Edward Johnson

The following is my entire planned list of enhancements for this artifact, as previously included in my updated CS499 Final project plan (Module 1 assignment).

- **Enhancement 13:** Develop an AVL tree data structure for storing and managing Client objects.
  - Sub Enhancement 1: Include the following functions, is_empty(), height(), balance_factor(), return_balance(), right_rotate(), left_rotate(), find_value(), insert_value(), remove_node().
  - Sub Enhancement 2: Use template data types to allow later reusability of the AVL tree.
- **Enhancement 14:** Implement a hash map data structure for storing employee-client associations.
- **Enhancement 15:** Create an authentication method using secure Rust crates and standards (using argon2).
- **Enhancement 16:** Implement password hashing and comparison functionality for login authorization (using argon2).
- **Enhancement 17:** Implement mechanism to handle invalid login attempts (too many / authentication state tracking).

1. Briefly describe the artifact. What is it? When was it created?

To satisfy the data structures and algorithms component of my degree capstone, I selected the final project from my CS410 reverse engineering course, as well as utilizing a data structure implementation that I wrote and used in the final project of my CS300 Data structures and algorithms course. The project was originally a rewrite of a C++ application for SNHU Financial Investing Firm. The application was designed and built to provide employees of the firm with a system that could store and track data for clients that had hired the firm, and to allow the employees to make modifications to the services that clients were being assisted with. We were initially given a small segment of broken code, and asked to fix it, and to remove any security vulnerabilities in it that we found during the disassembly / decompiling steps. I finished the rewrite of this application in C++ in June of 2024. For the CS300 component, my DSA professor, Dr. Web had asked me to challenge myself, and to try adding generics to my final project written in C++. The project focused on building an application that could import courses from a csv file, and then fill a data structure with the data parsed from those files. The values included course title, course ID, and credits. Instead of using a simple binary search tree, I wrote a self balancing tree known as an AVL tree. This AVL tree also implemented generic data objects, and was designed in a way that would allow me to use the data structure for storing data objects of any type. I completed this project in February of 2024.

2. Justify the inclusion of the artifact in your ePortfolio. Why did you select this item?

I believe strongly that the inclusion of these projects, and their contribution towards my second artifact strongly demonstrate a grasp of the core principles of data structures and algorithms. I selected this specific data structure, an AVL tree, due to it's highly desirable, dependable time complexity. I selected the project from CS410, because I believe that by enhancing it as thoroughly as I have, it again demonstrates that I have a strong understanding of relevant computer science principles, and that I can make sound determinations, and rationalizations regarding the use of specific code implementations. I strongly believe that my final projects for CS300, and for CS410 were both some of my best work in the program up to this point, and I spent a rather large amount of time on each of them. As far as specific components that show my abilities in algorithms and data structures, with the AVL tree, it would be the level of care and detail that went into the implementation of the functions that are responsible for tracking and managing the balancing operations for the tree. I remember when I was first reading about how the rotations are used to realign/return the tree to a balanced state, and it took a great deal of effort to make myself comfortable with, and to internalize the process.

What specific components of the artifact showcase your skills and abilities in software development? How was the artifact improved?

Enhancement 13: data_structs.rs

For my implementation of an AVL Tree data structure in rust, and to meet the requirements laid out in my plan for enhancement 13, I created a separate module to encapsulate the tree logic, named data_structs.rs. The entire tree is implemented using a generic type T data object, of which each node in the tree may possess a data object `<T>`. This a very valuable implementation that allows the data structure to be adapted to use any data object for which the developer desires. I initially was asked to look into using generics by my data structures and algorithms professor, Dr. Webb. He mentioned learning to use them would be a fun challenge, so I taught myself how to use and implement them in C++ when designing data structures.

