Haskell TDD

Evaluating Test Driven Development with the Haskell Functional Programming Language

Ryan Gorman (N0809433)
5-6-2022

Contents

Table of Acronyms

ACRONYM	DEFINITION
TDD	Test-driven Development
TLD	Test-last Development
IBM	International Business Machines
FVT	Functional Verification Test
BST	Binary Search Tree
OOP	Object-oriented Programming

List of Figures

Acknowledgements

Neil Sculthorpe – Advanced Software Engineering Lecturer at Nottingham Trent University.

Thomas Cooper – Pair Programming Partner. Student at Nottingham Trent University.

Explaining TDD

1.1 Why Testing Is Crucial Throughout Development

Test driven development is a process employed by software developers to ensure the functionality of code is maintained through the expansion and refactoring of a project. It was introduced as a methodology for reducing error prone development and improving a developers error identification ability.

This process involves creating tests throughout the development phase of a project to validate the functionality of a project while it is being developed, instead of writing all anticipated test cases prior to development as per the test first development approach, or creating test cases following development, as per the test last development approach.

As stated by Kent Beck (2002) in his book on test driven development "The more stress you feel, the less testing you will do. The less testing you do, the more errors you will make. The more errors you make, the more stress you feel. Rinse and repeat." (Beck, 2002), there is a clear cycle where increased stress leads to a direct decrease in testing, leading to a direct increase in errors generated through development.

An illustration of this process can be seen in figure 1, an influence diagram between pressure, testing and errors produced. In this diagram, an arrow with a circle indicates an inverse correlation, from the first object to the next, and an arrow without a circle indicates a positive correlation. In this case, we see that figure 1 supports this concept, that an increase in pressure, leads to less testing, which leads to more errors, leading to more pressure.

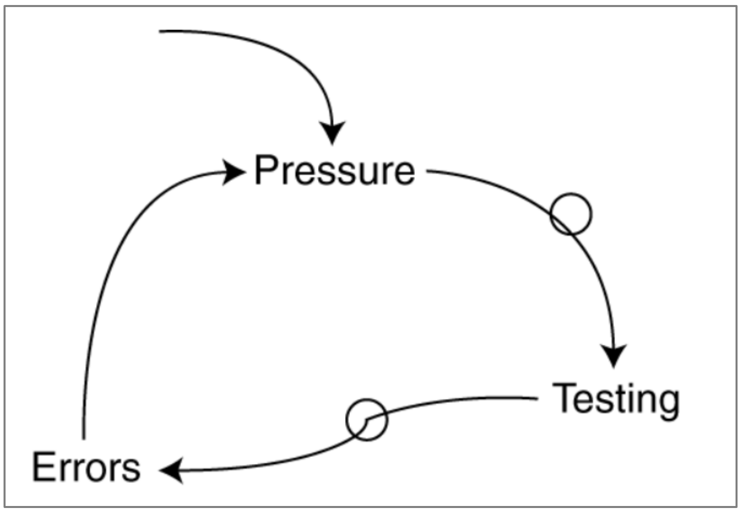


Figure 1. The "no time for testing" death spiral (Beck, 2002)

As mentioned, test driven development is an approach in which an initial set of tests are written before development starts, and more tests are written as the development is expanded, which is clearly illustrated in figure 2. Following this, the first step of this three-step approach is to write a number of test cases, expecting them to fail, as no development has been complete at this stage. Minimal functionality is then developed as part of step two, to pass said test cases. Once these tests pass, the developed code is then refactored to be written to a higher standard and quality.

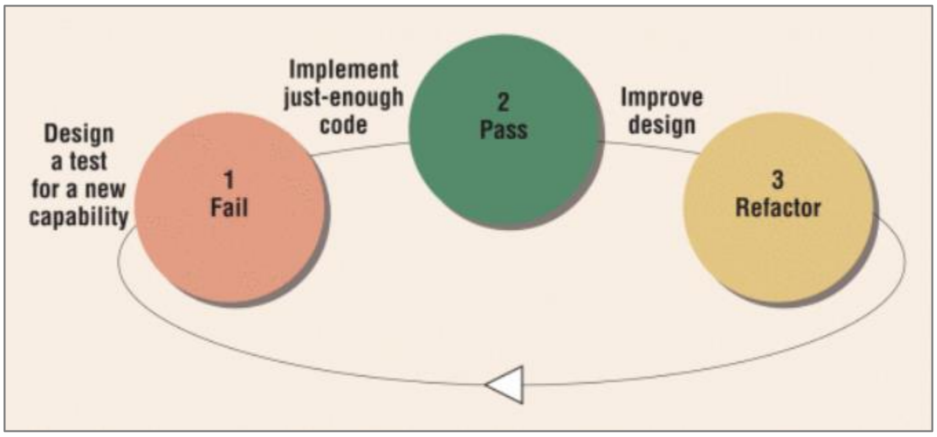

Design a test for a new capability
1 Fail
Implement just-enough code
2 Pass
Improve design
3 Refactor

Figure 2. TDD Three Step Cycle (Jefferies, Melnik, 2007)

As noted by Beck in his book "Test-driven Development: By Example" and illustrated by Jefferies and Melnik in figure 2, a common conception for TDD is that it consists of three steps. This is supported through many varying diagrams, illustrations of other forums, articles and books discussing the subject. However, in practice, it is my opinion that it is largely favourable and more appropriate and professional to consider a fourth step, where the original test cases are run again to verify the final refactoring of code does not cause any previously passed tests to now result in failure. A clear illustration of the TDD process in more detail, including this additional step can be seen in the form of a flow diagram in figure 3.

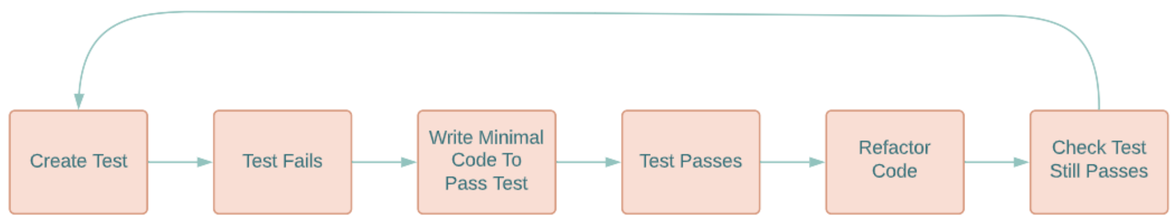


Figure 3. TDD Flow Diagram

Through personal experience developing projects while employing TDD, the fourth step of verifying the initially passed test cases still pass after refactoring has been complete, is an essential part of testing, as a common mistake when refactoring code is to lose track of what the method is doing at its core. This leads to programs removing or forgetting to re-include key functionality within methods or combinations of methods, leading to the functionality no longer working. Testing this functionality after refactoring the code prevents this issue for developers as they recognise immediately if the refactoring causes test failures.

1.2 TDD in Practice

IBM developed a project while following the TDD process to evaluate the impact it has on reducing error prone development, application design, the overall productivity of a team, and how well the project schedule is followed.

Following this project, a number of findings were made surrounding the benefits of TDD and whether other companies and projects should take TDD as a consideration for their projects going forward.

IBM identified several benefits, stating the "Benefits of automated testing include the following: (1) production of a reliable system, (2) improvement of the quality of the test effort, (3) reduction of the test effort and (4) minimization of the schedule [5]." (Maximilien and Williams, 2003), supporting the premise of TDD as it aims to support all these areas.

