Curtin College - Diploma of IT

Data Structures and Algorithms (DSA1002)

Trimester 2, 2021

PROJECT REPORT

Information for Use

• Introduction

The StudyHelp program is a Python implementation of a student help system that contains a network of student tutors in a university. Given the data files, the program has three starting modes: usage information, interactive testing environment, and a report mode which depend on the command line arguments given for running the program. The program provides information about the tutoring network and allows a user to read and save student data with the use of different abstract data types (ADTs) learned in the DSA1002 unit.

• Installation:

In order to run the program, these files should be included in the file submission:

- ➔ StudyHelp.py - Main file to run for StudyHelp program
- ➔ FileIO.py - File containing class for file handling
- ➔ DSALinkedList.py - Implementation of doubly linked list ADT
- ➔ DSAHashTable.py - Implementation of hash table ADT
- ➔ DSAQueue.py - Implementation of queue ADT
- ➔ DSAGraph.py - Implementation of graph ADT
- ➔ network.csv - Network data file given for DSA1002 Assignment
- ➔ units.csv - Unit data file given for DSA1002 Assignment
- ➔ students.csv - Student data file given for DSA1002 Assignment

The output files and documentation are as follows:

- ➔ README.txt - Readme file for DSA1002 Assignment
- ➔ ProjectReport.pdf - Documentation and Report for DSA1002 Assignment
- ➔ student.dat - Serialized student data file
- ➔ network.dat - Serialized network data file
- ➔ unit.dat - Serialized unit data file

This program also requires Python 3 and has numpy installed. Since it was created on WSL2 Ubuntu 20.04, some of the code written (e.g. os.system(“clear”)) may not work on other operating systems such as Windows.

File dependencies are:

- ➔ FileIO.py - imports three CSV files, linked list, hash table, and graph ADT
- ➔ DSAGraph.py - imports linked list and queue ADTs
- ➔ DSAQueue.py - imports linked list ADT
- ➔ StudyHelp.py - imports FileIO.py

• Terminology/Abbreviations
  - ➔ Abstract Data Type (ADT)
  - ➔ American National Standards Institute (ANSI) codes
  - ➔ Breadth First Search (BFS)
  - ➔ Depth First Search (DFS)
  - ➔ User Interface (UI)
  - ➔ Data Structures and Algorithms (DSA)
  - ➔ Big-O time complexity
  - ➔ Space efficiency

• Walkthrough:

The StudyHelp program has three starting options. (*Note that the program clears the terminal after every return or run of a program mode so the command line arguments can not be seen in the screenshots.*

[python3 StudyHelp.py]

- output usage information of StudyHelp program

```
STUDYHELP USAGE INFORMATION
> To enter interactive mode, add -i in the command line when running the program.
        [StudyHelp.py -i]
> To enter report mode, add -r and the filenames of the data files to be loaded when running the program.
        [StudyHelp.py -r <student_file> <unit_file> <network_file>]
```

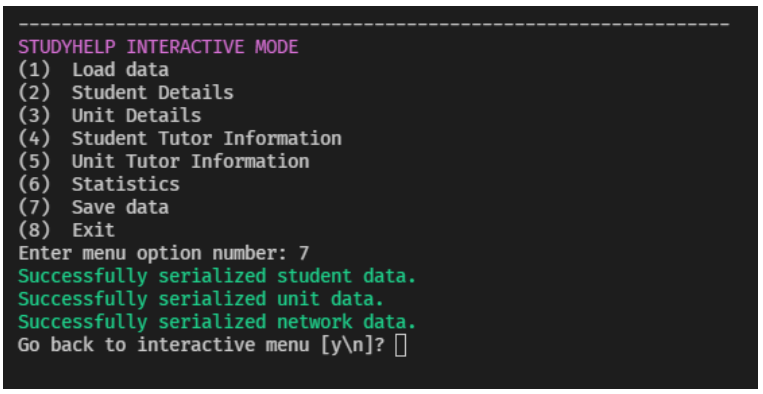
[python3 StudyHelp.py -i]

