SENG1120 – Data Structures
Semester 2, 2025

Assignment	2	**Due Date**	Sunday, 26 Oct 2025 @ 23:59
Purpose	To measure the student's ability to solve an underlying problem by implementing, using, and/or reporting on one or more complex data structures.		

Introduction

TreasureQuest Interactive is a game development studio headquartered somewhere between the fourth floor and the coffee machine of a Newcastle office block. They've recently announced their newest (and most ambitious) title: *Treasure Hunt: Legends of the Forgotten Isles*. It promises everything – mystical guardians, cursed treasure, dangerous caves, and a surprisingly large number of exploding chickens (pending approval by the QA team). There's only one problem: the systems that should make all this fun actually work … don't exist yet.

💬 "*We've got the dragons. We've got the loot. We even have a sound effect that goes 'KA-CHING!' when you pick up coins. But when it comes to actually storing those coins, sorting through a backpack, or fighting the guardians? We're dead in the water. Turns out you can't just duct-tape data structures together.*" said CTO Amelia Grant.

The developers at TreasureQuest have admitted that while they can draw shiny treasure chests, they have no idea how to actually make those chests hold things in a logical, efficient way. Guardians can roar, but apparently not keep track of where they're standing. The player can swing a sword, but the game has no way of remembering if they're holding a sword, a potato, or three hundred identical keys. In short: the gameplay backbone is missing.

And so, in a moment of inspired genius, TreasureQuest has turned to you – the brilliant students of SENG1120 – to bring *Treasure Hunt: Legends of the Forgotten Isles* to reality.

Your mission, should you choose to accept it (spoiler: you don't have a choice), is to implement the core backend systems that will bring this world to life:

- A hash table (with linked list chaining), to serve as the "world map engine", efficiently storing and retrieving locations, their guardians, and their treasure chests.
- A binary search tree (BST) to manage the player's inventory. Sorting, selling, and keeping track of items suddenly becomes possible.
- A text-based playable demo, where a brave soul (that's you, or rather your **`Player`** class) can explore caves, fight guardians, collect loot, sell items, drink potions, and generally try to stay alive. You'll need to implement a few supporting classes to get this running.
- JUnit 5 unit tests, because if the player's sword doesn't swing or a guardian forgets where they're standing, someone is going to leave a very angry Steam review.
- A short technical report, where you'll show off UML diagrams, exemplify your testing efforts, and reflect on how software development practices kept your code cleaner than a freshly polished cutlass.

The TreasureQuest team is eagerly awaiting your prototypes. They don't just want functioning code – they want proof that you understand the artistry (and necessity) of data structures.

Assignment Overview

This assignment provides experience with using binary search trees and hash tables. It also reinforces the programming concepts of polymorphism, generics, and both array and linked variants of lists.

In this assignment, you will develop classes that support a simplified, text-based adventure game in which a player explores a world of interconnected locations, encounters guardians, and collects treasures stored in chests. The world map is managed using a chaining hash table, while the player's inventory is supported by a binary search tree for efficient storage, retrieval, and ordering of items. You are required to implement these core data structures (among a few other supporting classes), integrate them into the gameplay loop, and demonstrate their functionality through unit testing and a technical report.

This assignment is worth 100 marks and accounts for 20% of your final grade. Late submissions are subject to the rules specified in the Course Outline.

Important

You may not use hash table or tree-based collections from the Java standard library, which includes **`java.util.HashMap`**, **`java.util.HashTable`**, among others. Use of disallowed concepts will be penalised, as per the accompanying marking criteria.

Do note, however, that you are expected to use **`java.util.List`** and **`java.util.ArrayList`** as part of the **`TreasureChest`** class. Similarly, use of **`java.util.LinkedList`** as part of your BST iterator, as discussed in lectures, is expected. These use cases are expected and will not attract any penalty.

Please reach out to the Course Coordinator to clarify before using any other functionality from the standard library.