IBM states in favour of minimising project scheduling that "We believe that the TDD practice aided us in producing a product that would more easily incorporate late changes." (Maximilien and Williams, 2003). This is largely due to the process of TDD, as it requires developers to create test cases for every minor piece of functionality developed. Meaning future expansions on a program that lead to issues in previously developed code can be more efficiently identified, as the specific code producing errors is noted as soon as test cases are rerun.

In my opinion, the most important benefit and justification for employing TDD within a development project is the decrease in defects and errors produced by code expansions or refactoring. "With TDD, unit testing is an integral part of code development. As a result, we achieved a dramatic 50% improvement in the FVT defect rate of our system." (Maximilien and Williams, 2003), IBM notes that a significant decrease in the rate of defects can be seen as a result of developing a project following the TDD process.

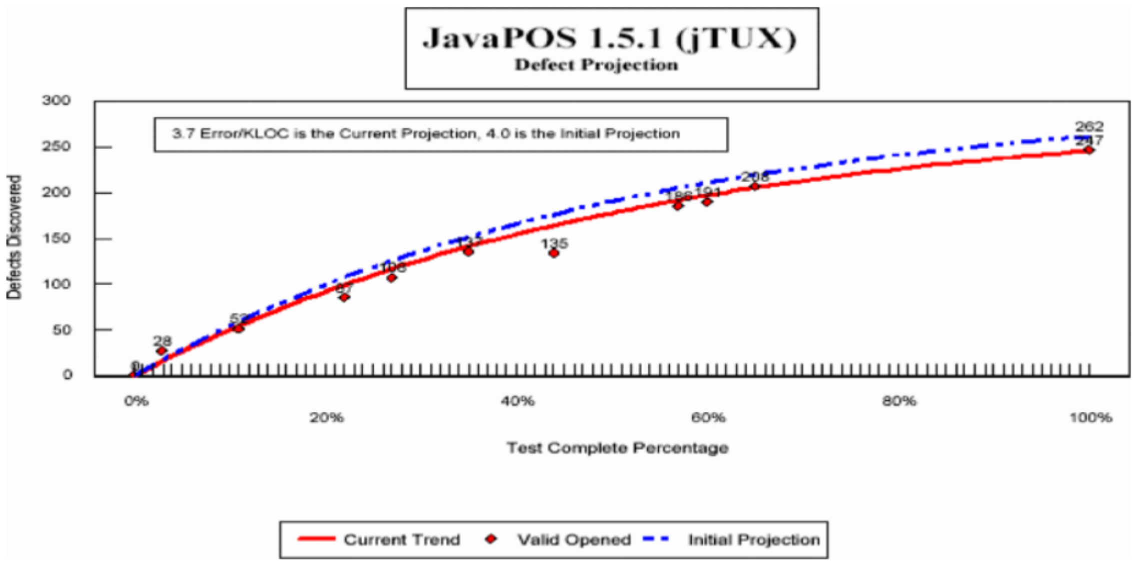


Figure 4. FVT Defect Projection (Maximilien and Williams, 2003)

This benefit can be visualised through figure 4 where the defect rate as development continues, is shown as a negative correlation when plotted against the number of tests complete.

At 50% test completion there is a defect rate of approximately 150, leading to an estimated final result of defects at 300. However, as shown by the final number of defects detected at 247, and the initial projection plotted against the current trend.

The expected result was 262 defects detected following this process, however it was even less as stated, at 247, showing the impressive benefit of TDD when reducing defects in a project.

1.3 Types of Tests

Now that the overall process of TDD and the benefits that come of this have been discussed in detail, the actual tests that should be created within the first step of the process can be identified.

When creating test cases to verify the functionality of a program, they can either be automated or interactive (require input or manual operation). For this project, I will only be discussing automated testing as it provides the most benefit and is applicable in all relevant testing cases.

Writing test cases can be separated at a high level into unit tests and property-based tests. Property based tests are written to allow for varying inputs to be taken but will always return the same output. For example, a property-based tests for a method checking if a number was divisible by ten, would be iteratively triggered with varying inputs, all of which returning true or false. This allows the developer to verify at a high level whether the expected behaviour of a method is correct. An example of a property-based test written in the Haskell programming language can be seen in 2.3 TDD Using Haskell, figure 5, and will be discussed in detail in that chapter.

Unit tests are written to validate more specific scenarios. Following the example given above, a unit test would be written to take the same input, for example ten, and specify the expected output as true. This unit test would not be altered to allow for differing inputs, it only tests for this specific case. An example of a unit test written in the Haskell programming language can be seen in 2.3 TDD Using Haskell, figure 6.

Applying TDD

2.1 Haskell Functional Programming Language

Haskell is a programming language designed with reducing developer errors by producing more readable, functional, and pure code.

Functional programming is a paradigm in which methods are written to be more readable, and easier to debug and test compared to OOP and Procedural. One branch of functional programming is pure code, which prevents changing state, meaning methods do not interact with or manipulate existing local or global variables and data structures by avoiding use of flow-control statements including loops, breaks, continues, and gotos.

As discussed in Tyson's "What is functional programming?" (2021), the way functional programming languages get around not including these flow-control statements, is by creating recursive methods, using "first class functions". A first-class function is a function passed as an argument to another function to create recursion. How this concept is employed within this project can be seen in all of the methods produced to implement a BST, as all of methods require a traversal at some stage. These methods can be seen and are explained in more detail in 2.3 TDD Using Haskell.

2.1.1 Pair Programming

The TDD process for implementing a BST is demonstrated through pair programming, it is stated in Böckeler and Siessegger (2020) that the pair programming practice consists of a "Driver" and a "Navigator". The Navigator in this instance is largely coming up with ideas and design solutions to problems, while the Driver is writing the code to implement the Navigators ideas.

The idea is that the developer in each role will swap at varying intervals. These intervals can vary from pair to pair, but generally speaking, an interval can be one day at a time, one unit test at a time, one method at a time, or even one task at a time.

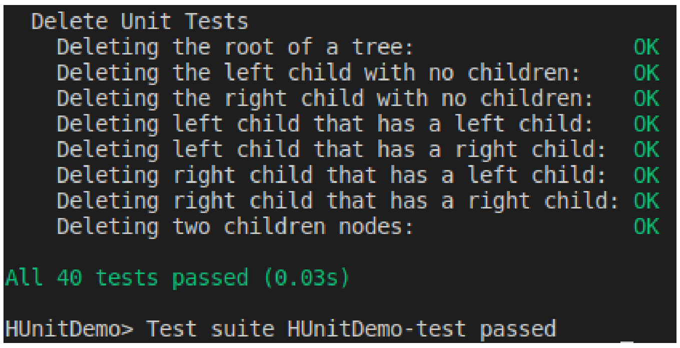
2.2 TDD Using Haskell

2.2.1 Test Environment Setup

Before creating test cases, a testing environment is required. Stack is a Haskell dependency management tool that automatically creates a project structure containing a "*src*" and "*test*" folder for core development and test case creation.

Once the project structure is created, a testing framework defining how unit and property-based tests should be written is required. HUnit is the most common choice for testing frameworks when creating Haskell projects. It is a testing framework designed specifically for Haskell, that allows for the creation of property-based and unit tests.

