DSA – Assignment Semester 2, 2018

Student id: 19170251

Student name: Ahmad Allahham

2018

# Code Overview & Class Description

We were given the task to create a program to assist politicians with displaying polling place information and to produce an itinerary for a selected party, to travel to polling places. For this problem, I broke down the application into a number of container classes, static classes and a runnable application. The breakdown is as follows:

## **MainProgram.java**:

MainProgram.java is the main executable application that contains the main the declarations of the most fundamental ADTs used by this application, as well as the mainMenu function that displays the main menu. According to the user’s choice from the main menu, information is loaded into the appropriate Data Structures and the displayed functions are called.

## **ReportLoad.java**

ReportLoad.java is a class that contains static methods used for loading information into ADTs.

### loadNominees

The loadNominees static method loads the list of nominees into the binary search tree (declared in **MainProgram.java**). This method stores the information for each nominee in a container object called Nominee (described in **Nominee.java**). This method is also deals with the only present odd case that does not fit the information structure of the entire input file; which is the case of the politician being part of the “Shooters, Fishers and Farmers” party. Due to the comma present in the name of this party, this function joins the two separated tokens: “Shooters” and “Fishers and Farmers” into a single token and inserts this combined token in the Nominee.partyNm field (present in the **Nominee.java** class). As for the binary search tree, the Candidate’s ID was used as the key, as it is the most unique entry in the nominee’s information.

### loadVotes

The loadVotes static method allows the user to input multiple polling place vote files and stores the contents of those file in a Linked List called list. It skips over the first two lines as the first two lines of the input files is always not part of the required information.

### loadDistances

The loadDistances static method loads the distance information necessary to construct the itinerary, based on the list by margin (discussed in **ReportFunction.java**). This method imports the Linked List: list by margin called marginList. Based on the marginList, this method only loads information necessary for connecting the divisions listed in marginList to the airport and the airports to each other (this produces a relatively long and impractical itinerary, as the politician must travel to the airport between every division. Regardless, this is the current implementation).

If the information being loaded is the airport distances (option 0), the program discards all entries where the vehicle used for travel is “car” (as for every car entry between the airports, there is a similar entry that connects the two airports, but rather by plane). It also calculates the total time in minutes from the time string by converting it into minutes and hours then converting the hours into minutes and adding the converted hours with the minutes. This information is then loaded into the Linked List distanceList.

If the information being loaded is the division distances (option 1), the program only loads the entries that have the starting location being one of the divisions in marginList. As all the divisions have at least a single connection to an airport, we can guarantee a connection between all divisions by travelling from the starting location to the airport first, then to the destination (via the destination’s state airport if necessary).

## **ReportFunction.java**

ReportFunction.java is a class that contains static methods used for executing the functions displayed to the user in the method mainMenu (present in **MainProgram.java**).

### listNomiees

the listNomiees static method lists a filtered and/or ordered version of the list of nominees present in the binary search tree: tree (loaded with entries using the method: loadNominees, discussed under **ReportLoad.java**).

The method first empties the list of nominees and their relevant information in ascending order (according to their Candidate IDs) into a queue.

Then, the method asks the user for the filters and/or orders they would like to apply to the list and counts the number of filters/orders applied. Then, according to the number of filters applied, it chooses the correct filter combination from all the filter combinations possible, then if the entry matches the filters applied, the emptied entry from the queue containing the list of nominees is applied added to the filtered array and the number of entries present in that array is incremented. After the entries have been filtered and added to the filtered array, the array size is shrunk to match the number of entries it holds by exporting those entries into a smaller array that’s length is the number of entries in the original array.

Then, the program applies the combination of orders chosen by the user from the list of possible order combinations and applies a mergeSort onto the filtered array (orders are done in the following order: Surname, divisionNm, partyAb, then stateAb. This order is chosen from the least common occurrence to the most common in order to create a multi-level order). The ordered and/or filtered array is then printed to the terminal and, if chosen by the user, written to a file chosen by the user.

