GeneBST`. Each has: a `key` (`Gene`), `value` (`disease`) alongside `left` and `right` pointers (`GeneBstNode` or `None`).

Example: simplified image of Python tutor demo.

Example: interactive Python Tutor demo.

The `stats.py` module contains a familiar class that you may find useful ...

`StatCounter` : holds all the comparison information that you might need to check your code against. We also use it to verify that you do perform the correct number of comparisons.

> ⚠ **Important:** You cannot rely on `StatCounter` being available to you on the quiz server at the time of submission. Do not use it in your final code!

3 Tasks [100 Marks Total]

3.1 Performing diagnoses with linear probing hash tables [20 Marks]

When initialised, the `LinearProbingGeneHashTable` creates a `list` of the correct size, where each index is filled with `None`. This is the basis of the hash table, where each index is filled with `None`. All hash and comparison counters in your class are reset during initialisation. For this task complete ...

LinearProbingGeneHashTable.insert When a new disease is to be added to the hash table, first we must find where it lives by `hash`ing the gene, and making sure this hash value is in the right range for our table (*hint:* modulo). Using linear probing, we must find the next free location. You may assume the gene has not been previously inserted into the table. Finally, when a free space is found, store the disease information as a `(gene, disease)` tuple at that index. If the table is full, raise an `IndexError` with an appropriate error message[1].

LinearProbingGeneHashTable.__getitem__ To find whether a gene is associated with a disease, we have to search the hash table for that gene (again in `__getitem__`). This is very similar to inserting values into a hash table, except when probing the table, we check each entry to see if the value matches the gene we are searching for. If the gene is in the table, return the associated disease; if the gene is not in the hash table, return `None`. Do *not* add this value to the hash table.

Note: We implement the method `__getitem__` because it allows us to search for a gene with `hash_table[gene]`.

```
>>> table = LinearProbingGeneHashTable(5)
>>> table.insert(Gene('atcg'), 'Asthma')
>>> table.insert(Gene('tcga'), 'Leukemia')
>>> print(table)
LinearProbingGeneHashTable[
  0: None
  1: None
  2: (('tcga', 'Leukemia')) -> None
  3: None
  4: (('atcg', 'Asthma')) -> None
]
>>> print(table[Gene('atcg')])
Asthma
>>> print(table[Gene('cgta')])
None
```

[1] Our tests will check that you raise the correct exception, but not the message associated with it. As such, the actual message you include is entirely up to you.

3.2 Performing diagnoses with chaining hash tables [35 Marks]

When initialised, the `ChainingGeneHashTable` creates a python list of the correct size, where each index is filled with an empty `GeneLinkedList`. This is the basis of the hash table, where we will be storing all the disease information for quick lookup. All hash and comparison counters in your class are reset during initialisation. For this task complete ...

ChainingGeneHashTable.insert As before, when a new disease is to be added to the hash table through `insert(gene, disease)`, first we must find where it lives by hashing the gene, and converting the hash value to be in the right range for our table. Place a `(gene, disease)` tuple into a `GeneLink` and insert it at the start of the `GeneLinkedList` in the appropriate slot. You may assume the gene has not been previously inserted into the table.

ChainingGeneHashTable.__getitem__ To find whether a gene is associated with a disease with `__getitem__`, we have to search the hash table for a disease. We do this by hashing the value, and searching the linked list to determine if the gene we are looking for is in there. If the gene is in the hash table, return the disease; if the gene is not in the hash table, return `None`. Do *not* add this value to the hash table.

3.3 Binary Search Tree (BST) [45 Marks]

This task requires you to complete the following functions and methods in the `bst_module`. These carry out various BST operations. You should follow the specification in the function docstrings and use the provided tests to help check that your function work as intended. Please ask if you are not sure about something or you think the specification is missing something important.

Note: `bst_depth`, `bst_in_order` and `num_nodes_in_tree` are functions (rather than methods). They take a root node as their first argument. See `GeneBST.__len__` for how these are called.

GeneBST.insert Stores a node with the given key and value in the appropriate place in the tree. **(Non-recursive)**

Important: if the key exists in the tree then the value in the node with that key should be updated to the given value.