- opens StudyHelp interactive mode

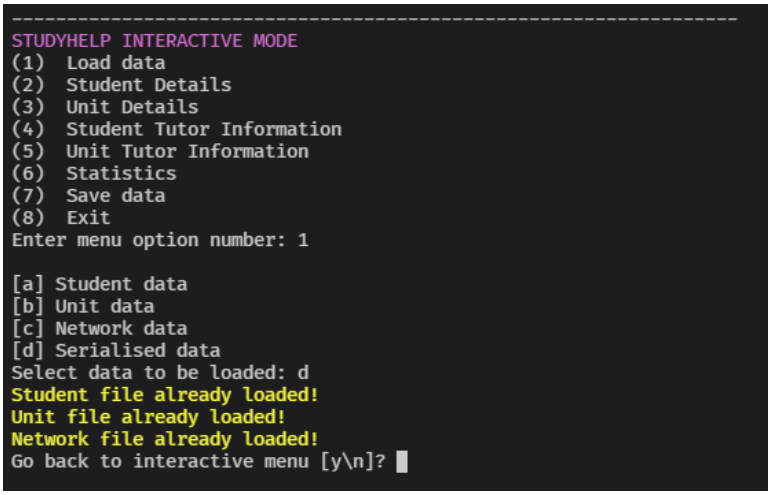
```
--------------------------------------------------------------------
STUDYHELP INTERACTIVE MODE
(1)  Load data
(2)  Student Details
(3)  Unit Details
(4)  Student Tutor Information
(5)  Unit Tutor Information
(6)  Statistics
(7)  Save data
(8)  Exit
Enter menu option number: 1

[a] Student data
[b] Unit data
[c] Network data
[d] Serialised data
Select data to be loaded: a
Enter filename containing STUDENT data: students.csv
Successfully loaded student data.
Go back to interactive menu [y\n]?
```

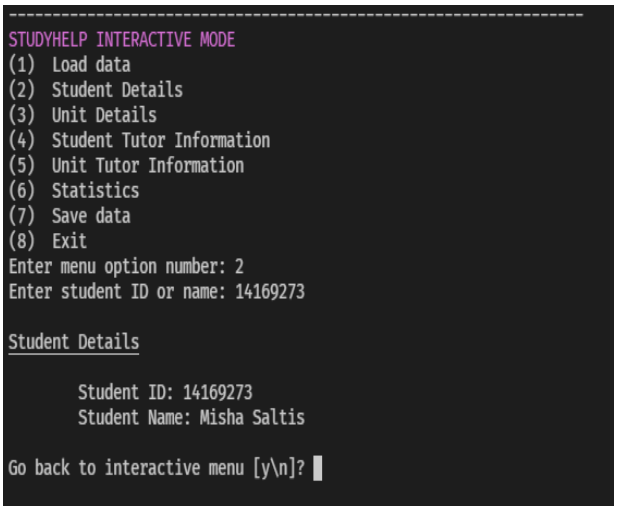
> As soon as the program starts, the user will be shown the menu and the user will input a number from 1 to 8. Preferably, in the first run of the program, all three data files should be manually loaded.

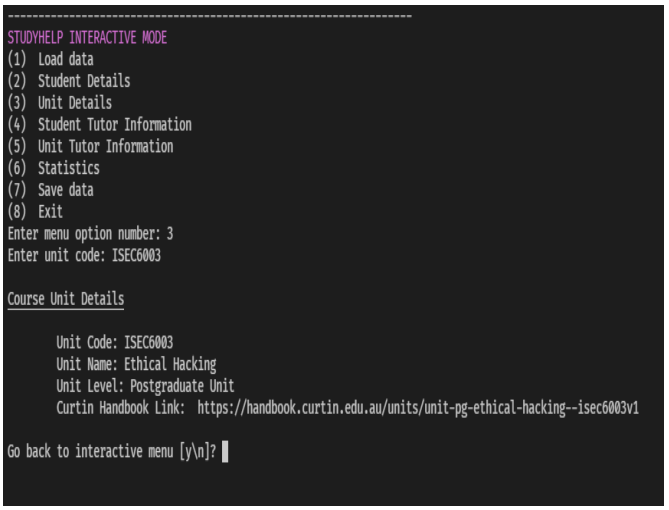


> After loading, any of the menu options can be chosen. It is also advisable to serialize the data with option 8. If so, the next run of the program can load directly from the serialized data instead of manually loading the three CSV Files.

