DATA STRUCTURES & ALGORITHM ASSIGNMENT

LARA AHMAD SALEM

OCTOBER 28, 2018

STUDENT ID: 19491787
Curtin University

Contents

Overview of Code

This semester in Data structures and Algorithm we were tasked with an assignment that involved implementing Abstract data types and structures in order to create a program that minimised time complexity and increased efficiency. In my overview of code, I aim to clarify any of the decisions I made that may seem unclear to any viewer.

Why have you created so many classes that only involve case statements?

The reason some of my classes contain only switch statements was because I wanted to cater for every possible input the viewer might have chosen. Since they had to choose usually between 3 categories at the most it seemed very logical and easy to assume all the different permutations and possibilities that may arise. I wanted to create the most flexible program I could think of to give the user the ability to choose between various options. For instance in my merge sort class based on the first option, it would be considered the priority and from there on anything else would be of lesser priority.

Why Does your code Use Regex to split some Data?

Upon receiving the csv files, I had come across a line that contained a comma within the csv file and was not part of separating the data. Based on intensive research I came to a conclusion that the only possible way to split my data from a csv file was to learn a bit of Regex commands based on help from the internet from websites like Regex101 (https://regex101.com). Using this website enabled me to test out every regex command and how it would split data before I actually implemented it in my code. This definitely saved a lot of time and after countless trial and error I finally constructed a regular expression command with the help of (https://www.rexegg.com/regex-quickstart.html).

Why did you use a Boolean data type instead of just getting the user to input the file name from the beginning?

This might have been a design choice that I decided would look much more sophisticated and added a much more functionality to my program. With the menu displayed it is much easier to implement other commands. Since learning how to use abstract data types it has assisted in allowing me to create and initialise data structures like a linked list without being requiring to know the maximum capacity unlike an array. Having a Boolean statement meant that I could initialise all my data structures and not worry about any limitations. So in other words my user would not be able to access any functionality of the program without having added the file first, but they wouldn't have to worry about doing that in the beginning as the menu would first be displayed and they would be given the option to do so.

Why did use the Binary Search Tree in your margin part just to find a key and insert it but doing nothing else after?

When I was calculating the margins in my margin class I decided that when I went through the lines I would have encountered multiple divisions and so I decided that since my tree method has a find function I would use that to my benefit to ensure no two divisions have been inserted into my queue. This may seem like a tedious approach, but it really is not because Binary Search trees are guaranteed to work in O(logN) time complexity and are hence much faster than a linked list which would have to go through every single node which contrasts with the tree that splits the nodes in different subtrees. A linked list has both a time complexity in worst and average cases of O(n). In conclusion I used the tree for its time complexity to traverse much faster than a linked list.

Why do you have so many functions with the same name doing the same thing?

I decided to use method overloading in my program to enable different parametre values depending on the user input. For instance, in my Filter Nominees class based on how many categories a user would like to filter by I use a switch statement to direct the user to the appropriate function depending on their selection. Although it seems like I have duplicated data that is simply not true because I decided that I would pass in an extra parametre such as a char value in some cases to differentiate whether to extract a state or party.

Why did you call your queue count before sorting?

The reason I called my queue count was for error checking purposes. If the queue does not contain anything that it doesn't make sense to actually sort anything and continue with listing the nominees. If something does exist in the queue, then what is the point of continuing. Another reason would be the fact that I need to intialise an array to sort. So, in the case that something exists in the array I will already know how the size I need to make the array. This is much more efficient than a linked list because I would have to iterate through the list just to know how big the list is.

Why did you add your user inputs in separate functions like "Which Party", why not just ask all these questions at the same time or put them in one function?

For modularity I decided that based on how many categories a user wanted I would call that function accordingly. It made sense in a way to make that a function on its own rather than continually hard coding it. It definitely would have made my code longer and tedious. Having the ability to make it into a separate function enabled me to apply more precise error checking. For example, if a user wanted to filter by a division and they accidentally put in a String. My error checking would catch the Input Mismatch and flush that string and let the user try to input the division again.

Description of Classes

Candidate

The purpose of creating this class was because I wanted to be able to store multiple fields of data that is being read in from a file for easier access and manipulation. Creating this class has also enabled me to do more concise error checking to make sure that every element within the object being created is valid. Since I knew the exact format of the file I was able to create a class specific to the data I am reading in. By using my candidate class, I saved a lot of time in aspects such as accessing my elements. Having a container class enabled me to pass around this data with ease and Improved the organization of my code in a single place.