### nomineeSearch

The nomineeSearch static method checks if a surname substring entered by the user exists within the Nominee.surname (present in **Nominee.java**). If it does exist, then output that specific nominee’s information and continue the search for more matches. The user can apply filters in the same way as in the method listNominees (present in **ReportFunction.java**), with the added filtration of checking for the substring (as mentioned earlier) using the method *indexOf().* If indexOf() returns -1, then the substring foes not exist within the nominee’s surname and no further processing is required. If it does exist and it matches the filters chosen by the user, then the program outputs the nominee’s information and proceeds to find more surname matches. The method also writes the outputted information to a file chosen by the user, if the user choses to write to a file.

### listByMargin

the static method, listByMargin outputs to the user a list of places that for a particular party, have vote margins within the vote margin threshold entered by the user. The method asks the user for the party they would like to produce the list by margin for. Then the method steps through the Linked List called: list (loaded with information according to the method **loadVotes**, present in **ReportLoad.java**), that contains the list of divisions and the votes allocated for each party from that division.

As the method steps through this list, it calculates for every division the votes for and against the party chosen by the user. It then calculates the margin and if the margin is within the threshold chosen by the user, it outputs this result with the division name, state, votes for, votes against and the margin to the user. For every new division, the method reset the number of votes for and against and sets a flag called: *isFirstTime* to true. If this flag sis set to true, it indicates that the following entry will be the first entry for that division. This is useful as to prevent the iterator from skipping the first entry of a new division. If the iterator reaches the end of the current division (by reading information about a new division), then it calculates the margin for the finished division, outputs the relevant information for that division if its margin is within the threshold stated by the user, before updating the current state and party abbreviations.

The method also accounts for the same odd case encountered in **loadNominees** (present in **ReportLoad.java**), where the party name is “Shooters, Fishers and Farmers”. During adding the votes for a particular party, it solves this problem by adding the correct field to votes for / votes against (ie: if it encounters the odd case: adds tokens[14] instead of tokens[13]).

### itineraryByMargin

the static method, itineraryByMargin produces an itinerary based on the list by margin produced by the function **listByMargin** (present in **ReportFunction.java**). This method only runs if the mentioned method produces a correct list by margin (ie: does not abort during operation). If the list by margin was successfully constructed, the method begins by loading the distances between divisions and airports, by calling **loadDistances** (discussed in **ReportLoad.java**). It then asks the user for the day they wish to begin their journey and stores that date into a Date data structure. All trips have been programmed to commence at 9 AM and so it adds to the date constructed the equivalent of 9 hours in milliseconds (as Date stores the time in milliseconds since Jan 1, 1970).

The program then iterators over every entry in the marginList (constructed using list by margin, as discussed), storing the name of the current division and initiating a new distanceList iterator (i.e.: for every division mentioned in marginList, the method iterators over all the distances in the Linked List distanceList). As it iterates over the distanceList, it skips all entries that depart from a division other than the current division and stops when it encounters the current division. When it reaches the current division, it skips all entries that start with the current division, until it reaches the final entry (which is the distance information between the current division and a nearby airport, which happens to always be the last entry for each division).

Once it reaches that entry, the method stores the time necessary to travel from the division to the airport. It then constructs a new Date object called *futureDate*, that is the current date + the time taken to travel from the airport to the current division (in milliseconds). It then outputs the current location (which for every visit to a division has been programmed to be the airport) and date as well as the time of arrival and destination.

The method then factors in the time necessary for meeting the locals (as stated in the assignment brief, is equal to 3 hours, converted to milliseconds to be stored in the Date class) and travels back to the nearby airport, outputting the time from after the meet and greet session, the current location, the destination (airport) and the time of arrival. This process is repeated for all divisions, with factoring in a flight between airports if the division is connected to the same airport (**NOT YET IMPLEMENTED**).