The Supplied Files

This section gives an overview of the files that you are provided as part of this assignment. You are recommended to take some time to understand the overall structure of the program.

Note

The code you have been provided will not compile or run, given that the necessary classes are not implemented. Particularly, you will see that **`GameDriver.java`** needs many of the classes you will implement in order to compile.

- **`App.java`** – contains the main function that starts the program. **You may change this file for testing, but should ensure your program works with the original.**
- **`GameDriver.java`** – implements the command-line interface, acting as a bridge between the input and your system and containing the main gameplay loop. **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.**
- **`Treasure.java`** – an abstract base class for different types of treasures in the game. It stores common attributes like name and value, and requires subclasses to specify how the treasure affects the player. **This file should not be modified.**
- **`Artifact.java`** – represents rare treasures that add value to the player's collection without directly changing their stats. **This file should not be modified.**
- **`Weapon.java`** – A treasure that increases the player's attack power when collected. **This file should not be modified.**
- **`Potion.java`** – A treasure that restores the player's health points (HP) when used. **This file should not be modified.**

Part 1: Implementation

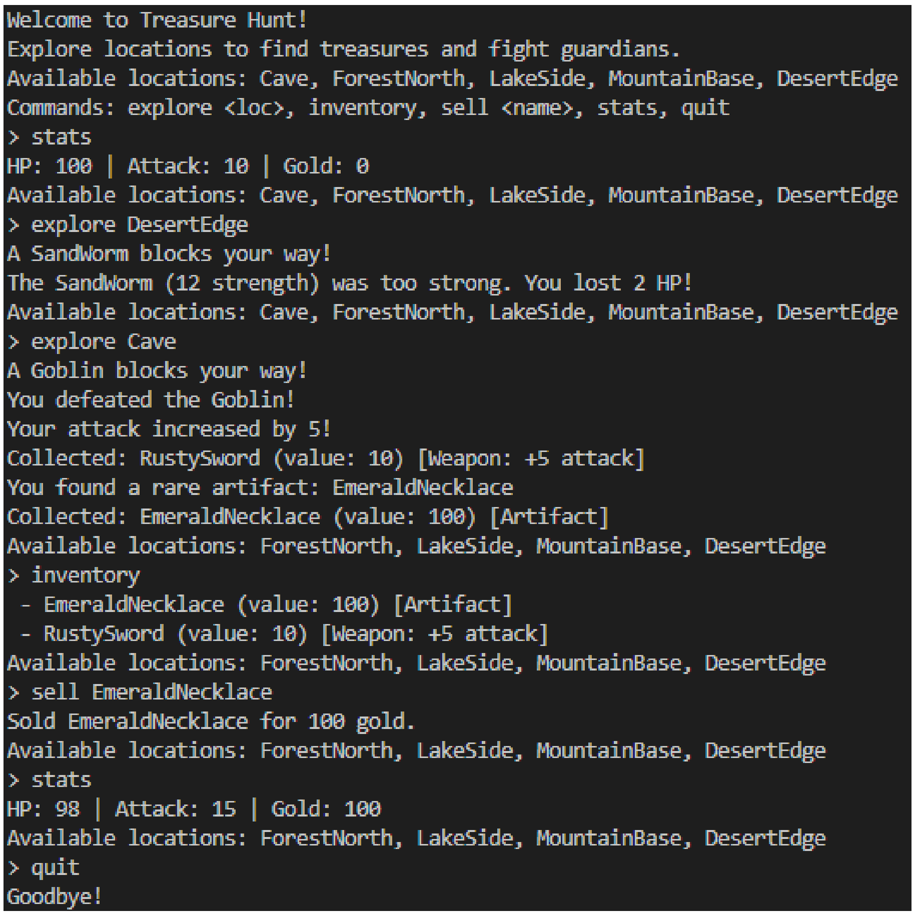
For the implementation part of the assessment (worth 70% of the assessment grade), you are required to implement various classes to support the Treasure Hunt system. In particular, you will need to implement the **`LinkedBinarySearchTree`**, **`ChainingHashTable`**, **`Player`**, **`TreasureChest`**, and **`Guardian`** classes.