My container class consists of all the fields that describe a single nominee. This includes aspects such as their name, surname whether or not they were elected before and many more. All these variables are used to describe a single individual. I have implemented constructors, mutators and accessors within my container class as well as validation for strings and char values that are being imported to create the object.

Within my alternate constructor I have decided to send off my data to the mutators to check that all the input is valid. In the event that it is not then the data is disregarded and an exception is thrown to indicate that the data has failed, and the object was not created. The reason I decided to include mutators within my container class was because it is good programming practice and ensures that no junk data is being presented to the objects which would affect my program later on when it might access those fields for filtering or searching purposes.

I have also decided to keep all my variables private and hence the only way to access them would be to call the accessors. By having a container class, I can ensure that the variables are protected and are all valid. Finally having my container class adds more functionality to the objects as I can easily create a 'toString' method that prints out the data in a clear and presentable way.

FileIO

The purpose of creating my file Input and output class was because I wanted all my data handling functions that read in and write out to a file to be in one place. I created a FileIO class because I wanted users to be able to be given the functionality of just passing in a file name without having

to do all the work and inputting each data one by one. This would have been extremely tedious considering that the Australian elections data consists of thousands of candidates and would be impractical to just take in a single variable each time. Data processing using FileIO and transferring using FileIO is a much more realistic approach when you have hundreds of lines of data to process.

One of my functions creates the Location for my graph. It consists of reading each line and sending it off to a function that splits the line according to where it locates a comma. In that split function I have used a regex command to ensure that whenever within double quotes there exists a comma to ignore it and move on. For example. one line I noticed that was called the 'Shooters, Fishers and Farmers Party' and when using regex my function was able to keep that line intact and move on. Once all the data has been processed the array is returned and the data is validated to ensure that it is not null and that all the values that are required to be a double or int are successfully parsed. Once this is done the location object is created and the edges and vertex are created.

Both my create Candidate and create State do more or less the same thing however I chose to read the fourth line in one of my functions because upon inspecting the csv file I realised the header files are actually on the first line. To minimise any unwanted errors I decided to just skip those header lines and move on because I knew the format of the file would not change. After all the objects for candidate and state are created I insert them into a list to be passed around for future functions. I decided to use a list here because I did not know specifically how many lines the file was going to contain and using an array would have resulted in more lines of code and inefficiency as I would have to read the file first to get the line number for the arrays and then read it again to populate the arrays.

This FileIO class also contains a display list function for both my state and candidates. The reason I have this function was mainly for testing purposes to ensure that my list contained the exact data and that nothing was missing. It is mainly used in my test harness to prove that the above functions are working just fine. Finally, one of the core reasons for my file IO class would be the 'save to file' function that enables me to pass in an array or linked list or even queue and just

write it to a file where the user is able to go over the data presented in more detail and in their own time. In terms of creating the location I realised I could have created a new class to hold all that data but it made more sense to just quickly adapt my graph to hold the distance and transportation types.

LocationV

The reason I created a Location container class was because I wanted to encapsulate all the data related to latitude, longitude and location in a single place without having to really manipulate my graph class. Within this class I am able to provide accurate validation in terms that the latitude and longitude are accurate and are within the range of a valid point. Furthermore, having this container allowed me to pass in all this information to my graph class with ease without having to create more classifieds in my vertex class.

In my mutators I included precise validation to ensure all data being passed in is valid and accurate and if not an exception is thrown. The accessors also play a role to ensure that you are only able to access the data and not manipulate it to your own liking. Using a container class enabled me to protect my objects.

My constructors also called the mutators to validate data that is being passed in. I chose to do this because I realised that creating validation within my constructors would consist of data duplication and is just meaningless. I concluded that I could also have each class field that is being tested and validated to throw an exception. This is definitely less susceptible to just combining everything in one single try catch. My aim was to make the errors more specific.

Main

The purpose of creating this function is to tie in everything together. This is the class that initialises all the data structures and directs the users to the certain functions they want to execute. In other words, it creates a cascading effect and each class calls the other class to execute the commands. For example, if the user wanted to filter the nominees, the user input class would be called to take into account the specific number of categories to filter by and which party, state etc.… and from there the filter nominees function would be called and all the appropriate commands would be executed, and it continues.

The main class facilitates the calling of the other functions and sequentially enables the program to flow. Before the user is allowed to execute any commands, they must input a valid command and so the main menu keeps looping until a valid input is processed. However, lets also consider the fact that none of the function will be executed until a file is read In. Once the file 'read in' function is called it ensures that the file name is valid and that a number is not inputted otherwise the exception would be caught. Once everything has been read in then and only then would you be able to list nominees, search, get the margin or itinerary.