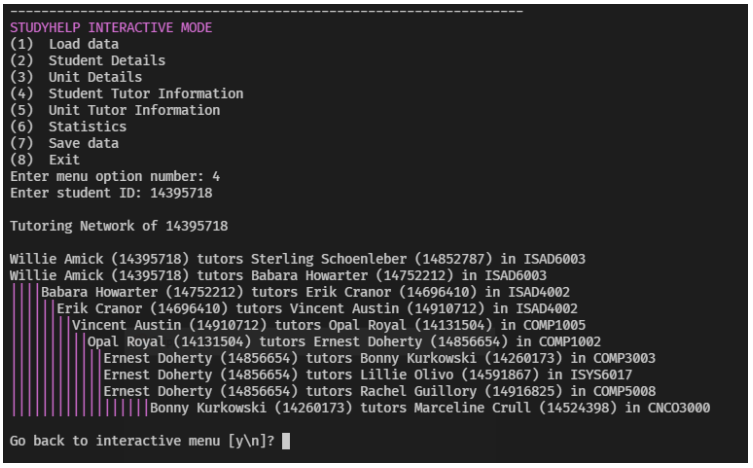


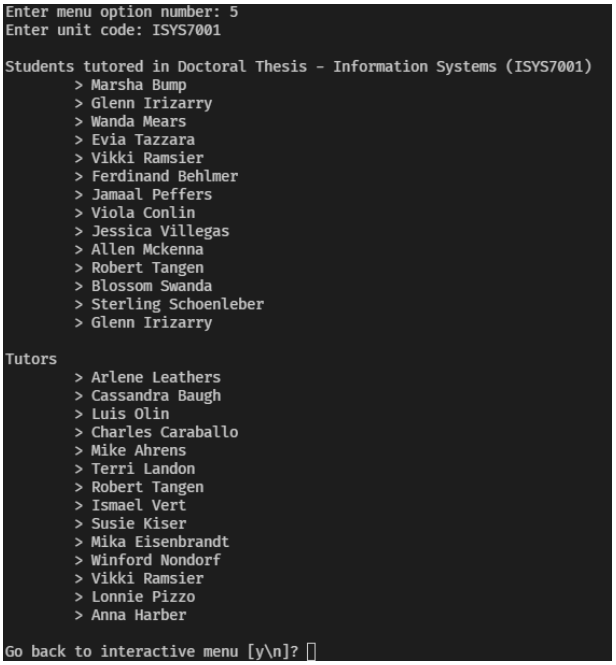
> Any duplicate actions in file handling will prompt a warning message and not proceed to load, serialize, or deserialize the file/s.





Do not enter a student ID not included in the tutoring system for option 4. I was not able to implement a condition for this.



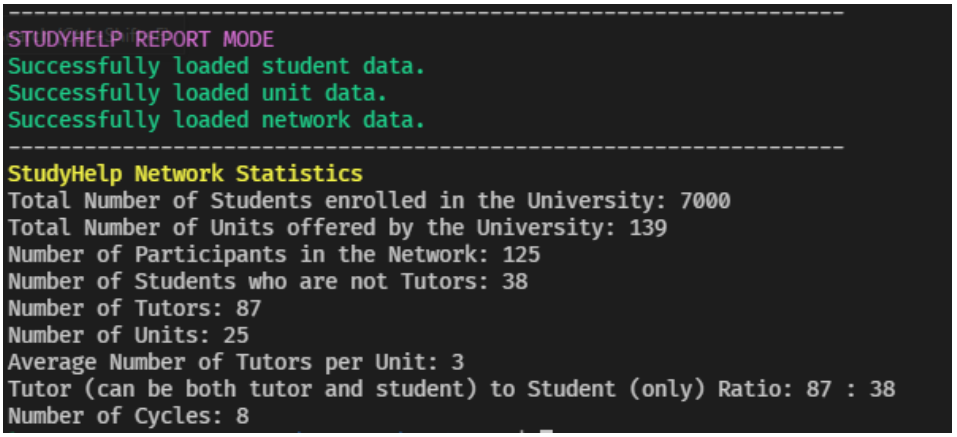


> I made option 5 output both the list of students and tutors when a unit code is entered. Please do not try to enter a unit code not in the tutoring network. I believe this has no proper handling also. Most of the error handling were focused on FileIO.

> For options 2 to 6, the user only needs to enter specified command input when prompted. If the user is done with the program, simply return to the menu and choose option 8.

[python3 StudyHelp.py -r students.csv units.csv network.csv]

- opens StudyHelp report mode



> The program will automatically load all three data files and output network statistics once it is run.
> Statistics are displayed at once instead of taking in user input for both interactive and report mode.
> Ratio of tutors (every vertex that has a linked student to it) to students only (every vertex with empty link) is used because I am not certain how to count unique items without using a list.
> Also not sure about the number of cycles.