Employing a testing framework allows the developer to create test cases that are automatically executed by the framework, deciding if the test passes or fails. Once these tests are run, the framework also provides a summary output displaying the results of all of the tests, failed or passed, and if they failed, the suggested reason why the test failed. A screenshot of a number of test cases created later in the project, with a pass result, is shown in figure 5.



```
  Delete Unit Tests
    Deleting the root of a tree:                 OK
    Deleting the left child with no children:    OK
    Deleting the right child with no children:   OK
    Deleting left child that has a left child:   OK
    Deleting left child that has a right child:  OK
    Deleting right child that has a left child:  OK
    Deleting right child that has a right child: OK
    Deleting two children nodes:                 OK

All 40 tests passed (0.03s)

HUnitDemo> Test suite HUnitDemo-test passed
```

Figure 5. Unit Tests Result Sample

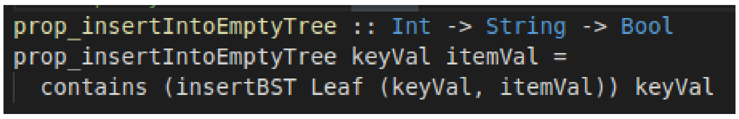
Lastly, before creating test cases. It is useful to have a tool for managing how the test cases are laid out and organised once they are created. "*Tasty*" is a Haskell test management tool that allows the developer to organise test cases into "trees" also known as groups of tests. Examples of which will be discussed during test development below.

2.2.2 Insert Method

Following the first step of the TDD cycle, my pair programming partner Thomas Cooper and I created a number of property-based and unit test cases to verify the soon to be developed simplified insertion and contains methods.

A property-based test for inserting a single node into an empty tree was created, employing the contains method, to verify that the node exists within the tree, by expecting a true or false result from the contains method.

As shown in figure 6, the first property-based test for an insertion, requires two inputs, an integer and a string, and produces a Boolean as a result. The integer represents the unique key of the node, the string represents the value associated with the key, and the Boolean output represents whether the node exists within the tree.



```haskell
prop_insertIntoEmptyTree :: Int -> String -> Bool
prop_insertIntoEmptyTree keyVal itemVal =
    contains (insertBST Leaf (keyVal, itemVal)) keyVal
```

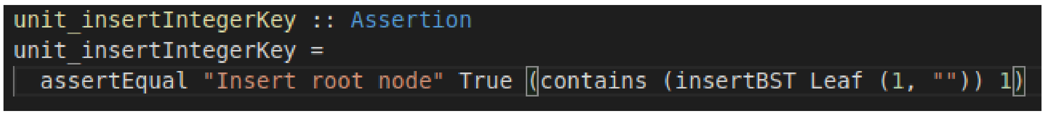
Figure 6. Haskell Insertion Property Based Test

Because property-based tests do not test specific scenarios, and instead test several scenarios against the overall output of the method, a library called "QuickCheck" was employed to generate random data for this test and future property-based tests.

Quick Check is a library that provides varying sized lists of differing contents such as integers, strings, or doubles. It allows for the recursive insertion of a large number of nodes into the BST without having to needing the developer to manually specify the exact key and value for each node. This is shown in figure 6 by the use of "keyVal" and "itemVal", instead of actual values.

Figure 7 shows the first unit tests created for the insert method. It is very similar to the first test case written for the contains method which can be seen in figure 8 in chapter 2.3.3, as both test cases are using the same methods to verify if the node exists in the BST following an insertion.

This test is an assertion that the root node containing "1" as a key, is present in the tree after being inserted. It operates by inserting a Leaf node into the tree containing a key of "1" and an empty string value (as it is beyond the scope of this test), and then performs a contains operation looking for a node containing a key of "1". If the test case returns true, it is considered a pass, and if it returns false, it is considered a fail.

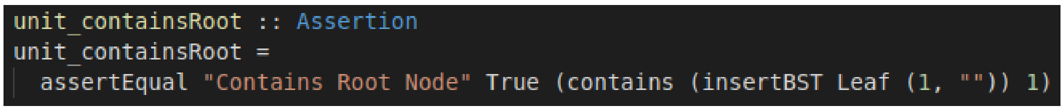


```haskell
unit_insertIntegerKey :: Assertion
unit_insertIntegerKey =
  assertEqual "Insert root node" True (contains (insertBST Leaf (1, "")) 1)
```

Figure 7. Haskell Insertion Unit Test

2.2.3 Contains Method

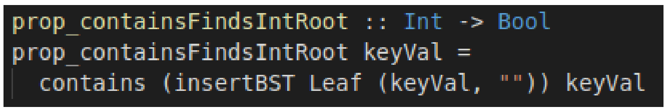
Separate property-based tests and unit tests were created to validate the contains method as well, of which the unit test can be seen in figure 8**.** Although, as noted previously, an insertion is required for the contains method, and the contains method is required to test the insertion, both test cases for insertion and contains are very similar.



```haskell
unit_containsRoot :: Assertion
unit_containsRoot =
  assertEqual "Contains Root Node" True (contains (insertBST Leaf (1, "")) 1)
```

Figure 8. Haskell Contains Unit Test

Given that the property-based test created for the initial development of the contains method operates similarly to the property-based test for a root node insertion. This test performs an insertion of a node indicated by "*keyItem*" and "*keyVal*" and then performs a contains operation, returning true if the node exists. This property-based test can be seen in figure 9.



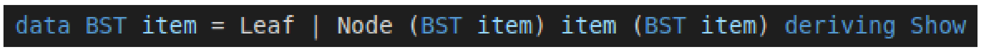
```haskell
prop_containsFindsIntRoot :: Int -> Bool
prop_containsFindsIntRoot keyVal =
  contains (insertBST Leaf (keyVal, "")) keyVal
```

Figure 9. Haskell Contains Root Node Prop Test

After verifying that these test cases are unsuccessful, the development for both the contains method and insert methods began. Following the test cases defined, a data type for a BST Leaf to represent an empty pointer, and Node containing a key, value, and two leafs, were created, which can be seen in figure 10.

Then the contains method was developed to take an input of a BST type and a sought key, outputting a Boolean value. The insert method was developed to take a

BST type and a pair containing the key and value for the node being inserted, outputting a new BST containing the new node. A new BST is created as an output of an insertion, as in functional programming local variables states cannot be changed, therefore a new item type is required to insert the new node.



```haskell
data BST item = Leaf | Node (BST item) item (BST item) deriving Show
```

Figure 10. Haskell Data Type

These methods were specifically developed not to include complex traversals at this stage to ensure the core functionality of creating the BST with a root node was working.

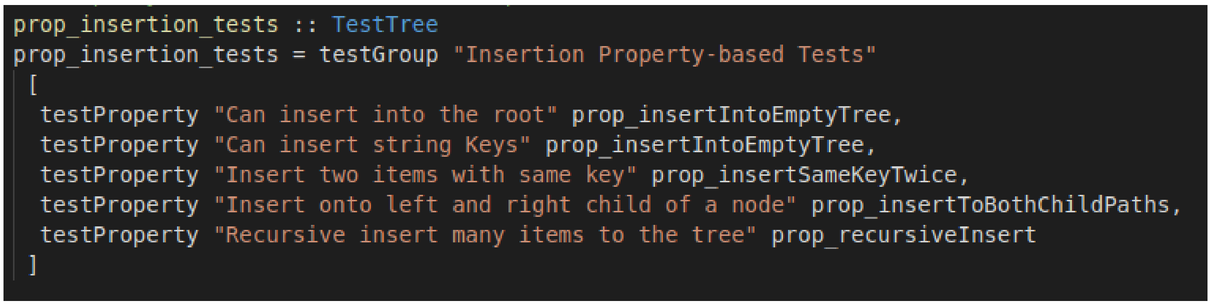
Once these test cases had passed, more test cases were developed by myself and Thomas Cooper to account for varying cases of use and misuse for insertion, contains, and in-order, including a test case for inserting multiple nodes into a BST, which verifies whether the insert method can insert multiple values, into the correct positions of the tree. This is a necessary test because a correct tree structure also determines the success of the contains, in-order and deletion methods, this test case can be seen in Appendix C figure 23.

