|  |
| --- |
| POE PART 3 |
| **ST10219213**  **BCA3**  **PROG7312** |

11/18/2024

|  |
| --- |
| *I hereby declare that I did not plagiarise the content of this assignment and that this is my own work.*  Assignment submitted via SafeAssign:*(Tick the Box)* |

Reece wayvig

Table of Contents

[1 Data structure explanation 1](#_Toc182864256)

[Q.1.1 Data structure 1: AVL tree 1](#_Toc182864257)

[1.1.1 Overview 1](#_Toc182864258)

[1.1.2 Significance to Service requests feature 2](#_Toc182864259)

[Q.1.2 Data structure 1: Graph 4](#_Toc182864260)

[1.2.1 Overview 4](#_Toc182864261)

[1.2.2 Significance to Service requests feature 4](#_Toc182864262)

[2 Project overview 5](#_Toc182864263)

[Q.2.1 Task 1 5](#_Toc182864264)

[Q.2.2 Task 2 6](#_Toc182864265)

[Q.2.3 Task 3 7](#_Toc182864266)

[Q.2.4 Key insights 9](#_Toc182864267)

[3 Technology recommendations 10](#_Toc182864268)

[4 Implemented feedback 11](#_Toc182864269)

[5 Refferences 12](#_Toc182864270)

# Data structure explanation

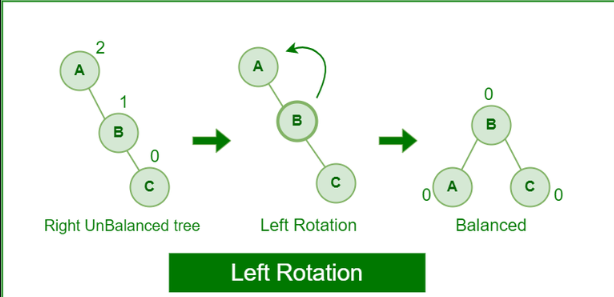
## Data structure 1: AVL tree

### Overview

According to GeeksforGeeks (2024), an AVL tree is a self-balancing Binary Search Tree, where the difference between the heights of both the left and the right subtrees of the tree cannot be more then 1. This tree’s main strength lies in its self-balancing operations.

GeeksforGeeks (2024) describe this process in detail, as the tree rebalances itself with right and left rotations. GeeksforGeeks (2024) states that right rotations are used when the imbalance occurs due to excessive height on the left subtree, and left rotations address right-heavy subtrees. Cornell University (2024) states that these imbalances can cause worst case search times for binary search trees, if not corrected properly, leading to search times of O(n). Here O describes the upper bound of an algorithm's runtime, while n simply refers to the size of the input. Cornell University (2024), defines the role of balance factors here as the difference between the height of the left and right subtrees (BF(t) = h(t.left) − h(t.right)).

Left rotation example:



Right rotation example:

A diagram of a tree

Description automatically generated

With regard to a binary search tree, this means that when the state of the tree changes, by inserting new nodes or removing them, the tree will become unbalanced. This happens (in a binary search tree) since the root node here is always constant, when adding many new nodes greater than then the root node the right side of the will become unbalanced, having more nodes then the left side of the tree. This eventually causes search times in a binary search tree to become O(n), which is the same as looping through an indexed array, something we are trying to avoid by using the AVL tree.

### Significance to Service requests feature

This brings us to the implementation of this data structure. The data structure has 2 points of interest regarding its significance in terms of efficiency. Firstly, this implementation allows us to rebalance the tree, as the number of nodes changes, and provides fast search times (guaranteed to be O(log n) ).

These two points are interesting for our scenario, as it allows us to have fast search times for specific service requests, while maintaining the searching speed when new service requests get added to the tree. This plays a significant role in data retrieval for specific records of service requests.

The search works by comparing the value you are searching for with the value of the current node. If the search value is less, the search will continue to the left subtree. If the value is greater, it will continue to the right subtree.

As an example, in my implementation of the feature I read data form csv files to populate my dummy data for the tree. I then added each new line as a service request instance. As an alternative I could have added these service requests in more bulk operations, but for the sake of simplicity I kept it to one file for all service requests. In a development environment this could easily be addressed, reading folder rather then file as more records get added and removed.

