Data Structure and Algorithm

Assignment Report

Name: Kay Men Yap

Student ID: 19257442

DSAAssignment.java

This file is just to kickstart the program and this design of having a single method call in main is from OOPD assignment 2018 semester 1.

In the assignment, I used candidates and nominees interchangeably.

Menu.java

runMenu() is a method that keeps running the program until the user picks the exit option. Its design is from my OOPD assignment. It checks if a list by margin is generated before allowing an itinerary to be made. Each of the option's result is stored in menu so it is easier to know if an error occurred when generating the result or outputting the result and increases level of cohesion.

I did not make container classes for the nominees or the list by margin or the itinerary because it will increase the number of classes in the program and these classes wouldn't have any added functionality other than storing data.

The reason for using DSALinkedList<String[]> throughout my program is because it will be easier to sort a 2d array and give the array index to sort the 2d array by and a DSALinkedList<String[]> is similar to 2d array so conversion between both of them is straightforward and simple. This may mean a bit of extra time needed to do conversions with a time complexity of O(n) for each conversion. However, this simplifies the use of merge sort. The reason for the first 2 options to be stored as 2d arrays since merge sort with 2d arrays is easier and no actual need to convert it back to a DSALinkedList<String[]>. It is also easier to extract specific data from each line from the file and put that as an array to treat it as one entry/line from file. With all this mentioned, I did not use a 2d array to read in the data from files because that would require another method to read in the number of lines the file has and then make the 2d array and load all data into the array so this would consume more time as we are reading each files twice.

The other methods called listNominees, filterNomineeList, sortNomineeList, nomineeSearch, filterNomineeSearch, generateListByMargin, and calculateMargin are Menu.java as they are methods to perform each option as stated by their method names and don't belong to a new class that would make much meaning as these are methods specific to the assignment and may increase coupling across classes.

For listNominees, I filtered the list of nominees before sorting so that there is no unnecessary sorting taking place and increase the duration needed to sort. I did not implement a filtering system that can filter a combination of 2 filter options as this would increase my filtering code by at least 50 lines of code and make it less readable.

For nomineeSearch, I searched for nominees whose surnames start with the substring entered and then filtering the list of nominees whose surnames start with the substring as that is what is indicated as the order of operations to perform in the assignment specification from my understanding.

For generateListByMargin, margins of all divisions are calculated first before filtering out those that are not within the threshold to make it easier to read and understand what generateListByMargin is doing.

For calculateMargin, a hash table is used for calculating the margin so that it is not needed to cycle through a linked list of divisions to find the correct division and add the votes to that division since each division is unique which is what hash tables will use to store a value to a division when going through all of the data regarding number of votes. This makes the program faster as the time complexity will only be O(n) instead of O(n^2) if 3 linked list (one for raw data and one for current total votes and one for votes for the party) have been used.

Sorts.java

The reason for using merge sort instead of the other 4 sorting algorithms taught in prac 2 is because merge sort's time complexity on average is O(n log n) which is faster than insertion, selection and bubble sort that have an average case of O(n^2). Quick sort has the same average case time complexity with merge sort but merge sort is a stable algorithm whereas quick sort is an unstable sorting algorithm so merge sort will keep the original order of the data if the field to be sorted on had the same strings. Merge sort is not a in place sorting algorithm but it is not an issue with our current size of data to be sorted and insignificant memory overhead in comparison to how much memory the lab machines have (32GB of ram).

With supplying a compareIdx to mergeSort(), it makes mergeSort capable of sorting on any string of an array. Since the mergeSort was written to work with arrays from Prac 2, I used a 2d array for sorting nominees and areasToVisit instead of adapting a mergeSort that works with linked list. Since we are working with strings, String's compareTo was used in conjunction of comparing the return value with 0.

UserIO.java

All input methods in UserIO.java are from my OOPD assignment of Semester 1 2018.

I have 3 separate input commands that allows the 3 different type of inputs that will be expected from the user which is String, Integer and Real. integerInput is used a lot of times in Menu.java because requesting a integer input based on a prompt message allows me not to worry about case sensitive input if I allowed string input for menu options. There is no invalid real input when only requesting for a real number. The only invalid string not accepted for user input is an empty string. The do while loop with try catch statement is obtained from OOPD practicals and assignment and allows the program to reprompt the user when entering an invalid data type.

displayCandidates is used across option 1 and 2 to allow code reuse and less lines of code in the program. displayCandidates, displayListByMargin and displayItinerary will simply display the report generated by the option selected to the screen.

FileIO.java

Most methods in FileIo.java use the design I used in OOPD assignment for reading and writing a file.

loadCandidateList method loads all data from the HouseCandidate file with each line extracted as a String array which makes it usable for option 1 and 2 of the program. I used the HouseCandidate file separate to the First Preference Polling place files as the HouseCandidate file will contain only unique entries of candidates whereas first preference files will contain duplicated entries of candidates, so the first preference files are only used to get the number of votes for each party in each division.

loadVotesList method loads the state, division, party and votes as they are the only fields needed to generate a list by margin.

readGraphFile method simply loads all lines from both file1 and file2 to the DSALinkedList<String> lineArray. The actual line by line processing is done in the graph constructor.

saveCandidates, saveListByMargin, and saveItinerary will just save the output displayed to the screen into a file if the user wants to save the report generated to the screen.

DSALinkedList.java

This file contains the code that makes up a linked list taught in the lecture and practical is based on my Prac 4 submission and has an extra classfield called count to keep track the number of elements in the linked list and allows the conversion of linked list to an array easily.

DSAHashTable.java

This file contains the code that makes up a hash table taught in the lecture and practical and is based on my Prac 8 submisison and has an extra method on top of the methods taught in the prac called getAllKeys which allows me to get all divisions I have in the hash table to get every associating number of votes for that division.

DSAGraph.java

I have implemented a DSAGraphEdge private inner class to DSAGraph class which contains a label, a source and dest vertex, value, transport type and visited (in case I needed to know if I have been to that edge before). I experience a Stack Overflow error when running generateItinerary that calls findPath.

generateItinerary optimises the time of the itinerary as I sort the areas to visit by state which means locations near each other will be next to each other in my array of and my way to getting an itinerary involves using pairs of vertex that need to be visited starting with the first 2 elements of the linked list of vertices to visit and call findPath that should find a path from source vertex to destination vertex using a depth first search approach by checking if source and destination are adjacent and get the edge that connects to both of them and if not adjacent, go to the first edge of source and find a path from the destination of that edge to the actual destination we need to get to. After finding a path to destination, the current destination becomes the source and the next element in the linked list of vertices to visit becomes the new destination and the cycle of finding a path between pairs of source vertex and destination vertex is repeated until the path to the final element is found. Due to being busy with UCP assignment and FCS test, I am unable to make option 4 functional by the due date of the DSA Assignment.

A better way to find a path between source vertex and destination vertex would be to use Dijkstra algorithm that finds for the shortest path from source to all vertices which can be modified to from source to a specific vertex. I tried to implement Dijkstra's algorithm, but I did not know how to use Dijkstra's algorithm in conjunction with my DSAGraphEdge class and an array to keep track of what is the shortest time and what are the vertices to go through to get to the destination vertex. I understand a minheap would be useful here to know what the minimum time is to get to a vertex, but I am still not sure how I would keep track of what vertices will be needed to go through once determined the lowest time to get to that vertex from source.

Test harness for entire program:

There is a text file called DriverCode.txt that contains the input to type it and compare all files ending with "_copy.txt" to the files with similar names but without "_copy" to see if the output of the program is correct or not.