Important

Deviating from the specification, even slightly, will result in lost marks, as it means your solution does not meet the requirements outlined in this specification. Precise adherence also demonstrates professionalism and attention to detail, which are critical in real-world software development.

When completing software implementation tasks, it's essential to match the specifications exactly to ensure your solution behaves as expected and can be reliably tested. Test cases are often derived directly from specifications, so any mismatch can make it difficult to verify that the software meets its intended requirements. Similarly, when software matches its specification, future developers can understand, maintain, and extend it more easily. In contrast, deviations introduce ambiguity, which make the system harder to debug, maintain, and extend.

An example of the output produced by the provided **`App.java`**, after running a few commands, is given in Figure 1. Some additional details about each class are provided in the following subsections.



```
Welcome to Treasure Hunt!
Explore locations to find treasures and fight guardians.
Available locations: Cave, ForestNorth, LakeSide, MountainBase, DesertEdge
Commands: explore <loc>, inventory, sell <name>, stats, quit
> stats
HP: 100 | Attack: 10 | Gold: 0
Available locations: Cave, ForestNorth, LakeSide, MountainBase, DesertEdge
> explore DesertEdge
A SandWorm blocks your way!
The SandWorm (12 strength) was too strong. You lost 2 HP!
Available locations: Cave, ForestNorth, LakeSide, MountainBase, DesertEdge
> explore Cave
A Goblin blocks your way!
You defeated the Goblin!
Your attack increased by 5!
Collected: RustySword (value: 10) [Weapon: +5 attack]
You found a rare artifact: EmeraldNecklace
Collected: EmeraldNecklace (value: 100) [Artifact]
Available locations: ForestNorth, LakeSide, MountainBase, DesertEdge
> inventory
 - EmeraldNecklace (value: 100) [Artifact]
 - RustySword (value: 10) [Weapon: +5 attack]
Available locations: ForestNorth, LakeSide, MountainBase, DesertEdge
> sell EmeraldNecklace
Sold EmeraldNecklace for 100 gold.
Available locations: ForestNorth, LakeSide, MountainBase, DesertEdge
> stats
HP: 98 | Attack: 15 | Gold: 100
Available locations: ForestNorth, LakeSide, MountainBase, DesertEdge
> quit
Goodbye!
```

Figure 1: Sample output from running the program

`LinkedBinarySearchTree`

The **`LinkedBinarySearchTree`** is an implementation of the **`BinarySearchTreeADT`** interface. The class should include a nested **`BinaryNode`** class, which holds the data and references to the previous and next nodes. Your 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.

You'll 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));`**

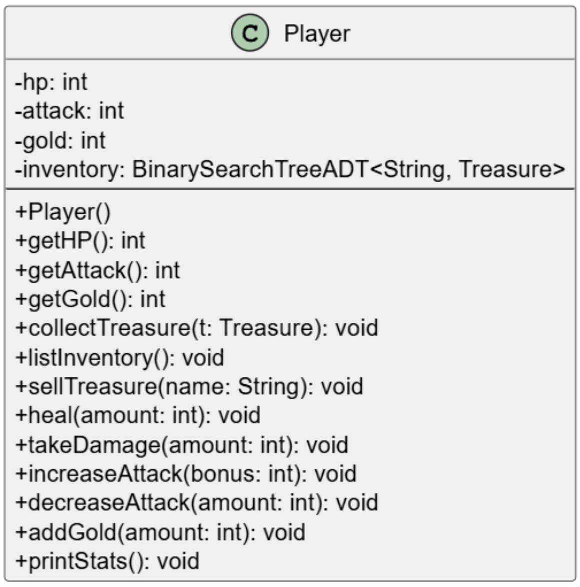
For a full list of methods required and some important details, you should examine the **`HashTableADT.java`** file and its associated documentation.