This allowed me to create a tree, that remains balances as new records are read and added to memory. As stated previously, in a production environment this would be more dynamic.

In a case where new records of service requests are added or removed while the application is running, the tree will be able to stay balanced, providing the guaranteed O(log n) search speed.

## Data structure 1: Graph

### Overview

According to GeeksforGeeks (2024), a graph data structure is a non-linear data structure the consists of vertices and edges. GeeksforGeeks (2024) state that these vertices (also called nodes) can be connected to one or more edges.

### Significance to Service requests feature

This brings us to the implementation of this data structure. The main points of interest is a graphs ability to relate different to one another. Graphs can be used to represent relationships between different nodes. This is useful in our case since this allows us to both cluster different service requests and add dependencies for service requests.

In my implementation I utilised this to create clustered data of service requests, according to both their locations and responsible departments. This allowed for simple and fast retrieval of service request records.

By organizing service requests into clusters based on their location or department, we can efficiently retrieve and manage related records. For example, service requests from the same location or department can be grouped, allowing for more organized data management and faster query execution.

Graphs also allowed me to effectively add dependant relationships between different service records. This meant that a given service request can have multiple dependencies, meaning that these service requests can only be completed when their dependencies are completed. In my implementation these relationships where pre-defined dependencies, in a more production heavy environment these relationships will be handled by some external service like an database server.

Using this data structure also allowed for some extra features to be implemented. This included a statistics feature that traversed the graph using a DFS algorithm, searching for the longest chain of dependencies. Another feature that came from this is checking whether two nodes were connected to one another. This feature also utilized a DFS algorithm that traversed the graph trying to find a connection between the source and destination nodes.

# Project overview

## Task 1

The overall objective of this section was to create a way for users to submit issue reports to the local municipality, in a way that was simple, intuitive and visually appealing. Further noteworthy requirements consisted of allowing users to upload files and adding a user engagement feature.

Addressing these points of interests, I chose to use a windows forms application for the project. It is simple to start with creating a project, low setup is required, and the interface design is simple and intuitive. My greatest challenge here was to decide on how to set up the navigation and overall feel of my user interface.

This challenge I overcame by doing ample research into different styles and designs, ultimately settling on a simple black and white theme, with a left handed pane for navigating between windows. I also needed to research methods for user engagement, opting for a reward system for users when they submit an issue. As part of lecturer feedback, I added a popup rewarding free power units for each milestone reached.

Next I needed to address the “Issues” window to allow users to submit issues, while providing for file uploads. This feature was simple to implement and mostly successful. It consisted of just adding the basic components, sufficient data validation and input response. The part of the feature that was mostly struggled with was the file upload. I misinterpreted the requirement, only allowing these files to be temporarily stored in memory (Lists). In a more production heavy environment this might have been changed to storing files on file storages online like Azure storages or Firebase.

Another small issue that was faced during this section revolves around my choice of data structures. I wanted to utilize Lists in some way, but wanted a more effective means of indexing a cluster of Issue reports. After some research I chose to implement this feature using a Multi Map. This structure is ultimately a structure that allows for multiple values per key. In my case all keys were unique, consisting of the location the report was filed in, effectively allowing for the clustering of reports according to location.

## Task 2

The overall objective of this section was to create a way for users to view local events and announcements, in a way that was intuitive and visually appealing. Further noteworthy requirements consisted of implementing a search feature, recommendations and utilizing more advances data structures.

Addressing these points of interests, I firstly needed to do more research into how I can effectively implement my events feature, to allow for a fast and optimal searching experience. During this I also took some time to create a simple UI, following the same principles and designs as the rest of the application, I can use to display any necessary information regarding events.

During this research I concluded that adding descriptive tags to my events would provide an effective means of both organising events and optimising potential user searches. Furthermore, understanding what the user is searching for would be essential for effective search algorithms, as it could prove a challenge when implementing many dropdown lists to allow for specification for user searches (dates or event names etc.), as this would harm the aesthetic appeal of my UI. Therefore I decided to implement a single search bar, tokenising user inputs (dissecting input into a list of keywords), then interpreting what a user is searching for.