• Future Work:

Suggested Enhancements

1. Implementation of heap for graph traversal to reduce runtime.
2. Implementation of stack in Depth First Search.
3. __getitem__ and __setitem__ special methods.
4. Restructuring of traversals (lessen for loops)
5. Store output files in an inner directory or folder

Requirements

Implementation of ADTs - modified previous practical submissions of linked list, graph, queue, and hash table. More information for the choice of ADTs can be found in the Justification section of this document.

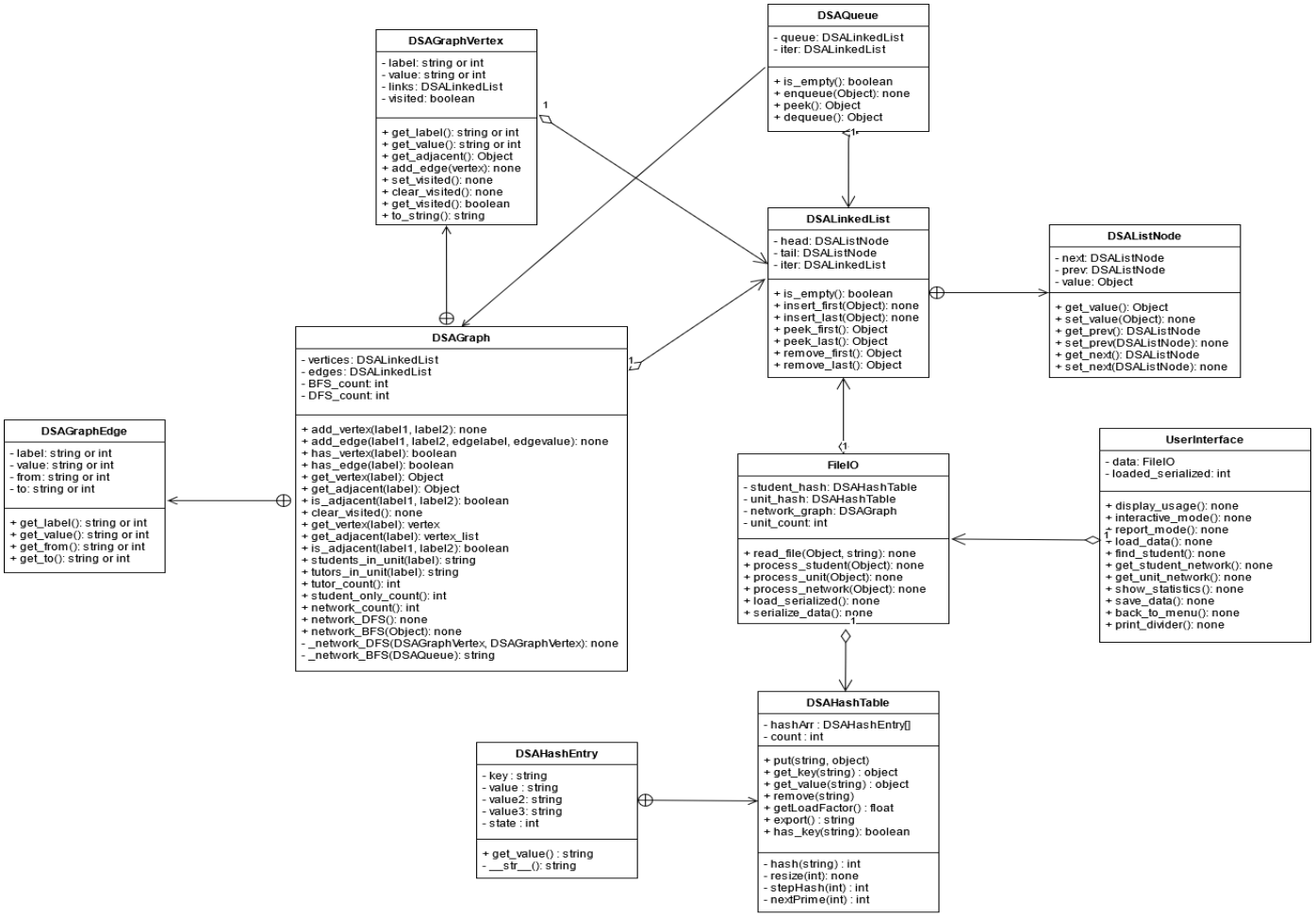
StudyHelp program - the functionalities specified were all implemented with some minor modifications/additions. There are also statistic data I am not sure about such as the number of cycles and the ratio. More information on the program can be found in the Information for Use section of this document. There are also useful information in the Class Diagram and Class Description sections for the classes created.