If a user was to list the nominees my function would keep looping until choice 4 is not the main menu option. For error handling purposes the choice is sent it to the switch to evaluate the choice if it is not a valid option an error message is printed, and the filter nominee function keeps looping. In the event that the user inputs a string my function will definitely catch it and flush the invalid string.

The search function does almost the same thing except that it is encapsulated with a try and 2 catches to ensure that no number is inputted when searching for a substring. I have chosen to error check in the main because I decided that it would be best to ensure the user inputs something valid than continue with invalid data and execute commands that should not work with invalid data. The list by margin function works calling different calls to acquire the threshold and to then get the margin and return the queue for future purposes in getting the shortest path and displaying it to the user.

The main reason I decided to use a main static class to call each class and tie everything together. This gives the user the power to manipulate between different classes and is more maintainable in terms of being able to test each function and classes on their own.

Margin

The sole purpose of this function is to be able to calculate the Margins for the data provided and read in. This is the class in which I use both a tree and queue to add the data. I decided to create this class because I realised that it was much more structured and consistent if I had a class for every section of my menu and occasionally calling other classes if a function I require exists in

another class. My get Margin function is not the most efficient way to go about because I realised that for every iteration in my while loop that calls the actually calculates the marginal seat it repeatedly would calculate the same division as it loops through the linked list. I chose to use a binary tree to overcome this issue because I realised that the find function does not allow any duplicate keys and in this context my divisions would not be duplicated. Furthermore, I created extra functionality in my binary tree to ensure that no duplicates were created by returning false if it does not exist. I also decided to use a queue because of its quick functionality and the ability to add in all the locations to visit for in future references when I do my itinerary.

In my calculations function I calculate a particular seat by using the division id and party name to identify whether the data I pass by in my linked list matches. Basically, if the party matches the one I am looking for then I add that as votes for and anything else in the same division that does not match is added as against. Once the margin Is calculated it is sent off to be compared. If it matches the threshold and has not been calculated before then it is added into the queue.

The reason this class was calculated was to separate the functions that are required to calculate margins from all the filtering and sorting.

Search

I created a search class because I realised it did not make sense to join unrelated functions to different classes and having everything categorized under a specific class allowed me to maintain structure and helps me manage complexity. I know that if I ever had to make a change for my searching class I would only need to go to that class. Therefore, it enabled me to pinpoint my errors quickly when I was writing my code.

My search by substring method is very simple and straight forward. I took in a single string and a linked list. All I did was iterate through the linked list and compared the string I was looking for and see if it matched the surname of anyone on the list. I decided to store my objects in a linked list because I don't need random access to any elements. In this case I want to iterate through the entire list to make sure that if a surname matches I will add it to my queue. So, I wouldn't mind going through the entire list. In other words, it is exactly the same as looping through an array but without the restriction of having to actually give a certain size to the array.

If it matches, then it is enqueued and after the linked list iterator has reached the end of the tail it stops looping. This queue is sent off to the filter function. In this function I ask the user to filter by either a party, a state, none or both state and party. This choice is taken to a switch where the option is evaluated and directed to the appropriate case. I realised that in my Filter nominees function I was also asking users to input a state and party and so I decided to call those functions to save myself the trouble of rewriting new code to do the same thing. Once the party, division or both has been chosen they are taken to a 'filterBy' function that does the job and returns the new filtered queue and prints out the data.

In my search function I have used method overloading depending on whether the user stated they wanted to filter by both state and party or just one. Both functions are the same except for the import statements. In this function the string that the user wants to filter by is placed in a while loop that iterates through the queue and adds anything that satisfies the requirements into a new queue that is returned to be printed.

I decided to create this function because the second part of the assignment requires you to search through the list of nominees. So, it seemed logical to have separate functions grouped together in a class that are responsible for searching through elements with a required string that matched the specification.

Sorts

Anything related to sorting has been placed in this class. This static class may have been duplicated a lot with many functions that seem the same but that simply isn't true. The reason I used a merge sort was because this sorting method was stable and has a time complexity of O(n log(n)). It has the best average case scenario performance (Retrieved from DSA Lecture Slides Lecture 3). I used this approach because I found its divide and conquer approach very intuitive.

The first function is related to merge sorting by every single category. So, I set the surname as a priority and sorted it last. Using merge sort definitely benefitted me because the implementation conserves the input order of my array of candidates.