GeneBST.__getitem__ If the key exists in the tree then this function returns the value associated with the given key. Otherwise, this function returns `None`. **(Non-recursive)**

bst_depth Returns the depth of the tree starting at the given node. Be sure to read the docstring for this one. **(Recursive)**

bst_in_order Returns a list of all the (key,value) tuples from the tree, in order by key. No key comparisons should be used here! **(Recursive)**

num_nodes_in_tree Returns the number of nodes in the tree starting at the given node. **(Recursive)**

4 Testing Your Code

There are two ways to test your code for this assignment (you are required to do **both**):

- Writing your own tests – useful for debuging and testing edge cases
- Using the provided tests – a final check before you submit

The tests used on the quiz server will be mainly based on the tests in `hash_table_tests.py` and `bst_tests.py`, but a small number of secret test cases will be added. Passing the unit tests is not a guarantee that full marks will be given for that question (it is however a strong indication your code is correct; and importantly, failing the test indicates that something fundamental may be wrong). Be sure to test various edge cases such as when an input list is empty or when no number genes are common etc...

4.1 Writing your Own Tests

In addition to the provided tests, you are expected to do your own testing of your code. This should include testing the trivial cases such as empty parameters and parameters of differing sizes. You will want to start by testing with very small lists, eg, with zero, one or two items in each list. This will allow you to check your answers by hand.

4.1.1 Provided test data

Test files in the `test_data` folder are named `test_data-{k}{sorting}-{i}{sorting}-{j}-a.txt` and contain:

- i lines containing a gene (represented by 32 DNA bases) and a disease it is associated with,
- j lines containing a patient's genome,
- k lines of disease names that the patient suffers from.
- *sorting* could be g for gene (alphabetical), d for disease (random), or r for random (random),
- a is the random seed used to generate these files.

4.1.2 Provided test tools

The `utilities.py` module contains functions for reading test data from test files. The following example code should help you understand how to use the module:

Example: Reading a test data file

```
from utilities import read_test_data

filename = "./test_data/test_data-1d-2g-1-a.txt"
disease_info, patient, suffers_from = read_test_data(filename)
```

```
print(disease_info)
# [('cgtcggtttataccagaagtgaggctcagtga', 'Alpha 1-antitrypsin deficiency
    (a.a.19)')]
print(patient)
# ['catagggaattctagaatcgtacggtatatac', 'cgtcggtttataccagaagtgaggctcagtga']
print(suffers_from)
# ['Alpha 1-antitrypsin deficiency (a.a.19)']

bad_gene, disease = disease_info[0]
print(patient[1] == bad_gene)
# True
```

4.2 Provided tests

Off-line tests are available in the `hash_table_tests.py` and `bst_tests.py` modules:

- These tests are **not** very helpful for debugging—do your own smaller tests using hand workable examples when debugging!
- You should use them to check your implemented code before submission: the submission quiz *won't* fully mark your code until submissions close.
- The provided tests use the Python `unittest` framework.
  - You are not expected to know how these tests work, or to write your own `unittests`
  - You just need to know that the tests will check that your code finds the list of diseases a patient may have, and that the number of comparisons that your code makes is appropriate.

4.2.1 Running the Provided Tests

The `hash_table_tests.py` and `bst_tests.py` provide a number of tests to perform on your code:

- The `all_tests_suite()` function has a number of lines commented out:
  - The commented out tests will be skipped.
  - Uncomment these lines out as you progress through the assignment tasks to run subsequent tests.
- Running the file will cause all uncommented tests to be carried out.
- Each test has a name indicating what test data it is using, what it is testing for, and a contained class indicating which algorithm is being tested.
- In the case of a test case failing, the test case will print:

– which *assertion* failed—a value your code produced was not the expected value for that test;

– which *exception* was thrown—your code produced a Python exception when run.

> ⚠ **Important:** In addition to the provided tests you are expected to do your own testing of your code. This should include testing the trivial cases such as empty lists and lists of differing sizes.

5 Updates

Any updates will go here.

A Python Tutor Examples

Below are some simplified images of the classes you will be working with: `GeneLinkedList` and `GeneLinks`, and then `GeneBstNode`.