`Player`

The Player class models the user's character in the game. It tracks the player's stats – HP, attack level, and gold amount – and manages the player's inventory of treasures. The class allows the player to collect treasures, sell items for gold, and display their current inventory and statistics.

As shown in Figure 2, the inventory is managed using a BST where the key is the item's name (**`String`**) while the item itself (**`Treasure`**) is the value.

> **Hint**
>
> This means, any interaction with the player's inventory must be done using the item's name, as it is the key of the tree.



```
Player
-hp: int
-attack: int
-gold: int
-inventory: BinarySearchTreeADT<String, Treasure>
+Player()
+getHP(): int
+getAttack(): int
+getGold(): int
+collectTreasure(t: Treasure): void
+listInventory(): void
+sellTreasure(name: String): void
+heal(amount: int): void
+takeDamage(amount: int): void
+increaseAttack(bonus: int): void
+decreaseAttack(amount: int): void
+addGold(amount: int): void
+printStats(): void
```

Figure 2: UML class diagram for **`Player`**

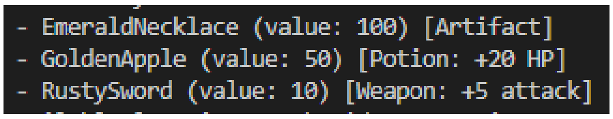
Some further details about the various methods are given below, noting that the method headers/signatures are provided in UML format and are **not** valid Java method signatures.

`collectTreasure(t: Treasure): void`
Inserts a treasure into the player's inventory (i.e., the BST) and applies its effect (e.g., healing, attack bonus).

> **Hint**
>
> The **`Treasure`** abstract class has a method **`applyEffect`**!

`listInventory(): void`
Performs an in-order traversal of the inventory to display treasures sorted by value. This should use a for-each loop over the inventory, which in turn uses the BST iterator, to print the items to the console. Items should be printed one per line, in the format " **`- <item>`**". An example of the output from the printing is given in Figure 3.

> **Note**
>
> The treasure classes have overridden **`toString`**, which should be used to format the output for the item.



```
- EmeraldNecklace (value: 100) [Artifact]
- GoldenApple (value: 50) [Potion: +20 HP]
- RustySword (value: 10) [Weapon: +5 attack]
```

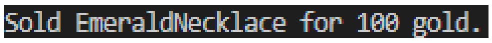
Figure 3: Example output from **`listInventory`**

`sellTreasure(name: String): void`
Sell a treasure from the player's inventory, given its name. This should remove the item from the inventory (i.e., the BST), apply the removal effect, and add the corresponding value to the

player's gold amount. If there is no item with the given name, the inventory should not be modified and no gold should be added to the player's stats.

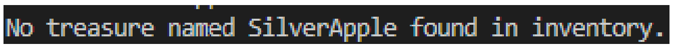
Appropriate status messages should be printed to the console depending on if the item existed or not.

- If the item is found (and sold/removed), a message such as "**`Sold <name> for <value> gold.`**", where **`<name>`** is the name of the treasure and **`<value>`** is its value, should be printed to the console. An example is shown in Figure 4.
- If the item is not found, a message such as "**`No treasure named <name> found in inventory.`**" should be printed to the console. An example is shown in Figure 5.



```
Sold EmeraldNecklace for 100 gold.
```

Figure 4: Example output from **`sellTreasure`** for an item that is sold.



```
No treasure named SilverApple found in inventory.
```

Figure 5: Example output from **`sellTreasure`** for an item name that doesn't exist.

`heal(amount: int): void`
Heal the player by a specified amount by adding the amount to their HP. There is no maximum HP that a player can have.

`takeDamage(amount: int): void`
Inflict damage to the player, reducing their HP. HP has a minimum value of 0.