Python and CSV files - all .py and .csv files needed for the program are included in the zipped folder. Short descriptions of each file can be found in README.txt in this directory and also in the Information for Use section of this document.

Test Harnesses - all test harnesses of each ADT and the FileIO class can be found in the same file as their classes. Most of the the tests use the methods for the StudyHelp program with their own data especially in the DSAGraph.py where I tested out both traversals with a small data size. StudyHelp.py has a driver code in the file itself so the command line argument can read in the proper name of the program. No test harness for UserInterface class since this needs command line arguments (did not make a bash script) but most of the methods called inside the class can be tested in DSAGraph.py and others are spread out in the other files.

Documentation and Report - some comments throughout the code and this document itself. I admit that there are a lot of code that could be refactored and I may not have provided sufficient comments for most methods which are mostly from the previous practicals that I still included. Any part of the program that is not clear can be questioned during the demonstration (if there is no explanation here or as a comment).

Class Diagram



Class Descriptions

DSAGraph.py

`class _DSAGraphVertex`	“Private” class that holds student names and IDs in a linked list with a linked list of students they tutor. This class was written in order to create a link between the students and traverse the network for tutoring relationships.
`class _DSAGraphEdge`	“Private” class to store a list of edges for the student system. Since network.csv had three columns of data, I chose to store the unit code as a label of the tutoring relationship of two students.
`class DSAGraph`	Directed graph with edges as adjacency list to store vertex and edge objects. This class is useful for representing the student help system with the tutoring network that has connections between students and tutors. Also, this class contains methods for BFS and DFS of the network as a solution to a menu option in StudyHelp.

The label of a vertex is the student name while the value is the student ID. For the edges, label one is assigned as the tutor's name, label two as the student's name, and edge label as the unit code. All of these were chosen as stored values for a better representation in the output.

DSAHashTable.py

`class _DSAHashEntry`	"Private" class that holds the hashed key, values, and sets the state of a hashed index to 1 for double hashing and avoiding collisions. Modified from the practical submission code however it still does not have a __getitem__ and __setitem__ method.
`class DSAHashTable`	Class with all the methods needed for a proper hash table. New methods specific to the StudyHelp program were added. The main function of this class is to search through the entries efficiently.

DSALinkedList.py

`class _DSAListNode`	"Private" class with the list nodes as containers of data and links to other nodes which is used to store tutor, student, and unit names as well as the tutoring relationship of each member.
`class DSALinkedList`	Class for the implementation of a doubly linked list. The overall structure of the code is the same as the submitted practical in consideration of the other methods being useful for other functionalities such as removing a student from the system.

DSAQueue.py

`class DSAQueue`	Class for the implementation of queue ADT using doubly linked list from DSALinkedList.py to represent a First-In-First-Out structure. This class was created for use in the BFS of the graph network.

FileIO.py

`class FileIO`	Class that handles all file handling processes for the program such as reading, parsing, serialization, and deserialization of data. This is a seperate class to maintain proper structure and organization of the different functions of the program.

Each method in the FileIO class has the appropriate error handling for dealing with files. There are also "assertions" or warning messages to indicate whether the loading, serialization, or deserialization of data was successful or not.

StudyHelp.py

`class UserInterface`	Class is considered as the main function of the program. This is needed to accept command line arguments and run the program in the correct mode. It also includes all methods related to program output as well as taking in user input to start a process.

For the text color output, I could have created a Colors class containing the ANSI code sequences and use that for each printed message to make it more Pythonic style. However, as I tested out the outputs manually, there were already a lot of ANSI sequences for each error handling in different files and I felt it would cost me a lot of time to fix it. I did not implement any sort of class inheritance in the program since I reused the code I have written for DSA1002 practicals wherein “private” classes were created instead of nested classes or use of class inheritance.