For the actual methods used in the AVL tree, and to meet sub-enhancement 1 of enhancement 13, I will start off with the is_empty method. It is implemented to check to see whether or not the root of the implemented Avl tree is still None, which would mean that it has not yet had nodes added to it, or it has been cleared and is ready to have nodes added to it. The constructor function initializes a new tree with the root by default set to None. So the is_empty function returns true if it is called root = none, else returns false. For the height function requirement, this is a core accessor function that retrieves and returns the value of the node height that has been stored within the specific node being used to call the function. This height value is what allows us to perform balancing operations to maintain a subtree depth / balance between the various subtrees. The balance factor is the calculated difference between the height of the left and right subtree of a node. It is used to determine whether or not the node and subtrees are in need of rebalancing. I instead of a return_balance() function, I used find_new_balance(). the find new balance function is what determines which specific subtrees are in need of balancing operations. It implements the logic needed to recalculate the height value of a node, and then determines what balancing operations are necessary in order to keep the tree in an optimal state of balance. It is responsible for calling the left and right rotation functions depending on the subtree that is in need. This function is called following an operation that modifies the state of the tree, such as the removal of a node, or the insertion of a node. Both the right and left rotation function were implemented, in a slightly modified state due to the need to fit within the memory safe, rust themed syntax. As I would technically be modifying a data object, I would typically need to express / take ownership of that data object and state/manage it's lifetime. To avoid the need to perform these operations, I instead perform a clone operation onto the data object that I wish to move to a new position, and I am then free to assign what is essentially the same object to a new position. The find value is implemented in a very similar way as I originally wrote for my data structures and algorithms course. I use a generic key value as the search parameter, and that is converted into a client_id value within the firm_models.rs module. This allows me to use create a similar templated / generic tree as I did in my C++ operation, and allows me to adapt the tree for use in any application. For the insert value, and the remove_node, the functions are both accessed through public accessor methods, as is the find method before it. This allows me to contain access to the internal methods, and limit direct access to them to only allow function calls that come from within the same module. This is a straight forward approach towards practicing encapsulation within rust, similar to how C++ would use a public accessor method to act as an application interface to outside operations. It also allows us to fine tune error handling/propagation, and to provide initial conditions that must be met before the internal operation can be initiated. The insert_value, and remove_node functions both rely on match statements to determine the path that the program should take towards locating the correct position to insert a new node into, or to remove a node from. When these methods have finished determining where they need to perform work, and if they actually meet the conditions to perform the insertion or removal, they then call the find balance function on them-self/the new subtree to return the tree to a balanced state. The reasoning behind this practice of balancing within the tree, is so that we can maintain a known time complexity for performing operations within the tree. Because the tree is always balanced, the time complexity of operations within it remain logarithmic, at big O (log n). This essentially means for each action performed, as you are removing half of the current remaining values you must travel through / search before finding the desired value. This time complexity remains true for all operations within the balanced tree, aside from a complete traverse, which would still be O(n), as you must visit each position. Aside from operating in constant time, this is typically considered to be the second most desirable runtime complexity. It is a very efficient algorithm to use.

Enhancement 14: data_structs.rs, operation_handlers.rs (ClientHandler struct/impl)

To meet enhancement 14, I initially had planned to incorporate the hash map structure into the data_structs.rs module as this is where my other data structure was implemented. This ended up changing to my operation_handlers.rs module, due to how simplistic and easy the implementation for the hash map ended up being. I think you will likely agree that it would make more sense to contain the setup of this hashmap to the module, as this is where the actual data operations for the application are taking place. I won't go too far into detail here, as this module is one of the core implementations for my database enhancements, and it will be covered there. But needless to say, when data is used in an operation within the application, the relevant function within the employee/client handler implementations then generates a transaction. This transaction then contains both the operation to modify data in the database, as well as the data that is stored within a local data structure. While the data structure for the AVL tree is defined in the data_structs.rs module, the actual implementation where it is filled and instantiated is within the ClientHandler struct implementation. The same is done for the employee_client_pairs hashmap. Both of these data structures are passed as fields, and then initially populated when the ClientHandler is implemented. The employee ID and client ID values are first used to populate the hashmap with those ID values, and then the Client data object is inserted into the local_avltree data structure, also passing ownership of the data object into the scope of the tree.