The second function was created using method overloading depending on the import statements. Since three options were received I created a very lengthy if statement. I could have just had every three different options and ignored the priority of the user. I wanted to include as much functionality as possible.

Since I wanted to include as much functionality and flexibility as possible I considered every single permutation that exists and whatever option1 was I treated that as the priority and sorted it last. Finally, it is sorted and sent off to the merge function where it obtains the array and an extra parametre a number basically so that it knows by which category to sort by.

The last four functions is where the real merge sorting happens. This function operates recursively and sorts the data based on the division, surname, state, and party. To conclude I created this class because I decided the most efficient way to sort the data was through merge sort. It outweighs all the others because its stable, fast and efficient.

State

This Class is a container class that is used to get the margins and to the use that data to create an itinerary using the margins. I decided to create a container class because I found that all this data would be more useful if it could easily be accessed. The csv file provided to obtain the margins could potentially consist of thousands of lines of data. That would appear impractical to not create an object and save it into a data structure such as a linked list. In order to optimize accessing data. I encapsulated the data into a container class.

This container class contains all the class field required for future references to listing the data by marginal seats and graphing it to achieve the shortest path. In this class I have decided to make all the class fields private like I did with the previous container classes for security reasons.

Before any of the objects are created I call my mutators to validate the data and if any of the class fields fail to meet the requirements then I consider it invalid and throw an exception. The class fields in this class can only be accessed when they are called by the mutators. Without this class it would have been difficult to process, access and achieve optimization with data handling.

User Input

This class is responsible for handling the user input that is provided. I decided to create this as a separate class because I realised that a lot of the stuff that requires listing, searching or ordering requires the same data. For example, In the list nominee section and the nominee search both ask for a specific party name. Without this class it would have been tremendously difficult to validate the inputs being provided and to catch any exceptions along the way. Because I am dealing with different data types it seems almost unwise to place party and division together when asking users for an input.

This class has helped me achieve code reusability and maintain successful error checking. Without this class I would have encountered many unnecessary issues with input mismatches and would not have enabled me to handle my errors elegantly. I also found creating this class useful because it allowed me to elegantly structure my program.

The first function I created for this class would have been the 'whichMargin' function. This function is encapsulated with a try catch as well as a nested if statement that makes sure that the input is valid. If the choice inputted does not equate to 1 or 2 the default is to print an error message and assume the threshold is the default value of 6.

I also found useful to create a 'whichFile' function that decides the appropriate file reading methods. If a user wants to read in a list of nominees, then the option 1 is returned and the suitable steps to reading a list of nominees would be executed. Based on the name my direct function goes through the criteria and passes the requirements to be selectively filtered.

Without this function it would have been very difficult to maintain user input and user selections. It has definitely optimized my program and enabled flexibility between what the user wants and what my program can do.

Queue

I decided to use a queue because it helped manage my data and allowed me to obtain the objects in the order that I put them in. For filtering purposes, I decided that it seemed much more suitable to manage my candidate objects in the order they were discovered. It looked much more viable to use an array in many instances when I was filtering, searching and obtaining the margins.

Unlike using a linked list removing something from a queue only takes O(1) amount of time because it is always going to be the first element. This contrasts with a linked list as I would have had to take O(n) time to remove the candidate Object. The high-speed insertions and deletions have proven to be much more useful to me because the data that I filtered is just going to be displayed to the terminal and saved anyways so no complicated commands are going to be executed such as going through the queue or whatnot.

Although I used a linked list to implement my queue, it does not behave like a linked list. I manipulated it to act like a queue following the FIFO (First in First Out) concept. The reason I decided to implement my queue using a linked list was because I wanted it to give my queue the ability to add as many Candidate Objects as possible and not be restricted with a maximum capacity like an array implementation.

The functions in my queue are very simple and concise. My constructor for my queue just intialises the linked list and creates a count to keep track of how many values were enqueued. My isEmpty method is useful as it allows me to iterate through the queue and dequeue all the elements when necessary. My enqueue method is even simpler and consists of inserting the elements at the end of the linked list and incrementing the counter to keep track. For error checking purposes I have decided to use my isEmpty method to ensure that when someone calls the dequeue method but has not initially queued anything an exception will be thrown.

My final two methods consist of my peek method which checks the top value in the first element. As for my iterator that just returns the linked list iterator that was used to create the list and enables me to search through the list if I wanted to.