These tests also include performing a contains operation on a multi-node BST, outputting the results of a traversal of one or more nodes. This test case can be seen in Appendix C figure 24, and is a necessary test case as it tests the codes ability to correctly traverse the tree to find the sought node.

Another test case that was created was inserting two nodes with the same unique key into the tree, and can be seen in Appendix C figure 25**.** This was a necessary test case as it checks the fault management of the node insertion, because the tree cannot contain more than one node with the same key, otherwise deletion, in-order, and contains methods would all be operating incorrectly.

Another example of a contains test case is shown in Appendix C figure 26**,** where three nodes are inserted into a BST, a root node, a right child and a right child to that right child. This tests the contains method ability to traverse the tree with more than one child and is necessary because most BST structures will contain more than three nodes in their entirety.

These test cases were organised into test groups, technically referred to as "trees", which are introduced in 2.2.1 Test Environment Setup. An example of a test tree can be seen in figure 11 with the insertion method unit tests group.

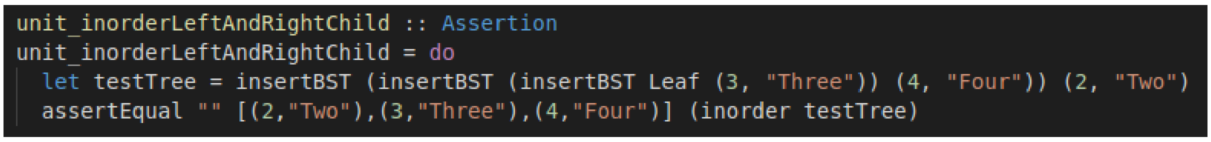


```
prop_insertion_tests :: TestTree
prop_insertion_tests = testGroup "Insertion Property-based Tests"
 [
  testProperty "Can insert into the root" prop_insertIntoEmptyTree,
  testProperty "Can insert string Keys" prop_insertIntoEmptyTree,
  testProperty "Insert two items with same key" prop_insertSameKeyTwice,
  testProperty "Insert onto left and right child of a node" prop_insertToBothChildPaths,
  testProperty "Recursive insert many items to the tree" prop_recursiveInsert
 ]
```

Figure 11. Haskell Deletion Test Group

2.2.4 In-order Method

An example of an in-order traversal test case, asserting on a BST containing three nodes, verifying that the output from a traversal is an array of pairs containing the key and value for each node, in the correct order, can be seen in figure 12.



```
unit_inorderLeftAndRightChild :: Assertion
unit_inorderLeftAndRightChild = do
  let testTree = insertBST (insertBST (insertBST Leaf (3, "Three")) (4, "Four")) (2, "Two")
  assertEqual "" [(2,"Two"),(3,"Three"),(4,"Four")] (inorder testTree)
```

Figure 12. Haskell In-order Unit Test

Further examples of in-order traversal test cases include creating a BST with a root node, and a single right or left child, which can be seen in Appendix C figure 27.

Another test case was written for a root node with a right child, which has a child of its own, which can be seen in Appendix C figure 28, and also, to cover all basic functionality, a test case was written to account for a BST with only a root node and no children, which can be seen in Appendix C figure 29. This last test case was written to check that the tree does not produce an error for attempting to access a child that does not exist.

Following these new test cases, the insertion code was expanded to account for insertion of multiple nodes and include decisions to determine the appropriate resulting structure of the BST, which can be seen in Appendix C figure 19. The contains method was then expanded to include the functionality to traverse the tree, depending on if the sought node was larger or less than the current node in the tree. Lastly, the in-order traversal method was developed in full, as it has no complex variation, it simply traverses the tree and outputs node values at certain points of recursion, which can be seen in Appendix C figure 21.

2.2.5 Refactoring

Following the TDD cycle, these methods were refactored to be better written and more functional. Due to the nature of typing "insertBST" for each separate node inserted into the tree, it prevents the use of large BSTs in testing in an efficient and practical way. A new method was developed within the source code that recursively calls the insert method to insert nodes into a BST by passing an array of keys and values to the method, instead of calling the method manually multiple times.

This new method allowed for the existing test cases to be improved, to no longer require the old syntax for an insertion shown in figure 12, can be seen in figure 14.

The primary benefit for this additional method however, was to create more efficient property-based tests, as these originally created a single insertion for each test. These test cases now create one hundred iterations of each test case to demonstrate the test is valid with any entry value.

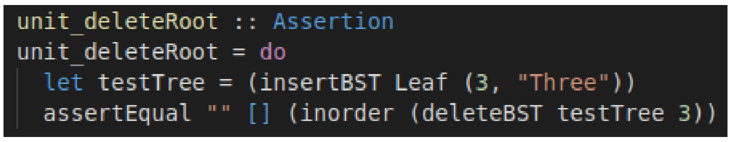
2.2.6 Removal Method

Once the development for contains, insert and in-order was complete, and the test cases had passed, new tests were written for the deletion method that will be developed.

These test cases employed the in-order method to check against expected results from the BST, however these test cases could also be verified using the contains method to check the node no longer exists in the tree, but the output of the in-order method also lets us check that the expected structure of the tree is true following the removal, and that the child of a parent node being removed is not just placed anywhere in the tree.

The first deletion test case was to remove the root node from the tree with no children, therefore no traversing was required, and no decisions had to be made about altering pointers to reposition nodes.

Per figure 13, this test case performed an insertion of one node with a key of 3 and a value of three, this was then followed by performing deletion of the node with the key of 3 and performing an in-order traversal of the BST following this. The expected result is an empty dictionary, as there are no longer any nodes present.



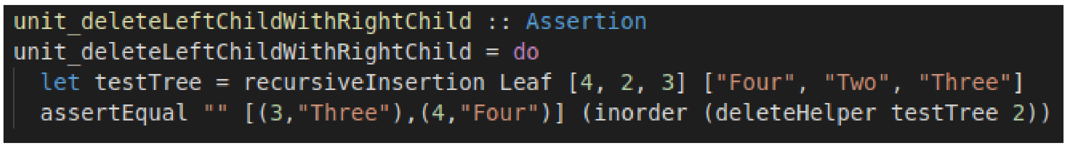
```haskell
unit_deleteRoot :: Assertion
unit_deleteRoot = do
  let testTree = (insertBST Leaf (3, "Three"))
  assertEqual "" [] (inorder (deleteBST testTree 3))
```

Figure 13. Haskell Unit Test Removing Root Node

Following this test case, the initial deletion method was developed to remove a node without requiring any traversal, to ensure the core functionality of removing a node. This method would take an integer and remove the node if it matched, otherwise, it would return the node provided.

Once this test case had passed, additional test cases were developed to account for the removal of a node from a BST with multiple children.

These test cases included removing the left and right child of a root node with and without another child in varying combinations to account for most core possible BST structures. This test case is necessary to test that the parent node being removed is correctly replaced by its child node. An example of one of these test cases can be seen in figure 14.



```haskell
unit_deleteLeftChildWithRightChild :: Assertion
unit_deleteLeftChildWithRightChild = do
  let testTree = recursiveInsertion Leaf [4, 2, 3] ["Four", "Two", "Three"]
  assertEqual "" [(3,"Three"),(4,"Four")] (inorder (deleteHelper testTree 2))
```

Figure 14. Haskell Unit Test Removing Child Node with a Child

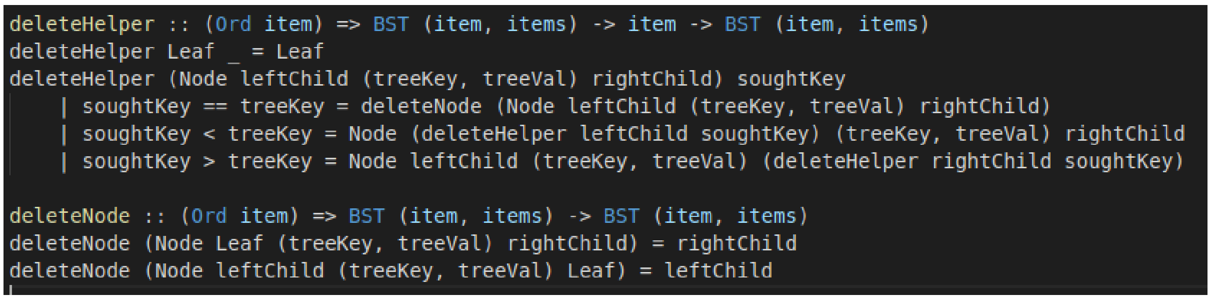
It is worth noting that the functionality for removing a node with two children was decided against being developed by Thomas Cooper and myself due to the complexity of the concept. With this in mind, test cases for this scenario were not developed either.