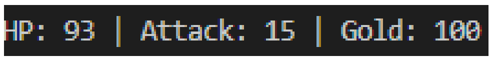
> **Note**
>
> HP cannot go below 0. Logic should be added to ensure that HP is never negative and is set to 0. For example, if a player has 2 HP and takes 3 damage, their HP should be updated to 0 and not −1. The game will end when the player's HP reaches 0.

`increaseAttack(bonus: int): void`
Increase the player's attack power by a specified amount. This is typically used when applying the effect of a **`Weapon`**. There is no maximum attack that a player can have.

`decreaseAttack(amount: int): void`
Decrease the player's attack power by a specified amount. This is typically used when removing the effect of a **`Weapon`**. There is no minimum attack that a player can have, though this should never reach a negative amount in the current game setup.

`addGold(amount: int): void`
Add gold to the player's total. There is no maximum amount of gold that a player can have.

`printStats(): void`
Print the player's current stats: HP, attack power, and gold to the console. The expected format is "**`HP: <hp> | Attack: <attack> | Gold: <gold>`**", as shown in Figure 6.

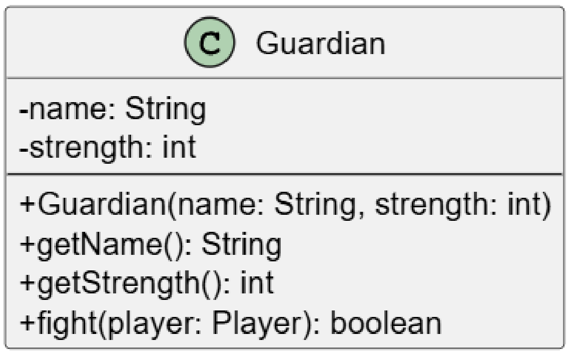


```
HP: 93 | Attack: 15 | Gold: 100
```

Figure 6: Example output from **`printStats`**

Guardian

The **Guardian** class represents an enemy that protects certain treasure chests. Each guardian has a name and a strength value that determines how difficult it is to defeat them. As can be shown in Figure 7, the class is relatively simple, with the primary logic existing in the **fight** method, as described below. **getName** and **getStrength** are simple getter methods, which return the name and strength values, respectively.



Guardian

```
-name: String
-strength: int
+Guardian(name: String, strength: int)
+getName(): String
+getStrength(): int
+fight(player: Player): boolean
```

Figure 7: UML class diagram for **Guardian**

fight(player: Player): boolean

This method compares the guardian's strength against the player's attack power. If the player's attack is greater than or equal to the guardian's strength, the guardian is defeated and the player takes no damage. Otherwise, the player takes damage equal to the difference between the guardian's strength and the player's attack power. The **boolean** return value indicates if the player wins.

A few examples are provided below.

- If the guardian has 5 strength and the player has 8 attack, the guardian is defeated. The player takes no damage. The method should return **true**.
- If the guardian has 5 strength and the player has 5 attached, the guardian is defeated. The player takes no damage. The method should return **true**.
- If the guardian has 8 strength and the player has 5 attached, the player takes 3 $(= 8 - 5)$ damage. The guardian does not have HP, and hences does not take any damage. The method should return **false**.

Appropriate status messages should be printed to the console during the encounter.

- When first encountered, a message such as "**A <name> blocks your way!** should be printed, where **<name>**" is the name of the guardian.
- If the player defeats the guardian, a message such as "**You defeated the <name>!**" should be printed.
- If the guardian is too strong, a message such as "**The <name> (<strength> strength) was too strong. You lost <damage> HP!**", where **<strength>** is the guardian's strength and **<damage>** is the amount of damage taken, described above.

`TreasureChest`

The **`TreasureChest`** class represents a container that can hold multiple treasures. Internally, it manages a collection of items the player can collect. Its primary responsibilities are to store treasures, allow new treasures to be added, provide access to the contents, and indicate whether the chest is empty.

As shown in Figure 8, the treasure chest is managed via an **`ArrayList`**. You do not need to implement this yourself, and instead should directly import as `import java.util.ArrayList`. Similarly, you'll notice that the **`getTreasures`** method will return a **`List<Treasure>`** – this is the Java collections framework list, which you will also need to import as `import java.util.List`.