Binary Search Tree

The second data structure that I used was a binary search because I realised that it was capable of searching through the objects incredibly fast with searching and sorting. In my case I used it to keep track of my margins and used the divisions as a key. This was to ensure that none of my divisions were duplicated which would definitely complicate my itinerary for future use. Using a tree seemed like a much more viable option than to opt for a Linked list in terms of searching. With a binary tree I split the nodes from the lower half and upper half. So, when I go to look for a division I wouldn't waste time searching through divisions that are greater or smaller than the division I am looking for.

Binary trees were particularly useful in my context because I wanted to obtain the marginal seats for each division, but I did not want to get the same divisions enqueued in my queue, which why I realised it would be suitable to use a binary search tree to traverse through the data and make sure the division does not exist. Both the ability to avoid data duplication and the fast traversal search play a major role in justifying my decision to implement a tree when calculating my marginal seats.

The various methods in my tree involve methods like insert, remove and find. My insert method takes in an int key and an Object value. It uses recursion to search through the tree to find a node that is null starting with the left child else if we find a right child whose node is null we make that new node the right child. The function keeps traversing until we find an empty node.

The find method works similarly to the insert method but instead searches through the nodes and recursively traverses through the tree. Using if statements it searches through to match they keys. If the key its looking for is less than the current key it traverses left else it goes right.

Deleting the node is much more complicated and I didn't use it in my code because I had no reason to really delete any of my divisions however I did include it anyway. If the key that is to be deleted is less than the root then is lies on the left side of the tree else go right. It will recursively traverse until the node is found and deleted.

Linked List

The third data structure I implemented was a linked list to store all the objects in my file. I chose to use a linked list over an array because it is easier to store data without being limited to a specific type. The insertion of an element in a linked list is also more efficient than other data structures as I am always going to insert my Candidate and State objects at the beginning.

Throughout my entire program I have used linked lists in various ways. Some when I initially loaded the program to insert my candidate and state objects. I have also utilised it in indirect ways like my queue which is actually a linked list that behaves like a queue.

I decided to use a linked list because it saves me the trouble of having to set a maximum capacity for an array. It enables me to iterate through the linked list. I found that I didn't have to write extra functions to get the exact line number. All in all, Linked Lists have proven to actually minimise the amount of code I have to write and have enabled me to actually implement it in various ways like creating data types.

The functions in my linked list involve an insert first and insert last method. The insert first method involves inserting objects at the beginning of the list. It sets the new node as both the head and tail if the list is empty. Otherwise if an object exists in the list then the new node becomes the previous node and the new nodes next class field is set to the head. The linked lists length is also incremented.

My insert last function does the same exact thing although I did not use it in my program. However, the private class I created I had continuously used to iterate throughout my linked list when searching for substrings and filtering my nominees. This private class is known as the iterator and definitely helped me traverse through the entire linked list by returning the objects in the list while the linked list was not empty.

Another function I did use was the isEmpty function to make sure that when I tried to iterate through my linked list that I was not going through an empty list. This function was particularly important when I wanted to terminate a loop during the time I was traversing through my linked list.

Graph

Finally, I decided to use a graph in my program because the location data appeared to work well with a graph. I imagined the data for each location could be interpreted as a vertex and the edges between them as the distance retrieved from the file. This served particularly useful as I could find the shortest path easily. Using a graph enabled me to visualize my locations and the places I needed to visit.

My graph consists of three private classes which include an edge class, vertex class and a built in priority queue. The graph vertex class is responsible for storing all the vertices and edges that are adjacent to that specific vertex. I have also implemented extra class fields that are responsible for keeping track of the time and previous node. The reason I have these class fields is because the Dijkstra method requires each vertex to keep track of its previous. It also stores the LocationV object which consists of the state, latitude and longitude.

The second private class I created known as the edge class is a responsible for storing the two vertices that are adjacent to each other as well as the weight and whether they have been visited or not. Finally, the last private class that I created is actually used for finding the shortest path for each node.

This is my priority queue and is only used to store all the vertices when I am calculating the shortest path using the Dijkstra algorithm. The concept of the priority queue is simple the smallest value will always stay at the top of the queue. The reason I decided to create an additional private class was because after a rigorous amount of searching I discovered that the only way to implement the Dijkstra method is to actually use a priority queue to always get the minimum value.

Finally, after intensive research about the travelling salesman problem as well as testing out many algorithms I have to conclude that the most efficient algorithm to implement to get the shortest path would be the Dijkstra Algorithm. I have decided to use this algorithm because I found that it is the most efficient way to implement the shortest path when using a geographical map like the graph class I created. The Dijkstra algorithm that I implemented essentially uses a breadth first search and compares the nodes to get the shortest path.