In varying combinations as mentioned, another test case for deletion is the removal of a right child, with a left child below it, testing that the left child replaces its parent, similarly to the test case shown in figure 14. This test case can be seen in Appendix C figure 30.

Another test case is testing that the tree structure is correct following the removal of two nodes at once, instead of removing just one node. This test is useful for checking the BST remains correctly structured following intensely demanding operations. This test case is shown in Appendix C Figure 31.

Following the creation of these test cases, the removal development was expanded to include traversing the BST to search for the desired node, as well as checking if the desired node has a child, and whether that child was on the left or right of the parent as this child is needed to replace the position of the node being removed.

This method was designed to follow the test cases by taking an input of a BST and a desired node key, outputting a new BST with the desired node removed. The logic within this method can be seen in figure 15. At a high level, it consists of traversing the tree to find the node key and then calling a removal function, then checking if this node contains a child. If the node contains a left child or right child separately, the child node replaces the parent node in the new BST.



```haskell
deleteHelper :: (Ord item) => BST (item, items) -> item -> BST (item, items)
deleteHelper Leaf _ = Leaf
deleteHelper (Node leftChild (treeKey, treeVal) rightChild) soughtKey
    | soughtKey == treeKey = deleteNode (Node leftChild (treeKey, treeVal) rightChild)
    | soughtKey < treeKey = Node (deleteHelper leftChild soughtKey) (treeKey, treeVal) rightChild
    | soughtKey > treeKey = Node leftChild (treeKey, treeVal) (deleteHelper rightChild soughtKey)

deleteNode :: (Ord item) => BST (item, items) -> BST (item, items)
deleteNode (Node Leaf (treeKey, treeVal) rightChild) = rightChild
deleteNode (Node leftChild (treeKey, treeVal) Leaf) = leftChild
```

Figure 15. Haskell BST Delete

By the end of testing, an insertion, contains, in-order traversal and removal method were developed, all following the TDD process, and successfully accomplishing the desired functionality while also passing all designed test cases.

Reflection

3.1 What Went Well

3.1.1 Following the TDD Process

In my opinion, the project provided useful insight into the TDD process and how it is viewed professionally and through studies on the practice, as well as my personal experience applying it. I feel as though the TDD process was then followed accurately with this research in mind, planning what methods would be developed and how, before creating any unit or property-based tests. Then following these ideas up with good pair programming practice to develop useful and meaningful test cases, that provided a useful guideline to follow in development.

I feel as though the overall testing for the application went well, with lots of use cases being developed to thoroughly test against the functionality of the code developed. The addition of property-based testing was also a very useful feature for the overall quality of the testing, especially once the recursive insertion method was developed.

I also feel that the development for the project went very well. There were issues to begin with when trying to implement the initial methods, but this is discussed in more detail in 3.2 Challenges Faced. The implementation achieved, including all insertion, contains, and in-order cases, and most of the removal cases with the exception of removing a node with two children, was a great achievement for myself and Thomas Cooper as we were worried about how difficult it would be to learn the language at first.

3.1.2 Pair Programming

Applying pair programming throughout this project provided a great addition to the project workflow and overall reduction in unforced developer errors, as myself and Thomas Cooper took turns suggesting ideas and typing code, both error checking each other's code as it was typed.

There were a number of challenges faced at the beginning, regarding pair programming remotely as myself and my partner do not live near each other, and with balancing other workloads. These challenges were handled well, following the advice found through research into Böckeler and Siessegger (2020), and by the end of the project, a good routine was developed between the two of us.

3.2 Challenges Faced

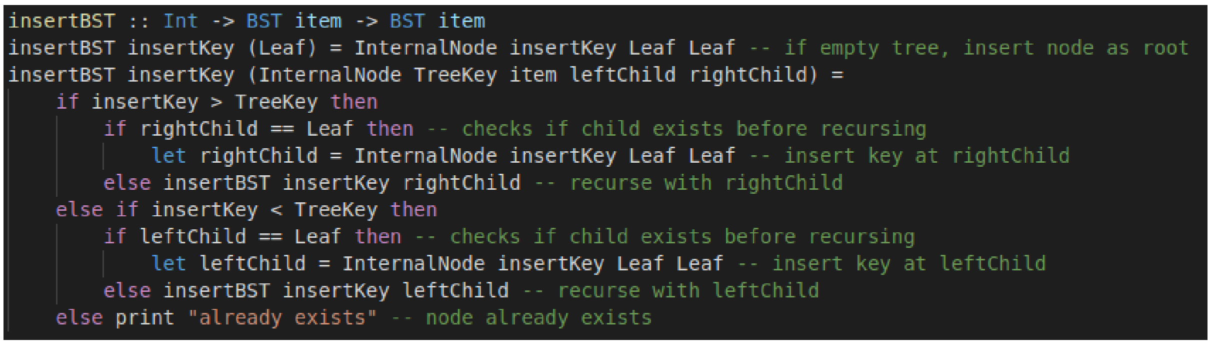
3.2.1 Learning Haskell

When learning the language, there was an initially steep learning curve, which I feel could have been aided greatly with more patience and perspective. A personal challenge I have when developing code is to become anxious about the time frame required to learn the language, and understand existing solutions to develop my own solutions, and therefore I try to find shortcuts, learning only the essential syntax to understand existing solutions, instead of getting a broad understanding of the language as a whole, and going from there.

This leads to issues like one I experienced when developing the insertion method with Thomas Cooper during our pair programming. As we did not have a better understanding of the Haskell language, we attempted to create the insertion method following the same logic and similar conditions needed to develop this method in C++, a screenshot of which can be seen in figure 16.

This produced lots of errors, as in C++ it is required to develop move and copy operations to implement a BST, but in Haskell it is not. This is because Haskell does not control local state using pointers the same way that C++ does, changing the state of the pointer throughout a loop, instead, Haskell is developed in recursive methods to get around this issue.

