Data Structures and Algorithms (COMP1002) Report

- By Aaron Gangemi (Student Number: 19447337)

Table of Contents

To the marker/reader:

To the reader/marker of this assignment. This document is solely based on my Data Structures and Algorithms (COMP1002) assignment. All the associated files with this document are in the submitted file on blackboard. This document incorporates a description of all the classes in my assignment, including the purpose, why I chose to create and structure my class the way I did, certain issues and improvements for the program and finally the justification of all the abstract data types and data structures that have been used in the program.

Overview of the Program:

The program I have created is based around the election data of 2016. The program reading in the required files and then provides the user with a menu in which they select 1 of 5 options.

(1) List Nominees:
This function of the program sorts the data based on state, party, division or all and sorts the electoral data in order of the criteria. The program does this by using an array of Candidate Class objects. The user is then provided the option to filter the data so they only see certain results.

(2) Nominee Search:
This function allows the user to enter a substring representing the nominees surname. The program then searches each nominee last name for that substring and if it is a match then the result is printed to the user. The program completes this function by using a DSALinkedList of strings, each node containing information about the candidate.

(3) List by Margin:
This function of the program passes the votes for and votes against into a DSALinkedList and calculates the marginal seats for each party. This function stores the marginal seats for each division in the party. This information is then used later on in the program to construct an itinerary.

(4) Create Itinerary:
Although I was not able to correctly complete the itinerary function for the program, this component of the program is meant to take the marginal seats calculated in part 3 and gets the distances between each divisions location. This then uses a DSAGraph created by myself to construct locations and distances between locations. The algorithm prints the time, mode of transport, source and destination.

(0) Exit:
Exits the program gracefully.

Structure of the program:

My program is structured to have a class for each function and a minimum of 1 function for each data structure. These have been completed in the practicals for Data Structures and Algorithms. Each class is interlinked and may use one or more data structures to complete its required task

Referencing Note:

All Data structures used in this assignment have been referenced correctly. You will find the references for this assignment in the block comments at the start of the program. Each data structure has been self-cited and is my work from throughout the semester.

Candidate Class:

The candidate class is implemented to create a candidate object for the listing of nominees. Creating this class seemed appropriate as the listing requires the data to be filtered and ordered by many fields. By creating a class to store information about the candidate, the sorting algorithm would store an array of candidate objects. An alternative to this approach would be to create a method in the IOClass which extracts the components straight from the line and sorts the data as it is being inputted. For this implementation, there was no underlying data structure as the sole purpose was to construct an object and class.

DSAGraph:

The DSAGraph Class is used to implement the itinerary function of the program. The class has been reference to practical 6 of Data Structures and Algorithms. I have made a note of this in the block comments of my program. The graph class was a useful Data Structure for my program as its functionality gave me more control over certain aspects of the program. The most important aspect the graph provided me was to store distances and locations using the inner DSAVertex and DSAEdge classes. These were implemented for the sole purpose of storing distances and locations. Please see the Itinerary class section for the justification of this data structure.

DSAGraphList:

The DSAGraphList Class is essentially a linked list class which acts as a linked list to store both the edge list and the vertex list. The graphs implementation was originally based off of creating an adjacency list, deeming a linked list class as the appropriate implementation and data structure considering its nature to be of any length. An alternative approach reference by Robert Lafore in his book “Data Structures and Algorithms in Java” is to implement the graph using an array rather than a linked list. The core advantages of doing this is that traversing through the graph would be easier to implement, however a disadvantage is that the user would need to know how many vertices they have before the graph is constructed. Please note that this class has been referenced based from practical 4 and practical 6 of Data Structures and Algorithms. The logic has been adapted from Robert Lafore’s book, “Data Structures and Algorithm in Java.”

REFERENCE BELOW:

Lafore, R. (2003). Data structures and algorithms in Java. Indianapolis, Ind. Indianapolis, Ind. :Sams

DSALinkedList Class:

The DSALinkedList class is arguably the most used data structure in my program. The reason for this being I found the linked list more efficient than other data structures such as arrays and in addition to this, it was more efficient considering that the files being read in had

thousands of lines to extract. Although an array could do the same job, the array would require me to read in the file twice. Once to read the contents and the next to count the number of lines. Therefore, the linked list was the most adequate and suitable option. The Linked list implemented in my program is doubly linked and double ended for the sole purpose of having access to a head and tail as well as the previous node when traversing through the list.

