Assignment	3	**Due Date**	Sunday, 8 June 2025 @ 23:59
Purpose	To measure the student's ability to solve an underlying problem by implementing and/or using one or more complex data structures.		

Introduction

You are a junior developer in training, currently studying at the University of Newcastle. This semester, your team has been approached by SafeAccount Pty Ltd, a local startup that specialises in secure personal data tools for individuals and small businesses. They've built several mobile-first productivity apps and are now venturing into the cybersecurity space. As a pilot project, SafeAccount is requesting that Data Structures students design and implement a basic password manager prototype.

Figure 1: SafeAccount Pty Ltd logo

This tool should store and retrieve login credentials in a secure and organised way — a digital "keyring" of sorts. While they plan to build a fully featured mobile app in the future, they want your help in prototyping the core back end functionality using Java.

Their brief outlines a few key constraints:

- The app must store user credentials (e.g. site, username, password).
- Users should be able to update their credentials without deleting and re-adding.
- It must allow for fast lookup of credentials by site name.
- It should also support sorted display of stored credentials.
- Password strength feedback should be provided to users, when requested.

The SafeAccount team is particularly interested in seeing how well your solution uses fundamental data structures to support these features. Specifically, they've asked that you use a Binary Search Tree (BST) to support sorted credential listing, and a Hash Table to support fast lookups and updates.

They've also expressed that while this is a prototype, it's critical that your design is modular, well-tested, and clearly documented, so their development team can evaluate it for possible integration into the full app later.

💬 "We're excited to work with the next generation of software developers" said SafeAccount's CTO, Jess Murphy, herself a UoN alumna. "What we want to see is clear, reliable code using the data structures these students are learning. If they can do that, we can take it and run."

Good luck — and don't forget to choose strong passwords!

Assignment Overview

This assignment provides experience with building and using binary search trees and hash tables.

In this assignment, you will develop a simplified password manager that stores passwords in both a binary search tree and a hash table (using a linked list for chaining). The binary search tree allows for passwords to be listed and displayed in sorted order (by default, using an inorder traversal), while the hash table allows for quick access to a particular password. Your implementation is not meant to account for any security measures, such as locking the password manager or storing the passwords in a secure format (i.e., we will just store them in plain text). The system will allow users, using the command line interface, to add, remove, update, retrieve, list, and check the strength of their passwords.

This assignment is worth 100 marks and accounts for 15% of your final grade. Late submissions are subject to the rules specified in the Course Outline.

> **Important**
>
> You **are expected** to use `java.util.LinkedList` for chaining in the hash table and may also use it for building the BST iterator, as discussed in lecture. You **should not** use any other existing data structures from the Java collections framework, such as `java.util.HashMap`.

The Supplied Files

This section gives an overview of the files that you are provided as part of this assignment. There are quite a few, so you are recommended to take some time to understand the overall structure of the program.

- `App.java` – contains the main function and logic to run the password manager interface. Your code will be tested using both the `App.java` file provided, as well as with additional tests (e.g., unit tests). **This file should not be modified.**
- `Controller.java` – a class that implements the command-line interface, acting as a bridge between the input and your system, facilitating interactive testing of your implementation. **This file should not be modified.**
- `BinarySearchTreeADT.java` – an interface that defines a binary search tree. This is the basis for your `LinkedBinarySearchTree` implementation. **This file should not be modified.**
- `HashTableADT.java` – an interface that defines a hash table. This is the basis for your `ChainingHashTable` implementation. **This file should not be modified.**
- `KeyValueEntry.java` – a key-value pair object, similar to the one used in other aspects of the course. The `hashCode` method is overridden to calculate the hash of just the key (not the value) and comparison is implemented to support various operations needed by the collections. **This file should not be modified.**