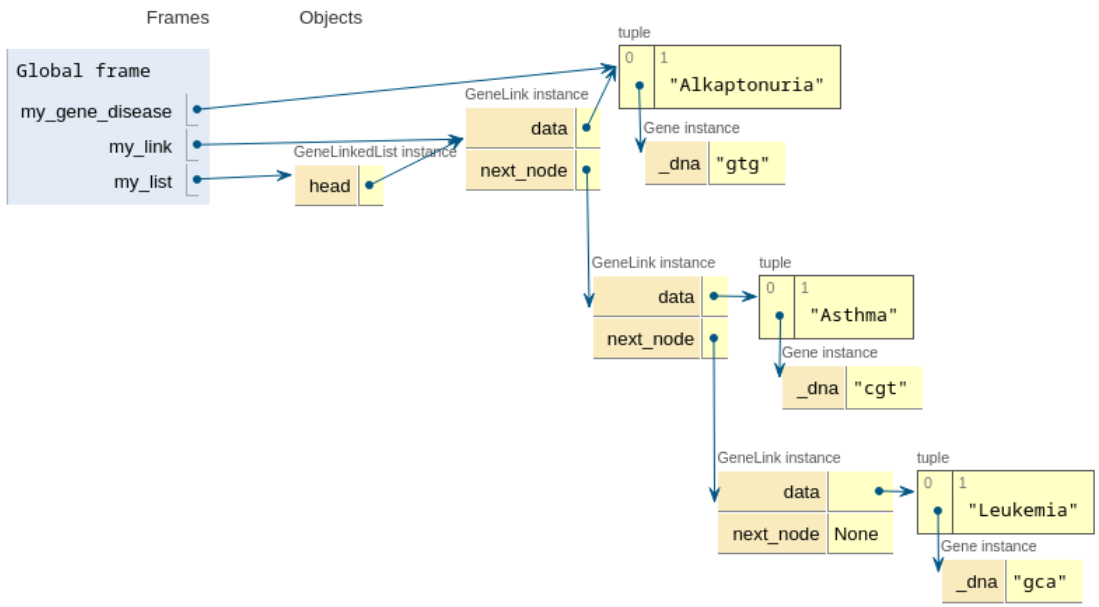

Figure 1: Example of a `GeneLinkedList` with `GeneLinks`.

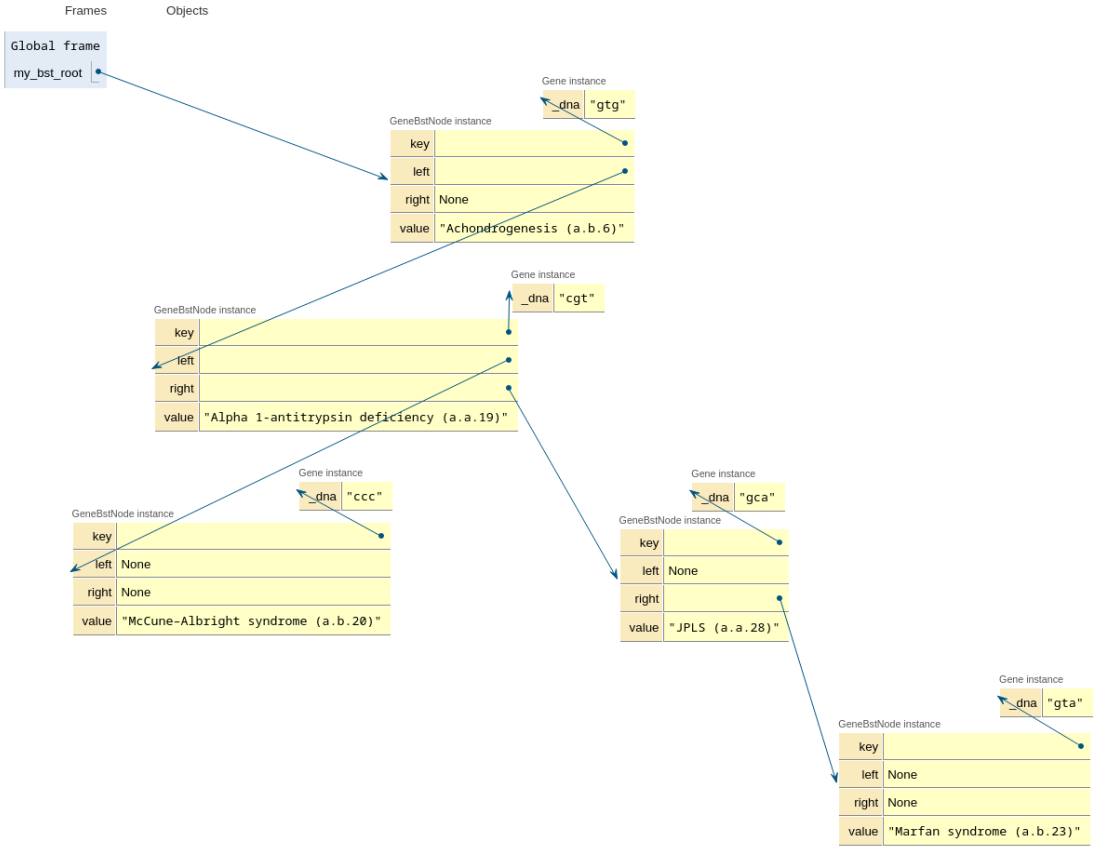

Figure 2: Example of a `GeneBstNode`.