IOClass:

The IOClass also known as the "Input/Output" class is used to read in the data from the politics file into a corresponding linked list. Reading the data into a linked list seemed adequate in my opinion due to there being roughly 70,000 lines across all files for the program. In addition, I found that traversing through a list rather than an array or binary search tree was easier to extract and depict. The main reason for this is purely efficiency of the program. The IOClass method is called at the start of the program as the specification states all data must be read in before the program menu is loaded. Although the IOClass only has one method, it has the same functionality for each file and other functions in my program sort and extract the required data. In hindsight, I could've structured this class better to incorporate to write out methods, however when I was writing the methods, I incorporated them in the main if they are being reused or their associated class.

Itinerary Class:

The itinerary class in my program is responsible for creating the itinerary at the end of the program. A note to point out is that I couldn't get the itinerary or shortest path working, so I create an itinerary which assumes you start at the first location and go to the next without calculation which politicians should go where. The Itinerary class utilizes the DSAGraph class In order to add locations and airports to the itinerary with their corresponding weighted edges. The class keeps track of the current time using a the calendar function. Although I had many choices to keep track of the time, the calendar had already kept track of the current time and made it easy to access the hours and minutes component. The class starts by accessing the linked list in which all the required data such as the airport distances and the locations are read into. These lines are read straight into the linked list. In hindsight, I could've re-structured this class to create a locations object which would construct a location for each line and contain a graph of locations in a state object, which looks more logical. Furthermore, after accessing each line and extracting the desired components, the current before and after time is updated for printing for the itinerary. From here, both the location source, location destination, weighted edge value, both before and after calendar times and the mode of transport are all passed into the graph. This data is processed and then printed by the graphs toString() method. From here, the margin list obtained is part 3 of the program is used to update the next set of information. For example, the margin list will then update the location source, location destination, state source and stated destination. Although the state would stay the same for most of the itinerary. This must be kept track of so that when the states are different, you change the current locationTo the airport. When the program validates the states being different, the program will add the airports with their corresponding information the graph. This process is then looped until there are no more margins in the margin list.

Justification of underlying Data Structure:

The underlying data structures used as mentioned in the previous paragraph is the DSAGraph. The DSALinkedList is used to store the margins and the DSAGraph is used to construct and display the itinerary. My choice for choosing the DSAGraph rather than a Binary Search Tree or array or Linked list is that given that the program must keep track of travel time and current locations. It made logical sense to construct each vertex as a location and each edge as a distance between locations. This structure is more complex than others, but with this attribute, the data structure gives me more control over the itinerary. In addition to this justification, the graph is structured to allow for both vertices and edges. A modification I had to make from the practical was to add an edge class which stores a list of edges with associated values. Implementing this for any other data structure, if deemed possible would be more difficult as a graph's implementation is designed for edges and vertices, whereas adding this characteristic would most likely require a secondary data structure which would make the program less efficient than using a single graph.

List by Margin:

The List by Margin class is responsible for the 3rd component of the program. This function was created with the sole purpose of calculating all divisions marginal seats within a desired margin. The function operates by calculating the votes for the party and the votes against the party. The margin class first reads in all the votes into a votes for list and a votes against list. Each list containing information about the state name, division name, division ID and votes for/against. This information is constructed into a string and passed into a list for further processing and data extraction. From here, the program splits each list into 2 arrays for each list. One holding the division name and the other holding the total votes for and likewise for votes against. For example, if the division name is found more than once in the linked list, it will increment the votes array and overwrite the name in the array with the same name, thus keeping track of the votes for each division. From here, 2 linked lists are constructed which act as a temporary linked list. These linked lists are passed and calculate the percentage and margin for each division. If the division meets the margin criteria, then they are both printed to screen and at the users choice, is written to file. In addition to this class, the class consists of several string methods to extract the components of the line that is constructed. The class also contains methods which obtain user input such as the custom margin.

Justification of underlying Data Structure:

The underlying data structures in this component of the assignment are linked lists and arrays. In the above discussion, the method used seems quite far-fetched and could most definitely be simplified. However, based on the specification, I identified that I required a data structure that could access uniquely identify each votes count. In hindsight, a binary search tree or a hash table would have been more appropriate as its functionality requires a key and a value. However, the problem with a tree is that a tree cannot contain the same key. Based on my implementation of practical 5, the tree would throw an exception if this was the case. Therefore I chose arrays and linked lists. The arrays index matches the position of the linked list as they will always be of the same length. Furthermore, an advantage to the array implementation unlike other data structures is that it does not

require an iterator. But rather an iterative method such as a “for” or a “while” loop. This made the overwriting of the division name and the votes incrementing easier.