It was only through research into produced errors and spending more time reading into the language that we began to make better use of pattern matching and recursive functions, to create a more appropriate and practical method for insertion.



```haskell
insertBST :: Int -> BST item -> BST item
insertBST insertKey (Leaf) = InternalNode insertKey Leaf Leaf -- if empty tree, insert node as root
insertBST insertKey (InternalNode TreeKey item leftChild rightChild) =
    if insertKey > TreeKey then
        if rightChild == Leaf then -- checks if child exists before recursing
            let rightChild = InternalNode insertKey Leaf Leaf -- insert key at rightChild
        else insertBST insertKey rightChild -- recurse with rightChild
    else if insertKey < TreeKey then
        if leftChild == Leaf then -- checks if child exists before recursing
            let leftChild = InternalNode insertKey Leaf Leaf -- insert key at leftChild
        else insertBST insertKey leftChild -- recurse with leftChild
    else print "already exists" -- node already exists
```

Figure 16. Haskell Original Insert Method

Through the aid of existing solutions and thorough research, I found several examples for implementing an insertion method for binary trees and BSTs. Two of which I found particularly useful Kedrigern (2013) and Hsinewu (2016). These two existing solutions provided very useful insights into applying pattern matching and recursion, not only for developing a BST, but specifically within the Haskell language. With this understanding, Thomas Cooper and myself were better able to produce a better written insertion method, shown in Appendix C, figure 19**.**

3.3 Test-driven Versus Test-last Development

Test-last development is a practice in which the developer completes all development before creating any test cases. There are a number of functional performance differences between TDD and TLD, but in my personal experience, I prefer TDD over TLD simply due to the structure it gives the developer, providing them with a less confusing work environment as they are encouraged to test everything as they go. TLD leads to lots of misunderstood errors arising and leads to unnecessary amounts of time spent searching for the root of a problem.

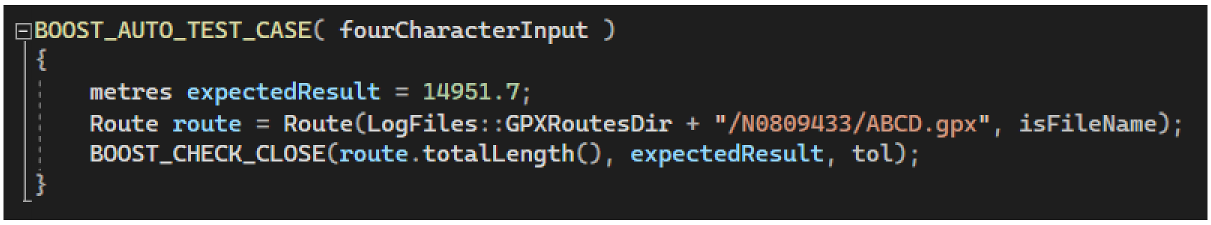
A key difference in deciding which approach a developer should take when starting a new project is how much importance they place on incremental design. By this, I mean, TDD follows an approach of creating a simple version of the code first to check it passes the test, then creating more refactored and advanced versions of the code, whereas TLD attempts to create the finished design from the beginning, avoiding gradual designs.

As mentioned, I personally prefer TDD, as even though TLD appears like the easier option, leaving all the testing until the end, allowing the developer to enjoy developing for longer, in reality, as shown through research into IBM's case study, TDD provides a reduction in defects that TLD cannot provide, making the job of the developer less enjoyable.

3.3.1 C++ Unit Test Cases

C++ is another language that is well equipped for developing automated unit testing, whether that is employed with TDD or TLD.

As shown in figure 17, C++ provides a library called "Boost", for creating unit tests to assert against expected outputs from varying scenarios. This figure demonstrates a use case not related to the development of this project however, it does demonstrate the similar structure to a unit test. This is shown by the test being split up into a method being called, and an expected result.



```
BOOST_AUTO_TEST_CASE( fourCharacterInput )
{
    metres expectedResult = 14951.7;
    Route route = Route(LogFiles::GPXRoutesDir + "/N0809433/ABCD.gpx", isFileName);
    BOOST_CHECK_CLOSE(route.totalLength(), expectedResult, tol);
}
```

Figure 17. C++ Automated Unit Testing

In the above case the "route" variable is set equal to the resulting output of the called method, and the expected result, combined with the total length and "tol" variables to determine if the test passes or fails.

In my personal experience, I found the automated testing libraries for both languages to be logical and easy to understand and follow. I did however prefer the library provided by Haskell, as it was generally easier to produce successful results. Although, it is worth noting, this could very well be due to the benefits of the functional programming language Haskell and developing pure coded methods, compared with the difficulty of a C++ project I struggled to understand when it was done.

3.4 Future Development

After using Haskell, I not only understand the use of functional programming, pure programming, and pattern matching, and the benefits of each, I feel as though it has a use I do not personally believe I will have much interaction with within my career. I have found more enjoyment programming with languages requiring more specific operations and methods such as C++ and JavaScript.

However, the TDD process and pure programming are concepts and practices I will be applying to my future development projects, specifically because they are concepts that can be applied to most programming languages and development teams. I also feel it provides with me with a skill set and knowledge that makes me a more professional developer.

Bibliography

- Kedrigern. GitHub Gist. 2013. Implementation of binary search tree in Haskell. [online] Available at: <https://gist.github.com/Kedrigern/1239141/5ee8f5f45facdf4f48785fc92a78ad4104f16537> [Accessed 20 April 2022].

- Hsinewu. GitHub Gist. 2016. Naive (unbalanced) binary search tree in haskell. [online] Available at: <https://gist.github.com/hsinewu/a141a401ec704eeaaa0f23489fd4ec01> [Accessed 23 April 2022].

References

- Docs.google.com. 2002. Test Driven Development By Example - Kent Beck.pdf. [online] Available at: <https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnx0ZXN0MTIzNHNpbTQ2NXxneDpiYTJmYWIwYTAyOGJiZmQ> [Accessed 24 April 2022].ISBN: 978-0321146533

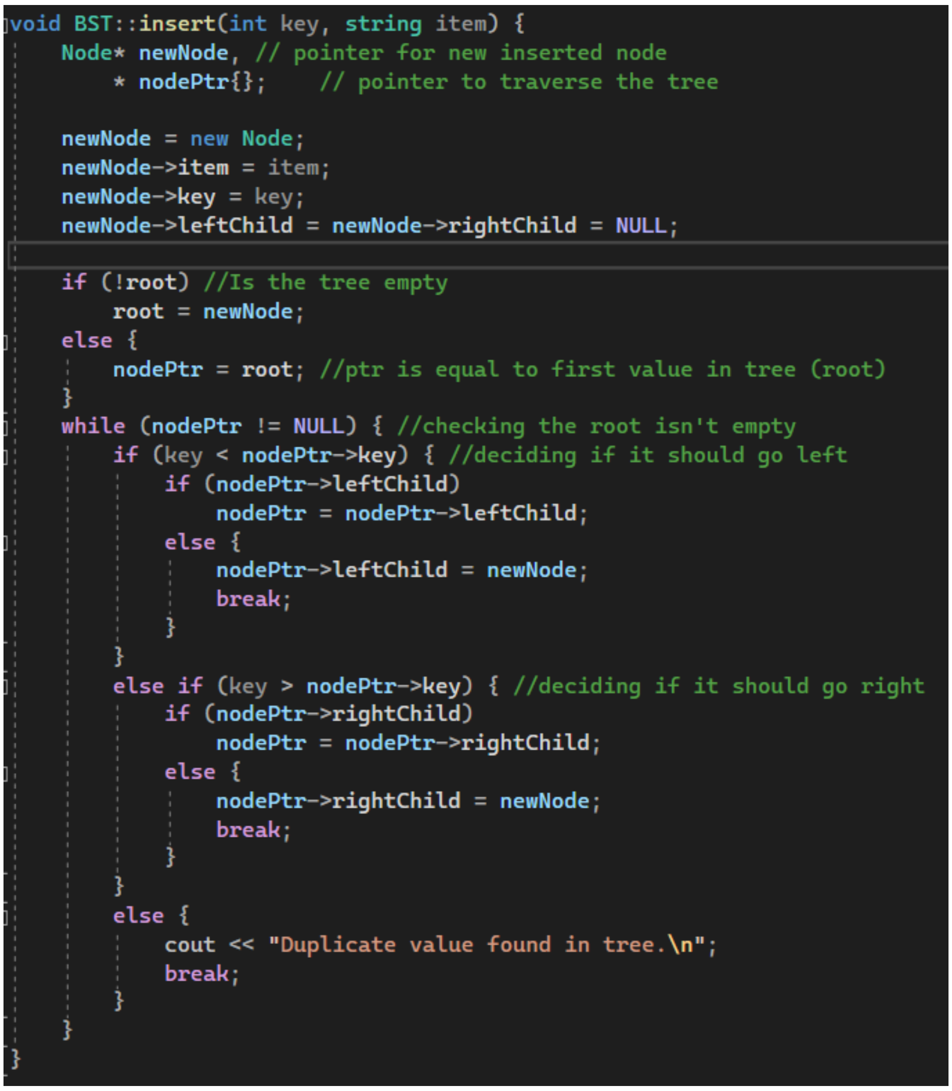
- Jeffries, R. and Melnik, G., 2007. Guest Editors' Introduction: TDD--The Art of Fearless Programming. IEEE Software, 24(3), pp.24-30.