## Abstract Data Structure

The following ADTs were used in the execution of the program:

* BinraySearchTree<Nominee>
* DSALinkedList<String>
* DSAQueue<Nominee>

Choices for choosing the following ADTs and potential improvements of implementation are discussed in **Justification of Major Choices** section.

## Nominee.java

Nominee.java is a continer class that contains all the relevant information regarding the nominees. Private fields are:

* StateAb: text that stores a particular nominee’s state (abbreviation)
* divisionID: Integer that stores a particular nominee’s Division ID
* divisionNm: text that stores a particular nominee’s division name
* partyAb: text that stores a particular nominee’s party (abbreviation)
* partyNm: text that stores a particular nominee’s state (abbreviation)
* candidateID: Integer that stores the particular nominee’s Candidate ID
* surname: text that stores a particular nominee’s surname
* givenNm: text that stores a particular nominee’s Given Name
* elected: char that stores whether a particular nominee was elected or not
* historicElected: char that stores whether a particular nominee was elected *historically* or not

the container class contains regular getters and setters as well as a toString method.

## MergeSort.java

MergeSort.ajva is a class containing a single public static method called *mergSort* that has been designed to order an array containing a list of Nominee Objects. The method sorts the array according to the user’s choice of order (state, party, division or surname). mergeSort a regular Merge Sort, with the only difference being the field used for comparing the values (ie: if the order chosen is orderBySurname, then the method compares the Nominee’s surname (Nominee.getSurname). etc)

# UML Diagram

# Justification of Major Choices

## Container Classes Justification

### Nominee.java

In order to save the nominee’s information, I chose to create a container class called Nominee that has the private fields mentioned in the **Code Overview** section of the report. By containing all of the nominee’s information in a single compact Object, single field manipulation was easier to do and more readable than to simply store all the necessary information in a String Array. While storing all the necessary information in a String Array would have required less classes for the operation of this program, it would have made data manipulation and particularly retrieval more cumbersome. For example, if the Binary Search Tree held a String Array instead of a Nominee Object, whenever the tree is used, the programmer must recall the particular index for a particular piece of information, for example: stringArray[9]. On the contrary, retrieving information using a Nominee class is easy and not error prone, by simply calling upon the specific getter method, for example, Nominee.getSurname().

## ADT Justification of choices

### Binary Search Tree for storing Nominee Objects

For storing the Nominee Objects (Justification discussed in section **Justification of Major Choices**), I used a Binary Search Tree as my ADT. With regards to the problem at hand, the Nominee list is exploited in two primary ways: Searching and Ordering. In light of this, I used a binary search tree for the storage of Nominee Objects due to the ability of the Binary Search Tree to order the stored items without any external storing algorithm (like a Merge Sort) and due to its quick search time relative to other ADTs.

A Binary Search Tree Is capable of three traversal methods: In-Order, Pre-Order and Post-Order traversal. Using the Binary Search Tree’s In-Order Traversal, we can visit all nodes (Objects store within the tree) in ascending order (according to the key used when creating the Binary Search Tree). This means that we can export the list of Nominees from a Binary Search Tree in an ordered manner, as opposed to other ADTs, such as a Linked List, where you would have to first exporting them into an Array and apply an external sorting algorithm before seeing having any order in your stored data. The flaw however in using a Binary Search Tree rather than a Linked List is that only a single order is possible prior to exportation. The problem at hand can potentially require a multi-level order rather than a single order, therefore, the extracted list of Nominees must be sorted externally anyway.

The other benefit of using a Binary Search Tree its ability to perform searches more quickly than a Linked List. On the average case, a Binary Search Tree can find a certain node in O(log N), while its competing ADT for this problem; the Linked List has an average search time of O(N). It is also worth mentioning that the average and worst search time for a Binary Search tree is the same O(log N). The downside though for using a Binary Search Tree for this problem is that multiple entries could contain the substring the user is searching for, which could not be done with my implementation of a Binary Search Tree, as it stops after it finds the first matching entry.