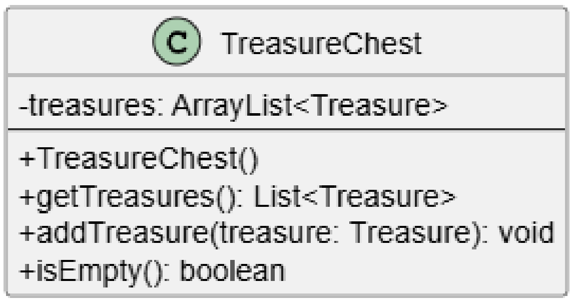

C TreasureChest
-treasures: ArrayList<Treasure>
+TreasureChest()
+getTreasures(): List<Treasure>
+addTreasure(treasure: Treasure): void
+isEmpty(): boolean

Figure 8: UML class diagram for **`TreasureChest`**

Unit Testing

As part of your implementation, you will conduct unit testing of the data structures. Specifically, you must create a test file, named **`DataStructureTests.java`**, where you will use unit testing (using JUnit Jupiter) to comprehensively test the implementation of the data structures, particularly the **`LinkedBinarySearchTree`** and **`ChainingHashTable`**. While there is no set number of tests you must perform, the point is to to consider a set of unit tests that cover the bulk of the functionality, convincing yourself (and the marker!) that the data structures are implemented correctly.

> **Note**
>
> While it would be ideal to have multiple unit testing classes, we will keep it simple and include all our tests in a single file.

> **Note**
>
> You do not need to perform unit testing of any other classes but the data structures listed above as part of the **`DataStructureTests`** class. Notably, you do not need to include unit testing of the **`Player`**, **`TreasureChest`**, and **`Guardian`** classes, but you should otherwise convince yourself that they are correct.

Your test class should include various test methods to verify the functionality of each data structure independently. A **`setUp`** method should be used to reinitialise any shared variables (e.g., a clean instance of the data structures) before each test executes, to ensure a clean state.

Part 2: Report

In addition to a working implementation and unit testing class, you will provide a written report (worth 20% of the assessment grade) that provides a UML class diagram for the system, discusses and evidences your testing process, and includes a reflective account of your experience with the assessment task. The report is expected to be 2-3 pages in length (maximum 12-point font), but there isn't really a strict page limit. Your report should be submitted as a singular PDF.

The report is not meant to be a major endeavour and should prompt your thinking and ensure that you have considered your solution in a bit more depth than just getting the code to compile and submitting your first draft.

Your report should include the following three sections:

1. **UML Class Diagram**: Your task for this section will be to create a UML class diagram for the Treasure Hunt system, including the components provided to you as part of the assignment. That is, you will produce a UML class diagram that includes all concrete classes (you can omit **`App`** and **`GameDriver`**), abstract classes, and interfaces. Your class diagram should include instance variables, methods, and relationships among the classes. See Figure 9 for an example of the relationship types, as generated by PlantUML.

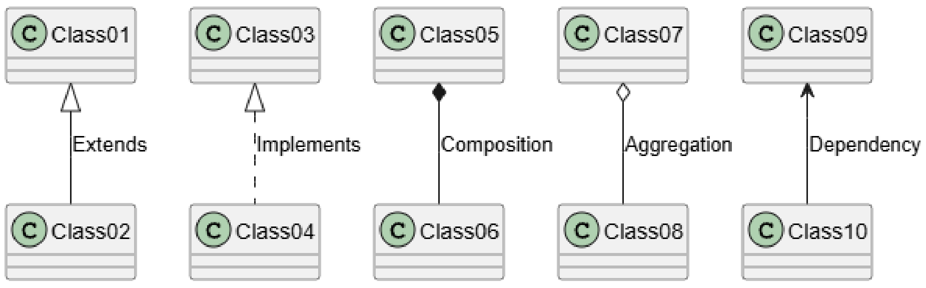


Figure 9: Relationships in UML class diagrams

   You may create the UML class diagram using any suitable tool of your choice (e.g., Microsoft Visio, draw.io, LucidChart, PlantUML). Ensure that you can export (or take a screenshot) to include the diagram in your report. **Note:** UML diagrams included in this assessment specification were created with PlantUML.