The Client management application from my CS410 course that I wrote in C++ was reliant on a static local vector of Client objects that were managed by a singleton class. The class was responsible for providing access to the vector, and performing operations with the vector. In the rewritten, improved, rust version, the data is both stored remotely on the database, as well as locally within the AVL Tree data structure. This provides access to the data within the database, without the need to constantly perform queries on the database to retrieve it. The only time that data within the database is queried, is initially to load the local data structure, and when modifications are being made to one of the objects that is stored within it. Regarding algorithmic improvements, my original implementation in C++ was relatively simple, and relied on a singleton class instance that provided access to the Client objects within the application, as well as the vector data structure used to store them. There was a relatively low degree of complexity to the application structure. In this updated artifact, I have completely redesigned the flow of data through the application so that it is utilized in a rust-centric manner. My functions contain modular algorithms which are then contained to their respective module within the greater application. The authentication module manages all program operations that are related to the process of validating login attempts. This is primarily achieved by the main method of the main.rs module calling the login_handler of the auth.rs module. The login_handler function manages all possible outcomes for the log in process, and tracks login attempts as a means by which to end the application early. This occurs if the user has made too many unsuccessful attempts

For the data structures and algorithms component of the CS410 project, primarily the focus would be on my hashing method, and with the algorithms used to safely, and securely validate authentication attempts. In my initial C++ implementation, I manually handled the majority of the validation, and needed to implement my own steps to avoid exposing the application to timing/side-chain attacks. For this rust implementation, I was able to rely more heavily on a well established, secure, industry standard method which is implemented using the Argon2 crate. I did add a password salting function that is responsible for generating a salt that is then used by the hashing method to further obscure the hashed passwords that are stored in the database. Argon2 manages the de-encoding of the salt / configuration data from the stored hashes, and greatly simplifies the password validation steps of the authorization module.

Enhancement 17: auth.rs

The structs and functions that I used to implement my auth.rs module include all of the necessary enhancement steps as I initially defined them in my planning documents. To handle the tracking of invalid login attempts, I define the initial Authenticator struct, which contains two fields. The first field tracks the current_attempts integer value, which is incremented for each valid attempt that does not result in a successful authorization attempt. This occurs during incorrect password attempts. The second field is the integer maximum attempts allowed, which is set to 5. Both of these fields are initialized with values when they are implemented, this takes place within the constructor function of the implementation.

Enhancement 15: auth.rs

The second function within the Authenticator implementation, authenticate, fulfills the criteria planned for enhancement 15. The function has three parameters, an instance of the EmployeeHandler implementation, the employee_id that a user has entered, and the password that a user has entered. The authenticate function begins by checking the value of current attempts made against the value of max allowed attempts, and if it is equal to or greater than, the application force closes. If this condition is not met, it then calls for the incrementing of the current_attempt value to track the quantity of attempts made. Following these initial steps, the function then implements a simple match statement to use the employeeHandler instance to call the get_employee_hash function, which retrieves the stored password hash from the database. It then utilizes a Some statement with the returned, stored database hash to check if the password hash matches the password that was entered in by the user. This is accomplished by calling the verify encoded function of the argon2 crate with the arguments stored database hash, and the password entered as bytes. I am using an unwrap_or method here to handle the result of the verify encoded function call as it is the recommended approach to handling the authentication step. In this case, when the verify encoded step successfully verifies that the passwords are a match, it would return Ok() status with a true inside as the result. If the verification step fails for any reason, whether it be an incorrect password, or some other crate specific error, the result is then simplified to a false, and the return from this result with the Ok would be Ok(false). This allows us to keep the process of authentication rather simple. The only case in which the authenticate function can return a true boolean, is when there is a successful verify encoded comparison.

Enhancement 16: auth.rs

The first sections of enhancement 16 are satisfied by the third function in the auth.rs module, the hash_password function. It is only called with the initial employee_account setup stage, and I did not leave any method to access it other than the initial seeding method to generate the test employee accounts for these turn ins. It operates by first defining the configuration details for the argon2 crate using the default settings as recommended in the documentation. It uses argon2's hash_encoded functionality to take the password string, to call our generate salt function to return a proper salt integer array, and then uses the configuration details to generate the hash using the password in byte format, and the salt.