Considering all that has been mentioned, due to the required multi-level ordering and multiple searches, using a Linked List to store the Nominee Objects is a better choice than using a Binary Search Tree.

### Linked List for Storing the Distances and Creating the Itinerary

#### My Current Implementation

For storing the distances between the airports and form and to divisions, I have used a Linked List called distanceList. In my implementation (due to shortage of time), I extracted all the of the entries from the airport distances and electoral distances files that had their current location being one of the divisions to be visited (which is stored in marginList, as discussed in the **listByMargin** section of **Code Overview**). As my implementation searches through the ADT that stores the distances, storing them in this way in a Linked List cut down the search time when compared to a full list of distances. After storing all the necessary distances in distancesList, the method itineraryByMargin (discussed in **ReportFunction.java** under **Code Overview**) finds the entry that links the current location with the airport it is connected to. As all the airports are connected in some way, travelling to and from the airport ensures that we have a connection between all the different divisions.

#### Flaws in implementation

The algorithm mentioned above, as well as using a Linked List as my choice for a ADT has serious flaws. In my current implementation, the program discards all distance entries in both the airport file and electoral file except for the entries in the airport file that use a plane to travel between airports and for the entries in the electoral file that contain one of the divisions to be visited (found in marginList, as discussed in the **listByMargin** section of **Code Overview**). While without these entries it is still possible to reach every single division, the politician cannot take a direct path between divisions and must travel to the airport first before going to the next division, effectively travelling to and from the airport for every division visited. This resulted in the itinerary being more than 6 days long when using ALP as the chosen party and a 6% threshold. This is without taking into account any sleep time during the entire period and no travel time between airports (instantly teleporting between airports), which are two additional major flaws of my implementation.

#### A Better Implementation: Using a Graph as the Chosen ADT

A better approach would have been to use a Graph instead of a Linked List for storing the distances and creating the itinerary. In order to store the distance information correctly, you would have to store the locations as the vertices in the graph and use the time taken to travel between the two divisions as the weight of an edge between two vertices (therefore, your implementation of the Graph ADT must have the possibility to store weights for the edges as well as the vertices).

As for creating the itinerary, we would use a traversal method called the Breadth First Search (BFS). In order to travel from one division to the next, we search for an edge between the current vertex (the current division) and the destination. If an edge exists, then we travel down that edge and output the time taken to travel between the two divisions. However, if the destination is not adjacent to the current division, then we check to see if the vertices in the adjacency list of our current division have a connection to the destination, in order of first occurrence. If none of them have a connection to the destination, then we check each of the vertices in the adjacency lists of those vertices and so on.

This approach eliminates the compulsive need to travel to and from the airport to reach other divisions, which is a fault in the current implementation. This also takes into account all the distances in the airport and electoral distances files, opening the possibility for multiple paths to be taken to the required destination. This approach was not implemented simply due to the time constraint to produce a working version of this program.

## Merge Sort Justification

For sorting the queue exported from the Binary Search Tree, I used a Merge Sort. For this particular problem, the specifications expect your program to have the ability to do multi-level ordering (ordering by any/all of the following parameters: Surname/State/Party/Division). Since a multi-level order is expected, a simple singular order based on the Candidate ID is not sufficient. We must also choose a sorting algorithm that is stable, if we are to expect the data to remain stable after each level of sorting and preferably, a fast algorithm. The best sorting algorithm that matches the mentioned criteria is the Merge Sort. The merge sort is a stable sort that has an average runtime of O(N \* logN), which is quicker than all its competing stable sorts (such as the insertion sort, which is stable, however has a runtime of O(N^2), which is considerable slower). The only down side to using Merge Sort over say Insertion sort is its overhead, however, as the list is not large enough for us to worry about memory management, it is a minor fault.