- **`LinkedBinarySearchTree.java`** – this is where your implementation of the binary search tree is to be completed, as discussed in lecture. Your implementations should be recursive. **This file should be modified and be part of your submission.**
- **`ChainingHashTable.java`** – this is where your implementation of the hash table is to be completed, as discussed in lecture. Your hash table should use separate chaining, using **`java.util.LinkedList`**. **This file should be modified and be part of your submission.**
- **`Credential.java`** – a credential that will be stored in the password manager. **This file should be modified and be part of your submission.**
- **`PasswordManager.java`** – a class representing a password manager, which stores passwords in both a binary search tree and a hash table. **This file should be modified and be part of your submission.**
- **`PasswordStrengthChecker.java`** – a class that has only one behaviour, which is to check the strength of a supplied password. **This file should be modified and be part of your submission.**

Implementation

Your implementation tasks can be summarised as completing the various classes, namely **`LinkedBinarySearchTree`**, **`ChainingHashTable`**, **`Credential`**, **`PasswordManager`**, and **`PasswordStrengthChecker`**.

Running the Program

When the program is first run, you will be presented with a welcome message and a prompt. The prompt will allow you to enter commands to interact with your simulation. Initially, the program will run, but not successfully given that you will not have implemented the functionality. You are thus recommended to have a look through the provided source code to better understand the requirements for the various methods you must write.

> **Important**
>
> While the initial code you have been supplied will compile and run, it will not function correctly. Notably, all the methods you are meant to implement are either empty or return default values – you are provided only with a skeleton of the final program and are expected to complete the required implementations, including adding any necessary instance variables to the classes.

> **Hint**
>
> Your primary task is to implement the public behaviours expected of the system. However, you are free to add (private) helper methods, as necessary, to support your implementations. Particularly, you may find it useful to use helper methods when building your BST to support the recursive operations and when building your hash table.

Figure 2 shows the list of commands that the program can interpret and execute. At each step, the program will allow the user to enter a command via the prompt **`Enter command:`**. The provided command will then be executed. Each command should be entered as a whole line, with arguments separated by a space. For example, to add a **`Credential`** for site **`google.com`** with a user name of **`user123`** and a password of **`password123`**, you would use the command:
`a google.com user123 password123`

```
q
  Quit the program.
a [site] [user] [pass]
  Add a new credential with the given site (String), username (String), and password (String).
u [site] [user] [pass]
  Update the credential with the given site (String), if it exists, to the new username (String) and password (String).
r [site]
  Remove the credential with the given site (String), if it exists.
g [site]
  Get the credential with the given site (String), if it exists.
l
  List all credentials.
s
  Check the strength of all passwords.
?
  Show this menu again.
```

Figure 2: Listing of commands

> **Note**
>
> The parsing and execution of these commands is handled in **`Controller.java`**, and is not something you will need to implement. Thus, your concern is only to implement the underlying functionality to provide the intended behaviour.
>
> While **`Controller`** has only primitive error handling, which should catch most exceptions thrown during its execution, it may still crash on invalid inputs. Your program will be tested with valid commands, but your data structures and other method implementations should still include basic error checking, as appropriate.

Figure 3 provides a sample of the execution of the program, running a variety of commands.

```
Enter command: a google.com user test
Credential added.
Enter command: a google.com user newPassword
Credential already exists. Use updateCredential to change.
Enter command: u google.com user newPassword
Credential updated.
Enter command: a canvas.com user lU12$%gjdsjh@@
Credential added.
Enter command: l
[canvas.com] Username: user, Password: lU12$%gjdsjh@@
[google.com] Username: user, Password: newPassword
Enter command: s
[canvas.com] Username: user, Password: lU12$%gjdsjh@@, Password strength: Very Strong
[google.com] Username: user, Password: newPassword, Password strength: Moderate
Enter command: r canvas.com
Credential removed.
Enter command: l
[google.com] Username: user, Password: newPassword
Enter command: g google.com
[google.com] Username: user, Password: newPassword
Enter command: q
Goodbye!
```

Figure 3: Sample execution

Some additional details about each class you are expected to implement are provided below. You should also examine the documentation in the provided Java files, which will also contain information about the expected behaviour of various methods.

`LinkedBinarySearchTree`

The **`LinkedBinarySearchTree`** is an implementation of the **`BinarySearchTreeADT`** interface. The class includes a nested **`BinaryNode`** class (provided for you), which holds the data and references to the previous and next nodes. The class should be implemented as discussed in lecture, particularly using recursive operations supported by private helper methods.