To satisfy the remaining hashing / encryption related enhancement, enhancement 16, I then implemented my login_handler function. This function retrieves / creates a new instance of the implemented employee_handler, which includes a cloned databaseManager implementation of the MySqlDatabase struct. It also creates a new instance of the authenticator implementation for use in an new series of login attempts, primarily as a means to track attempts made. It utilizes a match statement, and the returned boolean value from calling the authenticate function that was discussed in the enhancement 15 breakdown above. It uses the user entered password and the authentication to attempt validation, and on a successful login attempt, it returns the result as a true boolean.

3. Did you meet the course objectives you planned to meet with this enhancement in Module One? Do you have any updates to your outcome-coverage plans?

There were two course outcomes that I was aiming to focus on with this artifact enhancement, and I do believe that I achieved both of them.

The first of these two artifact outcomes is to, "Design and evaluate computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution while managing the trade-offs involved in design choices." I strongly believe that have completed this outcome through my usage of algorithmic principles, specifically integrating algorithmic encryption best practices, through my usage of complex data structures such as my AVL tree. Additionally, through the demonstration of my understanding of time complexity, and the use case of various designs strategies.

The second of these outcomes is, demonstrating "an ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals." This outcome I feel very similarly to the first, has been demonstrated both through my documentation of the code I am providing, through my usage of recent, up to date implementations, and my references to the relevant areas of the documentation for the language. I have generally sought to use best practices when determining my implementations, and have spent a considerable amount of time changing and tweaking various areas of my program over the past several week. I have done so until I felt confident that it provides a strong demonstration of both the quality of work that I can provide, my resourcefulness, and my ability to deliver on the values that were determined during the initial planning stages.

4. Reflect on the process of enhancing and modifying the artifact. What did you learn as you were creating it and improving it? What challenges did you face?

The majority my learning experiences with this enhancement involved continuing to build comfort and rapport with the rust language. The more I work with the language, the more I am beginning to see why so many people speak highly of it. When I initially started learning about the language, there were a number of concepts that were really difficult to wrap my head around. The whole concept of needing to define the ownership of data, or to "take ownership" of the lifetime for a specific piece is still something that I find a challenge. Having come from the background of C/C++ and Java. Rust to me almost felt like learning to program again for the first time, all over again. I have read this is a fairly common experience / opinion. I do really like the way that rust handles error propagation. Rust generally feels like it would make handling edge cases much smoother. I also really like the fact that at least with the code I have been writing so far, I haven't needed to use any manual pointers (referred to as unsafe operations) in rust. And while I still would not say that I am comfortable with lifetimes, I am certainly much more comfortable now than I was when I started working on this project. I do still find myself cloning data more than I probably should to avoid the added need to manually define lifetimes, so I think that going back and refactoring those spots in the future might be a good approach to building more comfort in this topic.

Challenges faced:

- I found myself needing to refactor various sections of my original C++ implementation of the AVL Tree to reduce areas with repeated code blocks, and to assist in cleaning up the structure of the code. One such area was where I removed and isolated sections of duplicated logic within the insert value function, and in the remove value function of the AVL Tree. I created a new function "find new balance" to handle the necessary logic, and then just called the function from both the remove_value, and insert_value functions. The new function then handles the various necessary rotations that may be needed within a subtree after an insertion or removal of a new node occurs.
- The process of handling lifetimes of the data objects within the avl tree when performing balancing operations and rotations. With C++ it was less complicated than with rust. In rust I had to come up with a way of managing and moving the nodes and their data to other positions within the tree without taking ownership of the data. There were a few ways that I attempted to do this before settling on what appeared to be the simplest approach, cloning the node data of the focus node instead of manually defining the lifetimes of it. The only function I ended up needing to define the lifetime for the data in was in the find_value function. As this function is retrieving a data object from the tree, it was necessary to define the lifetime of that data object if found. I do think that there is likely a more elegant way that I could perform these operations by taking ownership / defining lifetimes. For the time being this is a working and straight forward approach that still includes the functionality that I was looking to implement from my original AVL tree design and usage in C++. It also does so in a way that does not undermine the safety that rust provides as a language with it's unique memory management techniques. The