This allowed me to effectively deduce keywords like dates, event names, or even event tags from their inputs. Using Regex patterns to look for specific characters was essential for deducing dates from keywords.

This bring me to my data structures. I needed to have easy access to events according to their dates and categories. I thus decided to implement them by using a sorted dictionary. This allowed me to cluster and index a series of events according to their date, allowing for quick and simple traversal through my data. As for the categories, each event had an associated hash set that included any relevant tags for the event, like festival, local, food etc. This not only allowed me to easily search for events according to event dates, but also according to tags or categories.

The next challenge I faced, was appropriately queueing events for display according to their relevancy to the user search. This was done by associating appropriate values to the results of a user search, adding more weight to items like event names, less weight to event dates (as there can be multiple events per date), and even less weight to event tags (as there are multiple tags per event). This allowed for my events to be queued for display according to how accurately it matches user inputs. This feature also provided me a potential answer for providing recommendations.

I addressed this challenge by recording user inputs in memory, storing them as keywords. I then used the same process of weighing the result of user searches, adding more weight to keywords (from user query, such as dates, names, tags) that are also included in past searches. To put more emphasis on the most recent user search, I also added extra weight to the most recent search keywords, adding the recommendations relevancy to the user.

Lastly was the challenge on how to utilize the stacks and queues. I decided to display events individually, for the pure reason of utilizing the pop() method for stacks. This essentially means that I added my list of prioritised events to a stack, using the pop() method to display the event on top. I also added a way for users to backtrack events. When a user presses next, it will display the event next in line on the stack, then add the current event to another stack. Basically, allowing users to go back to previous events.

## Task 3

The overall objective of this section was to create a way for users to Service requests, in a way that was intuitive and visually appealing. Further noteworthy requirements consisted of allowing users to track specific service requests, organising the display of requests, and utilizing more advances data structures.

Addressing these points of interests, I firstly needed to do research on the different data structures mentioned in the requirements. When I finally decided on which data structures to utilize, my first challenge was implementing a tree structure that will allow me to search for specific service request efficiently and fast. I chose to use a binary tree for this purpose, providing me fast searching times of O(log n).

My next concern was getting enough dummy data to use for testing, for this I used a CSV file with pre populated data. I was initially adding a new service request record to my tree per line read from the csv, this ended up being a problem. The CSV file that was generated had randomized service request id’s, this was a minor oversight. I switched my tree implementation to a AVL tree to compensate for the constant new inserts of records.

The next challenge was utilizing a graph to create relationships between service request nodes and using them to organize the service request displays. The requirements for this section was largely ambiguities, so I made my own interpretation of the requirements, evident in my implementation of this specification. I created three separate graphs, one for organising service requests according to locations, one for organising service requests according to responsible departments, and one for representing dependency relationships between service requests. The last of which functioned as a pending list of prerequisite requests that need to be completed before a service request can be completed. I did this by simply creating relationships between nodes of service requests. These relationships were stored in a csv file, purely for demonstration purposes, in a production environment this would change.

This all allowed me to implement the necessary requirements, along with some extra features, as I was not creative enough to think of any other implementation to utilise data structure related algorithms, like BFS/DFS for graphs (As my search was implemented with the AVL tree). I ended up implementing these algorithms as extra features, like finding the longest chain of interconnected nodes or service request in the graph, checking whether two given nodes or service requests are connected dependencies, and displaying statistics like a summery of service requests statuses along with their count (the AVL tree was used for this one).

This in large part encompasses all the relevant challenges and issues that were faced during the development of this project.

## Key insights

Most of my key learnings and insights as seen previously are attributed to research techniques and an improved understanding of data management, user search queries, and advanced algorithms.

The fist of these entailed developing a notion to delve deeper into the specifics of how specific data structures work, their use cases, and how to affectively implement associated algorithms, like traversing a binary tree.