The class provides various methods for inserting, removing, and retrieving data. As the **`BinarySearchTreeADT`** extends **`Iterable`**, the **`LinkedBinarySearchTree`** class must implement an iterator. This (default) iterator should return an in-order iterator. This should be accomplished through a nested **`TreeIterator`** inner class. A non-functional skeleton for the iterator class is also provided.

Youll notice that the class includes 3 iterator implementations, one for each of the pre-order, in-order, and post-order traversals. You are expected implement each of these iterators, noting that the in-order iterator will be the default. As discussed in lecture, you may implement these by populating a linked list (using **`java.util.LinkedList`**) with the nodes in the required order, then passing the iterator for the list to the **`TreeIterator`** class.

For a full list of methods required and some important details, you should examine the **`BinarySearchTreeADT.java`** file and its associated documentation.

`ChainingHashTable`

The **`ChainingHashTable`** is an implementation of the **`HashTableADT`** interface that uses a linked list for separate chaining. As mentioned previously, you should use **`java.util.LinkedList`** for the linked list. Hence, your class should include an instance variable, `private LinkedList<KeyValueEntry<K,V>>[] table`, to maintain the state of the hash table.

> **Hint**
>
> As you'll be using **`java.util.LinkedList`**, it would be beneficial to review the various methods available here: LinkedList
>
> In particular, you should recognise that the list will contain **`KeyValueEntry`** objects. This means, you may find it easiest to construct a "dummy" entry (i.e., a **`KeyValueEntry`** with the given key but no value) when operating on the list, given that the list will attempt to match elements based on their key. For example, to remove an entry with the key from a particular cell (i.e., the list at a specific index), you could use `table[index].remove(new KeyValueEntry<K, V>(key, null));`

For a full list of methods required and some important details, you should examine the **`HashTableADT.java`** file and its associated documentation.

`Credential`

The **`Credential`** class is a simple data class that consists of a site (**`String`**), a username (**`String`**), and a password (**`String`**). The class should then provide accessors for all the properties, along with mutators for the user name and password. In addition, the **`toString`** method should be overridden to provide a convenient string representation of the credential.

Specifically, the format of the string should be as below:

[site] Username: <username>, Password: <password>.

Some examples of output are given below:

[canvas.com] Username: user1, Password: password
[google.com] Username: user, Password: test

See **`Credential.java`** for a more complete listing of methods and expected behaviours.

`PasswordManager`

The **`PasswordManager`** is where the (simplified) management of credentials is handled. The class provides methods to add, update, retrieve, remove, and list stored credentials. In addition, it can loop through and evaluate the strength of all stored passwords.

The **`PasswordManager`** class should contain both a BST and a hash table – both of these should have a key type of **`String`** and an element type of **`Credential`**. As defined in the method documentations, both of these collections should store the same set of credentials, with the BST used for listing and checking passwords in sorted order, while the hash table will be used primarily for quick access to credentials given their key (i.e., to support efficient retrieval and updating). Hence, when adding or removing elements, this should be done on both collections.

Attempting to add a credential for a site that already exists should not add a new entry, or update the exist one, but should return **`false`**. Similarly, attempting to update or remove the credentials for a site that does not exist should return **`false`**. Attempting to retrieve a credential for a site that does not exist should return **`null`**. **None of these operations should throw an exception.**

See **`PasswordManager.java`** for a more complete listing of methods and expected behaviours.

`PasswordStrengthChecker`

The **`PasswordStrengthChecker`** class provides only one behaviour - to evaluate the strength of a password. This functionality is provided by the method `public static String evaluate(String password)`. As a static method, it does not require an instance of the class to execute, and should be called as `PasswordStrengthChecker.evaluate(password);`

The strength check is done in a relatively rudimentary way, which is assigning a score from 0 to 5 for a password based on the following criteria. Specifically, satisfying each of these criteria adds 1 point to the score, the sum of which determines the strength.

- A minimum of 8 characters.
- One lower case letter.
- One upper case letter.
- One digit.
- One special character (defined as a character that is not a letter or digit).