Justification

Table Summary of Average and Worst Case Complexities of Selected ADTs

Data Structures	**Time Complexity**						**Space Complexity**
	Average Case			**Worst Case**			**Worst Case**
	Insertion	Search	Deletion	Insertion	Search	Deletion	
Graph Adjacency List	O(m)			$O(n^2)$			$O(n^2)$
Hash Table	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
Queue	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)
Stack	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)
Doubly Linked List	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(n)

GRAPH

A directed graph as adjacency list (modified from Practical 06) was implemented to store the connections of tutors to students in the network. Since it is a network, a graph as an adjacency list is an appropriate ADT for this mostly for the traversals needed to count the number of cycles (DFS) and display the indirect and direct tutoring relationships of a certain student (BFS).

For the addition of edges as labels, it does not consider the uniqueness of these labels. Due to this, the statistics such as number of units and students in the network made me use more

for loops in the program. Edges are a good representation of the tutoring connections however for that particular requirement, it does add more complexity to the code.

DOUBLY LINKED LIST

To store the vertices and edges, a linked list ADT is good for keeping the links between two labels and the list can grow and shrink with the amount of data so there is no waste of space. It's worst case space complexity is O(n) so for large amounts of data, this will take up a lot of memory.

The time complexity of accessing and searching is constant O(n) as it needs to traverse through all the nodes to find an element. Compared to other ADTs such as an array and binary search tree, linked lists have slower access times especially with the student data of 7000. With this, I only used the linked list in storing vertices and edges in the graph ADT which only have at most 153 elements. Linked lists are O(1) in insertion and deletion times and since the data does not have to be sorted, there is no need to store them in a different ADT but a heap sort can reduce the traversal time in the graph network and it uses less memory (O(1) worst case space complexity).

Because there are a lot of for loops done in different graph functions, I think a heap would be more efficient than a linked list in terms of traversals. The practicals made use of linked lists so that is what I implemented in this assignment and did not have enough time to change ADTs.

QUEUE (and Stack)

Queue is a first in, first-out (FIFO) data structure used in the BFS for getting the tutoring relationships starting from a certain student. As defined, a queue is implemented for BFS and stack for DFS but when I tried using the stack implementation from my previous submission, I was bombarded with errors that cost me half a day to figure out. In the end, I did the number of cycles manually but implemented similarly to that of the last-in-first-out structure of a stack.

HASH TABLE

A hash table was used to map keys (student ID and unit code) to given values (student name and other unit details). Hashing is often used to efficiently search through a database. The average case time complexity of searching in a hash table is O(1) so it is ideal to use particularly for the student data with 7000 items. Also, for collision, double hashing was implemented with a max step size of 5 and a next_prime method from the pseudocode in the lecture slides to ensure that there will be an available slot for any inserted element. For double hashing, a second hash function is needed so time complexity increases and the resize of the table also takes up more space than the number of elements since the table size should always be greater than this. This is to lower the load factor and the chances of collisions.

Generally, the two ADTs can be used along with the graph for the student network. I decided to use a hash table rather than a binary search tree for the following reasons:

- More concerned with searching through the data

Out of all the ADTs we learned, the hash table is the fastest in search, insertion, and deletion O(1) and the closest ADT to this is a binary search tree O(log(n)). After inserting data, most of the functionality of the program requires looking up keys or values so even if a BST is generally better for insertion, it is not optimal for the main functionality of the program.

However, a worst case scenario of retrieval was implemented in the program with the lookup of student details by student name input. I used the hashed student IDs as indexes of the elements and in order to find a student given a name, I created a for loop to traverse through the elements hence a time complexity of O(n). Also since I called the get_key method (searching) during the parsing of network.csv, an O(1) time complexity makes a big difference from O(log(n)) because it is executed for each line in the csv file. A constant time is advantageous for these situations as well as when implemented with another ADT.

- Size of input is known

A binary search tree guarantees insertion no matter the number of elements and the space needed is exactly the same as the number of elements or data. But here, the number of students and units are known so I set that as the size parameter of the table. With this, there is minimal need for rehashing thus no increase in time and space complexity.

- There is no need to sort the given data

If there was a requirement to, for example, sort the units by the number of students or tutors then a BST or any other sorting algorithm must be implemented along with the graph. For this assignment, however, the data does not need to be sorted in any way.

The commands and outputs of the StudyHelp program can be better visualized in the *Information for Use walkthrough section* while the input files are simply the three data files provided: network.csv, unit.csv, and student.csv.

References

- ADT implementations were adapted from previous DSA1002 practical submissions
- DSA concepts from Curtin University Data Structures and Algorithm lecture slides
- Additional DSA concepts from: https://www.geeksforgeeks.org/data-structures/