I learned the importance of having in depth knowledge regarding the specific data structures I am working with, effectively utilizing them for their intended use, contributing to the overall efficiency and function of my application. This was essential for aligning my project with the necessary requirements and my own objectives. Objectives like providing interesting and in depth search and querying algorithms for part 2 of the project, and having sufficiently logical and resourceful implementations of data structures in part 3, to allow for interesting bonus features.

Furthermore, my understanding of data structures like binary trees and graphs has improved with this implementation. I learned of many new strategies and techniques during this project, like understanding the role of processes like recursion in algorithms such as DFS and BFS, rebalancing binary trees, creating more complex relation between nodes in graphs, and using regex patterns for search queries.

All this contributed to and supported my understanding and utilization of data structures, effectively aligning them to proper use cases. Use cases like utilizing multimaps for clustered indexed records, using stacks to display ordered lists of prioritised data, using AVL trees for fast user searches, and using graphs to show complex relationships between different nodes.

All this helped me to understand and see the intended use for these data structures. Utilising this experience, I will be able to create new software projects with a greater focus on data management and memory usage, as correctly implementing these data structures and algorithms greatly aids in achieving memory efficient and performance based software.

# Technology recommendations

Acknowledging the dependency of this project’s implementation (largely due to the stated requirements of the project), there are many improvements that can be made to make this more production worthy software.

The first of which is using some centralised form of storage. This can mean using an online database storage, or simply a file based storage on the cloud. This will aid in removing the dependency of using pre populated CSV files an inputs for the majority of data. An example of this could be to use something like Microsoft SQL Server to create and manage a database instance. This can then be hosted on the cloud.

This would effectively allow for multiple instances of this software to synchronise data across multiple users. This will also scale much more reliable, as less data will need to be stored in memory to allow the app to function.

Another recommendation can be to use external libraries or tools to help with analysing user search queries. These can be tools like AI that can analyse user search queries more accurately, allowing for more contextual data to be processes, like spelling errors for instance. This would allow for a better experience when users are searching for events in the Events and announcements for instance. One example of this can be to use the OpenAI API to analyse user inputs. This feature can also be further utilised for providing auto complete suggestion features for users, improving the overall functionality of user searches.

Another recommendation would be to use some form of user authentication. This is specifically for managing Issue submissions and service requests, as these would be more personal and allow for easier communication between users and Municipal staff. This would allow the municipality to track and manage requests and provide greater feedback to user submitted issues, through email or adding additional features to the application itself. This implementation would also allow the application to differentiate between users and staff members, allowing the application to have further functionalities implemented for managing events, issues, or service requests directly on the application.

Another tool that might be useful to the application, is integration with google map API’s for specifying locations for local events and announcements. This will allow the user to click and view the location on a map by using the google API. This feature can also be further extended to sending notifications for events that are happening in nearby locations.

Another recommendation that may improve the quality of the application is implementing Data Synchronization and Offline Support. This would allow users to interact with the app even when they don't have an active internet connection. This is especially useful when implementing the previously mentioned database integrations. This would allow for data to be synchronized when user go offline and back online again. This improves the experience for users in areas with unstable connections.

# Implemented feedback

Feedback:

* Improve and standardize UI
* Add more to part 1 reward system
* Increase font size

I addressed this feedback by adding background panels to each of my form windows. This was done to increase the contrast between the text, UI components, and background. I also made sure to use larger fonts for UI components like headings and smaller texts. This increased the readability of my data displays.

For the reward system I added pop-ups that reward the user units of power for submitting new issues. This will add to user satisfaction, rewarding users for making meaningful contributions.

# Refferences

Cornell University, 2024. AVL Tree Insertions and Balancing, [n.d.]. [Online]. Available at: <https://www.cs.cornell.edu/courses/cs2112/2015fa/lectures/lec_avl/> . [Accessed 18 November 2024].

GeeksforGeeks, 2024. Graph Data Structure and Algorithms, 9 August 2024. [Online]. Available at: <https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/> [Accessed 18 November 2024].

GeeksforGeeks, 2024. Introduction to AVL Tree, 8 August 2024. [Online]. Available at: <https://www.geeksforgeeks.org/introduction-to-avl-tree/> . [Accessed 18 November 2024].