The strength of the password will then be determined by the score associated, based on the criteria above, according to:

$$\text{strength} := \begin{cases} \text{Very Weak, if score} = \{0, 1\} \\ \text{Weak, if score} = 2 \\ \text{Moderate, if score} = 3 \\ \text{Strong, if score} = 4 \\ \text{Very Strong, if score} = 5 \end{cases}$$

As a few examples:

- The password **`test`** would have a score of 0 (Very Weak) as it only meets one of the criterion (lower case letter).
- The password **`Test123`** would have a score of 3 (Moderate) as it meets the criteria of one lower case letter, one upper case letter, and one digit.
- The password **`@9a&F7b!fu1g`** would have a score of 5 (Very Strong) as it meets all five of the criteria.

> **Hint**
>
> Consider the various methods in the **`Character`** class here: Character. In particular, **`isDigit`**, **`isLowerCase`**, **`isUpperCase`**, and **`isLetterOrDigit`** (which, of course, you want the negation of). Each of these methods will accept a character and return a boolean indicating if the character is of that type. Naturally, what you want to do is determine if each of these criteria are true for any character - so consider a loop across all characters in a string. Remember, you can use the method **`charAt`** on a string to return the character at a particular index. Also remember, you only want to count each of the criteria once. That is, having a special character only adds one to the score, not one point for each special character.

See **`PasswordStrengthChecker.java`** for a more complete listing of methods and expected behaviours.

Marking

Your implementation will be assessed on both correctness and quality. This means, in addition to providing a correct solution, you are expected to provide readable code with appropriate commenting, formatting, and best practices.

> **Important**
>
> Code that fails to compile is likely to result in a zero for the functionality section.

At the discretion of the marker, minor errors may be corrected (and penalised) but there is no obligation, nor should there be any expectation, that this will occur. Your code will be tested on functionality not explicitly shown in the supplied main file (i.e., using a different **`App.java`** file). Similarly, you can expect that unit testing will be conducted on your code to ensure it meets the specifications. Hence, you are encouraged to test broader functionality of your program before submission.

Marking criteria will accompany this assignment specification and will provide an indicative guideline on how you will be evaluated.

> **Important**
>
> The marking criteria is subject to change, as necessary.

Submission

Your submission should be made using the Assignment 3 link in the Assignments section of the course Canvas site. Assignment submissions will not be accepted via email. Incorrectly submitted assignments may be penalised.

Your submission should include only the completed versions of the core classes, namely: **`ChainingHashTable.java`**, **`LinkedBinarySearchTree.java`**, **`Credential.java`**, **`PasswordManager.java`**,

and **`PasswordStrengthChecker.java`**. You do not need to include any other files in your submission, such as those provided to you. Be sure that your code works with the supplied files. Do not change the files we have supplied as your submission will be expected to work with these files – when marking your code, we will add the required files to your code. As described above, we will also assess your submitted code on additional test cases to test its correctness.

Compress the required files into a single **`.zip`** file, using your student number as the archive name. For example, if your student number is c9876543, you would name your submission **`c9876543.zip`**. Do not use **`.rar`**, **`.7z`**, **`.gz`**, or any other compressed format as these will be rejected by Canvas. As above, you should not include the entire VS Code project in your submission – only include the necessary java files.

> **Note**
>
> If you submit multiple versions, Canvas will likely append your submission name with a version number (e.g., **`c9876543-1.zip`**) – this is not a concern, and you will not lose marks.

Remember that your code will conform to best practices, as discussed in lecture, and should compile and run correctly using the supplied files in a standard Java environment. There should be no crashes in the execution. Please see the accompanying marking criteria for an indicative (but not final) guideline to how you will be evaluated.

Helpful Tips

1. Read the supplied files carefully, particularly any interfaces you will be expected to implement – they may provide further information on the specification for various methods.
2. Work incrementally – don't try to implement everything at once. Consider getting a bare-bones version to compile, which will enable you to test methods as you implement them.
3. Remember that you can debug your program in VS Code. See Lab 4 for a brief guide on using the debugger.
4. Write some unit tests as you go. At various points, rerun all the tests to ensure your solution works correctly and, importantly, that previous test successes are still successful.
5. Start early! The longer you wait to start, the less time you will have to complete the assignment.