List Nominees:

The List nominees Class is used widely for the first part of the program. The program is based on an insertion sort meaning that the data structure I chose was an array of Candidate Class objects. Each candidate object contains sufficient information to be sorted. As mentioned earlier, an alternative approach to this component of the program would be to read in each line into a String array and sort line by line. However, I see that approach as unstructured and will require components to be extracted and placed into a second data structure. The class starts by looping through the linked list in which stores the data about the candidates. The getArraySize() method returns a count identifying the total number of elements in the linked list. The class then sorts the components by calling Java’s “split” method which splits the line on comma, and passes each corresponding value into the mutator for each Candidate object field. After the fields are set and the object is constructed, each object is passed into a candidate class array. This Candidate class array is then sorted based on the user input. For example, if the user wants to sort by state, then the program will sort by the state field of each object in the array using the “.compareTo” method. To do this, I had to implement a case statement in the insertion sort, switching between the objects class fields. After the array is sorted at the user’s discretion, the user is then asked whether they would like to filter their data. The filter then sorts through the selected criteria and prints any matching nominees to the terminal in ordered form. The insertion sort method was chosen as it is both a stable and in-place sort. It’s time efficiency is much quicker than a bubble sort, and although a quick sort or merge sort can be argued that they are more efficient depending on the data, I believe that the insertion sort algorithm is best considering that there is a large amount of data that is raw and unstructured.

Justification of underlying Data Structure:

The underlying data structure as mentioned above is an array. The array was chosen not only due to the sorting methods being written in the practicals, but the fact that it was easy to implement understand when considering the criteria will differ. I could have just as easily implemented the sort using a linked list or binary search tree and traverse through each data structure. However, both of those data structures would have a greater time complexity than an array and both require an iterator in order to traverse through the data structure. An advantage I found with using the array is that I know the exact length and how many iteration will be required. Although reading through the linked list for the array size is quite inefficient, I believe it was the best possible option considering the data is read into a linked list and the alternative would be re-opening the file, creating another file stream and reading in the file again. The array provided the sort with simplicity and ease of access.

Main:

The predominant purpose of the main is to make method calls based on user input.
The Main function starts of by reading in all the files and data into several corresponding linked lists. From here, the main calls a menu which asks the user which function they would like to execute. The options are as specified in the specification:

(1) List Nominees
(2) Nominees Search
(3) List by Margin
(4) Itinerary
(0) Exit

The main switches between these options using a case statement. In this case statement, all the required functions to perform the specified tasks are called and further user input may be required. After each function is executed, the main asks the user whether they would like the resulting report to be printed to file. If the user then selects yes, then the main will ask for a file name to write to and then write out the users resulting data. To do this, the program stores the resulting report in a linked list, and then iterates through the list and writes out the list. The switch statement is stuck in a loop until the user enters 0 for exit. There is no underlying data structure for the main module as it makes designated calls to functions.

Nominee Search:

The Nominee Search function was created with the purpose to allow the user to search for a nominee based on the last name of the nominee. For example, if the user enters “AB” and enters “NSW” as the state, then the program should output the details for Tony Abbott. To write this algorithm, I use java’s “.startsWith” String method. After reading the lines into a linked list, I iterate through the last name of each nominee. If the “.startsWith” method returns true on the last name, then the last name starts with the string the user entered, thus printing this result to the screen. Although I have used the linked list which contains each line, a more sufficient approach to this task would have been to create nominee objects which like the candidate objects store enough information to be able to filter through. If the user enters a last name or a substring of a last name and this string does not exist, then the program will output no names and will inform the user that no matches have been found.

Justification of underlying data structure:

The underlying data structure for this task as mentioned earlier is the linked list. This was used as I decided to complete this task based on the strings read in, rather than constructing objects. In hindsight, a more efficient thing to do would be to use the Candidates array that was constructed in part 1 of the assignment. However, I chose a linked list because of the ease of expansion and the ease of being able to access the next elements toString using the linked lists iterator. Although an array would do exactly the same task just as well, I would potentially have to read the file again or iterate through the list beforehand in order to get

the array size. Therefore, in comparison to an array, the linked list seemed to be the most efficient option for the task.

Validation:

My program contains much validation. Every time the user is required to enter input that is not a string, the program will not end the submodule until the user input is a valid menu option. Although this will leave the user stuck in a loop until they can enter valid data. In addition, the program catches InputMismatchException's. This means if the user enters the wrong data type then the program will inform the user of the mistake, flush the invalid data and ask them to re-enter it. Finally, the itinerary component of the case statement in the main catches a NullPointerException. This is to ensure that if the user does not enter a margin, then they cannot construct an itinerary. Other validation can be found in the created data structures.