   In addition to the LinkedIn Learning course provided on Canvas (**`Modules > Course Materials > Program Foundations: Object Oriented Design`**), some resources you may reference regarding UML class diagrams:

   - Class diagram - Wikipedia
   - Class Diagram | Unified Modeling Language (UML) - GeeksforGeeks
   - UML Class Diagram Tutorial - visual-paradigm.com
   - Class Diagram | PlantUML

2. **Testing**: Describe how you tested your implementation, particularly the data structures. Describe any test cases you conducted and discuss the results. The main point is to consider a reasonable set of tests that cover the functionality, ensuring that the code works as

expected. This is, of course, inherently linked with the unit testing class that you wrote for this assessment.

> **Recall**
>
> We don't only report on our final, successful test executions. Rather, we use this as a log and timeline of our testing and refinement process. Particularly, our test process should highlight failed tests, then describe how we rectified the error(s) to lead to passing tests in subsequent executions.

You don't necessarily need to test and report on every method but should ensure that you have convinced yourself (and the marker) that you have a working solution beyond the functionality shown in the provided `App.java`.

This may be easiest to provide in tabular form, such as:

Test ID	**Description**	**Expected**	**Actual**	**Notes / Actions**
1	Add 3 and 5	8	6	Failed. Result appears to be adding LHS twice, rather than LHS + RHS. Revised the implementation of the `add` function to fix.
2	Repeat of add 3 and 5 after fix described in Test 1	8	8	Success.

3. **Reflection**: Reflect on what you learned from this assignment and how you could improve your implementation. When writing the reflection, consider aspects such as challenges you faced, strategies you used for debugging your implementation, and how you followed the software development life cycle process. For example, consider questions such as:

   - *Are there areas where error checking would make sense, but were not included in the specification?*
   - *Is there a different design that would be better for this problem?*
   - *Is there another data structure that would be more efficient for (some of) the operations?*

   Again, we are mainly just looking for a few paragraph summary of how your skills were improved through this assignment and if you have any ideas for ways things could have been improved in your solution.

   You can find more information about reflective writing in the library guide found here: What is Reflective Writing?

Marking

This section provides some general information about how you will be evaluated. Marking criteria will accompany this assignment specification and will provide an indicative guideline on how you will be evaluated.

Your core implementation (including the test class) will account for 70 marks and will be assessed primarily on correctness. An additional 10 marks will be allocated to commenting and code quality. This means, in addition to providing a correct solution, you are expected to provide readable code with appropriate commenting, formatting, and best practices.

Remember that your code should 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.

> **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. You should expect that your code will be tested on functionality not explicitly shown in the supplied files (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.

> **Important**
>
> 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.

Your report will account for the remaining 20 marks and will be assessed on completeness, depth, and general writing quality.

Please see the accompanying marking criteria for an indicative guideline to how you will be evaluated.

> **Note**
>
> The marking criteria is subject to change without notice, as necessary.

Submission

Your submission should be made using the Assignment 2 link in the Assignments section of the course Canvas site. Assignment submissions will not be accepted via email. Incorrectly submitted assignments may be penalised.

For the implementation task, your submission should include only the completed versions of the files specified earlier. You do not need to include any other files in your submission, such as those provided to you as part of this assignment. Also, ensure that you name your files using the standard Java convention, where the name of the file is **`<className>.java`**. For example, your class **`LinkedBinarySearchTree`** should be contained in a file **`LinkedBinarySearchTree.java`**.

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

> **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.

Helpful Tips

1. Read the comments in 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.