- Maximilien, E. and Williams, L., 2003. Assessing test-driven development at IBM. [online] Ieeexplore.ieee.org. Available at: <https://ieeexplore.ieee.org/abstract/document/1201238> [Accessed 30 April 2022]. pp. 564-569, doi: 10.1109/ICSE.2003.1201238.

- Böckeler, B. and Siessegger, N., 2020. On Pair Programming. [online] martinfowler.com. Available at: <https://martinfowler.com/articles/on-pair-programming.html> [Accessed 29 April 2022].

- Tyson, M., 2021. What is functional programming? A practical guide. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3613715/what-is-functional-programming-a-practical-guide.html> [Accessed 27 April 2022].

Appendix A – C++ BST Methods



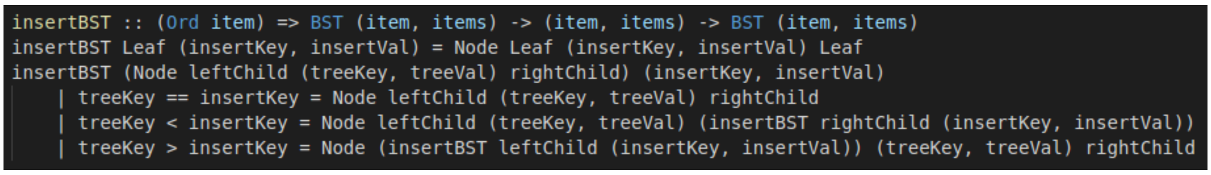
```cpp
void BST::insert(int key, string item) {
    Node* newNode, // pointer for new inserted node
        * nodePtr{};    // pointer to traverse the tree

    newNode = new Node;
    newNode->item = item;
    newNode->key = key;
    newNode->leftChild = newNode->rightChild = NULL;

    if (!root) //Is the tree empty
        root = newNode;
    else {
        nodePtr = root; //ptr is equal to first value in tree (root)
    }
    while (nodePtr != NULL) { //checking the root isn't empty
        if (key < nodePtr->key) { //deciding if it should go left
            if (nodePtr->leftChild)
                nodePtr = nodePtr->leftChild;
            else {
                nodePtr->leftChild = newNode;
                break;
            }
        }
        else if (key > nodePtr->key) { //deciding if it should go right
            if (nodePtr->rightChild)
                nodePtr = nodePtr->rightChild;
            else {
                nodePtr->rightChild = newNode;
                break;
            }
        }
        else {
            cout << "Duplicate value found in tree.\n";
            break;
        }
    }
}
```

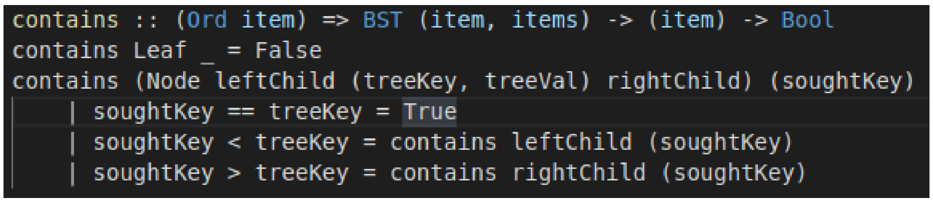
Figure 18. C++ BST Insertion Method

Appendix B – Haskell BST Methods



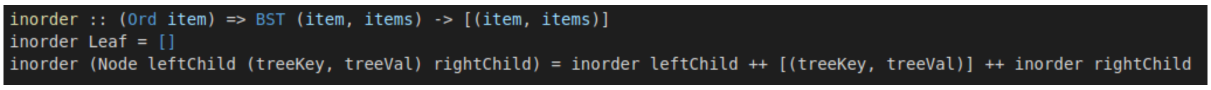
```haskell
insertBST :: (Ord item) => BST (item, items) -> (item, items) -> BST (item, items)
insertBST Leaf (insertKey, insertVal) = Node Leaf (insertKey, insertVal) Leaf
insertBST (Node leftChild (treeKey, treeVal) rightChild) (insertKey, insertVal)
    | treeKey == insertKey = Node leftChild (treeKey, treeVal) rightChild
    | treeKey < insertKey = Node leftChild (treeKey, treeVal) (insertBST rightChild (insertKey, insertVal))
    | treeKey > insertKey = Node (insertBST leftChild (insertKey, insertVal)) (treeKey, treeVal) rightChild
```

Figure 19. Haskell BST Insertion



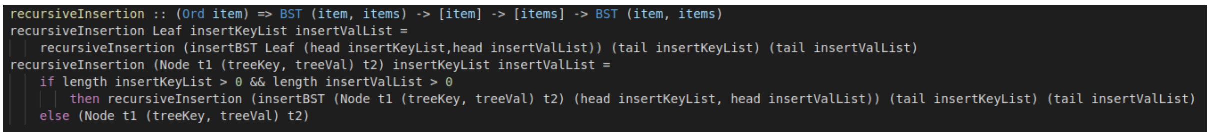
```haskell
contains :: (Ord item) => BST (item, items) -> (item) -> Bool
contains Leaf _ = False
contains (Node leftChild (treeKey, treeVal) rightChild) (soughtKey)
    | soughtKey == treeKey = True
    | soughtKey < treeKey = contains leftChild (soughtKey)
    | soughtKey > treeKey = contains rightChild (soughtKey)
```

Figure 20. Haskell BST Search



```haskell
inorder :: (Ord item) => BST (item, items) -> [(item, items)]
inorder Leaf = []
inorder (Node leftChild (treeKey, treeVal) rightChild) = inorder leftChild ++ [(treeKey, treeVal)] ++ inorder rightChild
```

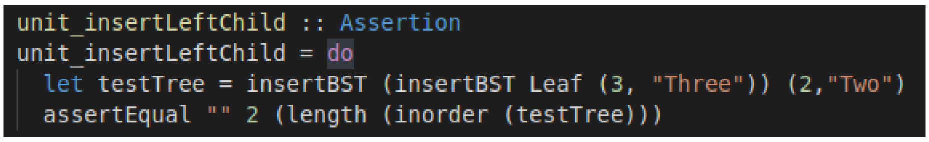
Figure 21. Haskell BST In-order Traversal



```haskell
recursiveInsertion :: (Ord item) => BST (item, items) -> [item] -> [items] -> BST (item, items)
recursiveInsertion Leaf insertKeyList insertValList =
    recursiveInsertion (insertBST Leaf (head insertKeyList,head insertValList)) (tail insertKeyList) (tail insertValList)
recursiveInsertion (Node t1 (treeKey, treeVal) t2) insertKeyList insertValList =
    if length insertKeyList > 0 && length insertValList > 0
        then recursiveInsertion (insertBST (Node t1 (treeKey, treeVal) t2) (head insertKeyList, head insertValList)) (tail insertKeyList) (tail insertValList)
    else (Node t1 (treeKey, treeVal) t2)
```