following is an example that I am referring to regarding cloning the data, instead of figuring out the optimal way to manage this operation using lifetimes.

```
  fn right_rotate(mut child_node: Box<Node<T>>) -> Box<Node<T>> {
    let mut parent_node = child_node.left.take().unwrap();
    let temp_node = parent_node.right.take();

    // clone data before performing movements to avoid upsetting borrow checker
    let child_data = child_node.data.clone();
    let child_right = child_node.right.take();

    child_node.left = temp_node;
    parent_node.right = Some(Box::new(Node {
      data: child_data,
      left: child_node.left.take(),
      right: child_right,
      height: child_node.height,
    }));

    // updating height
    if let Some(right) = &mut parent_node.right {
            right.height = 1 + std::cmp::max(Self::height(&right.left), Self::height(&right.right));
        }
        parent_node.height = 1 + std::cmp::max(
            Self::height(&parent_node.left),
            Self::height(&parent_node.right),
        );
        parent_node
    }
```

- I also faced a few additional challenges when figuring out how to set up the authentication process. I wasn't sure how to configure the authenticate function, and was receiving errors due to providing an outdated Argon2 dependency, and I also needed to be importing just the Config portion of the Argon2 crate. This was eventually fixed by spending additional time reading through the updated documentation provided on the documentation page for the Argon2 crate, and making sure that I was using the most recent version dependency within my cargo.toml file. This experience has taught me a about how Rust specifically handles, importing dependencies. I am not used to the need to specifically declare only the portions of an import that I am using in the file. I am used to being able to lazily use the star * to import the whole module / package. The issue with using this in rust, is that many packages / crates have their own implementations of common traits / functions. I found this out first hand when I spent well over an hour troubleshooting error messages that the compiler was throwing. I had the correct imports for the standard library functions that I was attempting to reference, but I hadn't realized that one of the imports I was using a star * to import all for, also implemented the same type, and the way in which it was implemented was not the way in which I was trying to use it. I also spent a really long time thinking that I was doing something wrong with my custom error handling, because I was getting a similar compiler error regarding a trait not properly implementing a type. To fix this in my database module at least, I ended up needing to add a "From impl" definition in my errors to allow for the conversion to my custom error types from the other specific error type implemented for the particular external / standard library traits. I now understand, and can recognize those errors by their reference to an "unexpected type found". After figuring this step out, I made sure to go through each of my modules, each of the functions that I was implementing from other crates, and to verify that I am only importing the necessary components.

Useful Documentation Links Used in Coding:

- *argon2 - Rust. (n.d.). https://docs.rs/argon2/latest/argon2/*
- *How to write documentation - The rustdoc book*. (n.d.). https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html
- *Introduction - The Rust Style Guide*. (n.d.). https://doc.rust-lang.org/nightly/style-guide/
- *Error handling - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/ch09-00-error-handling.html
- *macro_rules! - Rust By Example*. (n.d.). https://doc.rust-lang.org/rust-by-example/macros.html
- *Attributes - the rust reference*. (n.d.). https://doc.rust-lang.org/reference/attributes.html

- *Closures: Anonymous Functions that Capture Their Environment - The Rust Programming Language*. (n.d.). https://doc.rust-lang.org/nightly/book/ch13-01-closures.html
- *Advanced Functions and Closures - the Rust Programming Language*. (n.d.). https://doc.rust-lang.org/book/ch19-05-advanced-functions-and-closures.html
- *Using Box to point to data on the Heap - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/ch15-01-box.html
- *C - Derivable traits - the Rust programming language*. (n.d.). https://doc.rust-lang.org/book/appendix-03-derivable-traits.html