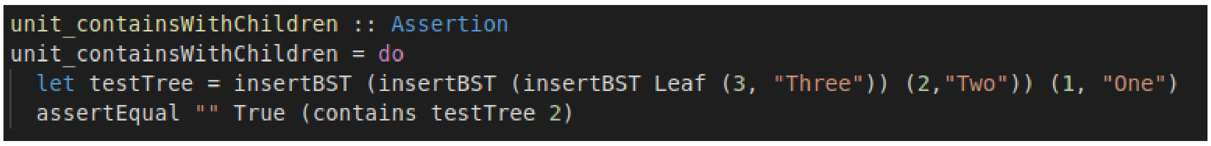
Figure 22. Haskell Recursion Method

Appendix C – Haskell Test Cases



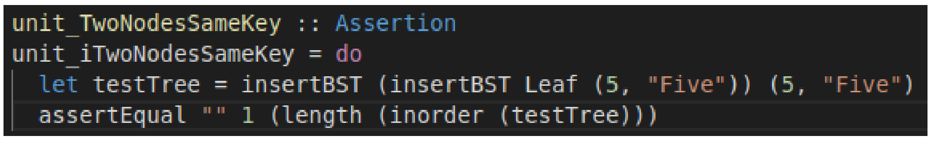
```haskell
unit_insertLeftChild :: Assertion
unit_insertLeftChild = do
  let testTree = insertBST (insertBST Leaf (3, "Three")) (2,"Two")
  assertEqual "" 2 (length (inorder (testTree)))
```

Figure 23. Haskell Insert Multiple Nodes Unit Test



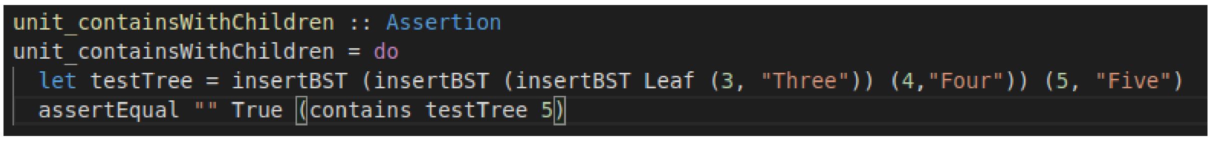
```haskell
unit_containsWithChildren :: Assertion
unit_containsWithChildren = do
  let testTree = insertBST (insertBST (insertBST Leaf (3, "Three")) (2,"Two")) (1, "One")
  assertEqual "" True (contains testTree 2)
```

Figure 24. Haskell Contains Node With Child Unit Test



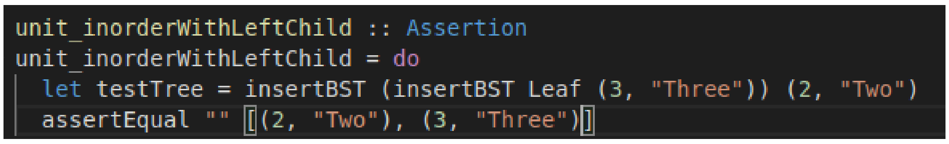
```haskell
unit_TwoNodesSameKey :: Assertion
unit_iTwoNodesSameKey = do
  let testTree = insertBST (insertBST Leaf (5, "Five")) (5, "Five")
  assertEqual "" 1 (length (inorder (testTree)))
```

Figure 25. Haskell Inserting Two Nodes with the same Keys



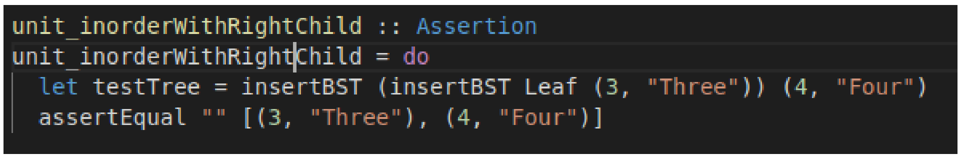
```haskell
unit_containsWithChildren :: Assertion
unit_containsWithChildren = do
  let testTree = insertBST (insertBST (insertBST Leaf (3, "Three")) (4,"Four")) (5, "Five")
  assertEqual "" True (contains testTree 5)
```

Figure 26. Haskell Contains With Two Right Nodes Unit Test



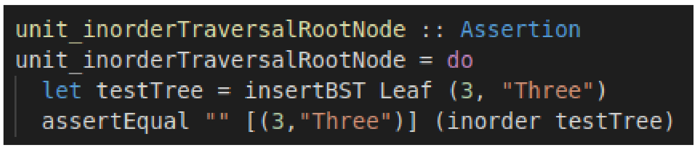
```haskell
unit_inorderWithLeftChild :: Assertion
unit_inorderWithLeftChild = do
  let testTree = insertBST (insertBST Leaf (3, "Three")) (2, "Two")
  assertEqual "" [(2, "Two"), (3, "Three")]
```

Figure 27. Haskell In-order Left Child Unit Test



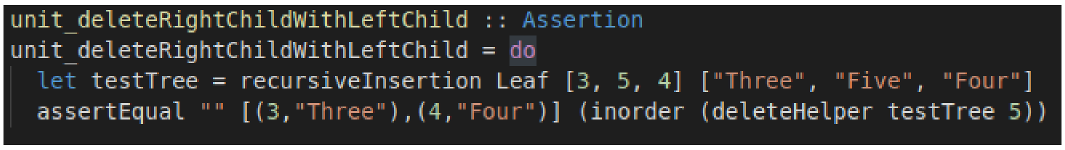
```haskell
unit_inorderWithRightChild :: Assertion
unit_inorderWithRightChild = do
  let testTree = insertBST (insertBST Leaf (3, "Three")) (4, "Four")
  assertEqual "" [(3, "Three"), (4, "Four")]
```

Figure 28. Haskell In-order with Right Child



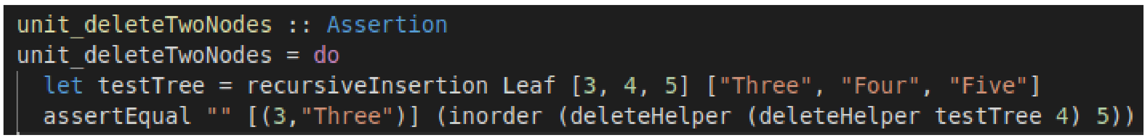
```haskell
unit_inorderTraversalRootNode :: Assertion
unit_inorderTraversalRootNode = do
  let testTree = insertBST Leaf (3, "Three")
  assertEqual "" [(3,"Three")] (inorder testTree)
```

Figure 29. Haskell In-order with Only Root Node



```haskell
unit_deleteRightChildWithLeftChild :: Assertion
unit_deleteRightChildWithLeftChild = do
  let testTree = recursiveInsertion Leaf [3, 5, 4] ["Three", "Five", "Four"]
  assertEqual "" [(3,"Three"),(4,"Four")] (inorder (deleteHelper testTree 5))
```

Figure 30. Haskell Removal of Right Child with a Left Child



```haskell
unit_deleteTwoNodes :: Assertion
unit_deleteTwoNodes = do
  let testTree = recursiveInsertion Leaf [3, 4, 5] ["Three", "Four", "Five"]
  assertEqual "" [(3,"Three")] (inorder (deleteHelper (deleteHelper testTree 4) 5))
```

Figure 31. Haskell Delete Two Nodes at Once Unit Test