Case Study of Test-Driven Development by John Hills N0923065

Task 1: What is the practice of TDD (Test Driven Development)

Test Driven Development is a development process in which the developer creates tests for their code alongside the code itself and uses those tests to ensure that the code they've developed is bug-free and as a result, there are always tests for the different sections of the code. The steps of the cycle are:

1. Identify the functionality.
2. Write the test code.
3. Write the functional code.
4. Run the tests If the tests fail, debug the code and retry the tests.
5. If the tests pass, refactor the code if possible and re-run the tests, while fixing any potential issues that arise during refactoring.
6. If there are no more issues with the functionality, move onto a new one.

The development process is very strict. For the process to be properly carried out, the steps must be followed in that exact order, which means that things like developing the functional code before the test code, or refactoring the code before it is fully tested cannot happen in this development process. One aspect of TDD is that in using the process, the code that is being developed is supposed to evolve over time, which means that the developer is not allowed to develop the entire code structure in advance. The code structure is not the same as the structure of the functionality, meaning that the developer is allowed to structure the different systems of what they are developing, but they are not allowed to structure the individual functions of the systems and how those functions interact in advance.

The process being about the code evolving over time also means that looking for refactoring opportunities is a mandatory part of the process since refactoring is the part of the process where the most changes to the original code design happen. In addition, the developer is allowed revisit previous sections of the code if TDD has already been applied to that section. This means that if a developer figures out a way to refactor previously developed code, they are allowed to go back and refactor it assuming that the refactored code passes the tests that were created for it.

Something that is very important to remember in test-driven development is that the developer is allowed to create extra functionality that is designed to aid in testing the functionality. So, for example, if a sorting algorithm is being tested, then it’s okay for the developer to program a function that checks whether or not something has been sorted. However, creating a piece of functionality that is not required to make the test pass is strictly forbidden in test-driven development as there is no tests for that functionality.

In test-driven development, there are 2 types of testing that are mandatory to the process, which are unit testing and automated testing. Unit testing is the process of isolating sections of your code and testing them, and automated testing is when multiple tests can be run on a section of code without needing the developer’s interaction.

Both forms of testing are much better suited to be done with pure code than impure code since with pure code, you only need to test with data as specified by the input parameters of the section of code you’re testing. That doesn’t mean that it’s impossible to do test-driven development on impure code because there are definitely ways of getting around the impure parts of the code, but it is definitely more difficult.

Due to the time that is needed to write the tests and how thorough those tests need to be in order to make sure that the functional code is bug-free, test-drive development tends to take more time than other development processes. What it loses in terms of time efficiency, it gains in the fact that it makes regression testing much easier, since the tests for the functionality are written before the code is actually programmed, and it also greater guarantees that the code developed is bug-free.

Task 2a: Developing the binary search tree.

For this coursework, the binary search tree I developed had functions that could:

- Create an empty tree.
- Lookup an entry.
- Insert an entry.
- Produce an ordered list of the entries.
- Remove an entry based on a key.
- Remove entries based on a higher order condition.

And I also developed a dictionary class that inherited the tree code, and I made the tree code able to use a key and item of any data type, although the parametric polymorphism was something that I implemented into the tree after the previous functions were developed.

Task 2b: Summarise development and describe 3 examples of the TDD process.

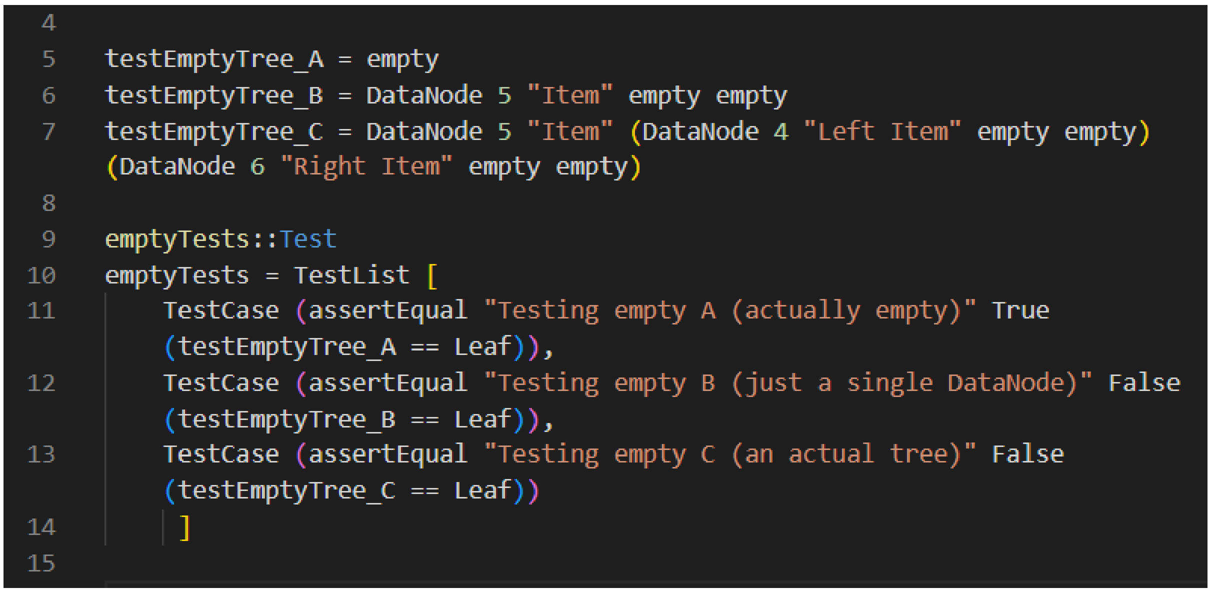
For the most part, development went perfectly well. Haskell's testing suite offered the tools needed to test the code, and visual studio code's terminal allowing me to use GHCI to debug the code and run specific functions was extremely useful. One issue that cropped up during development was that in order for the test code to be properly debugged before moving onto the functional code, there needed to be a function definition in the source code file. In other languages like Java and C++ this isn't much of a problem because you'd just provide a simple definition for the function with the appropriate inputs but no code inside until the test code has been checked, which means that ultimately, the functional code can still be developed after the test code. But in Haskell, any function definition needs some basic code to go with it that outputs the relevant data type.

This sounds like it would be breaking the TDD process since you'd have to write some functional code in order to properly debug the test code and move onto writing the functional code. However, if there is a very simple piece of place holder code that outputs a relevant data type, but that doesn't represent the actual functional code, then this doesn't break the TDD process because the actual functional code won't actually be worked on before the test code is finished. So as an example, if a function was being developed that outputted an integer value, it would be perfectly fine to create a single line of code that accepted the relevant inputs and outputted 0 since this is unlikely to represent the actual functional code.

In order to show the aspects and the troubles of the development process I have decided to show the process of developing the empty and removeIf functions, as well as the dictionary class.

Developing the empty tree function

I decided to develop this part first because it was the most basic part of the tree and the other basic parts of the tree required different parts of the tree to be built in order to make the testing work, like how inserting a node would require a tree to insert onto and the ability to search the inserted tree for the new node to see if it worked, and finding nodes required a tree that contained nodes to find. For the tests, 3 trees were manually created and then they were tested to see if they were equal to the Leaf constructor, which is the constructor for an empty node. The original test code is displayed below.



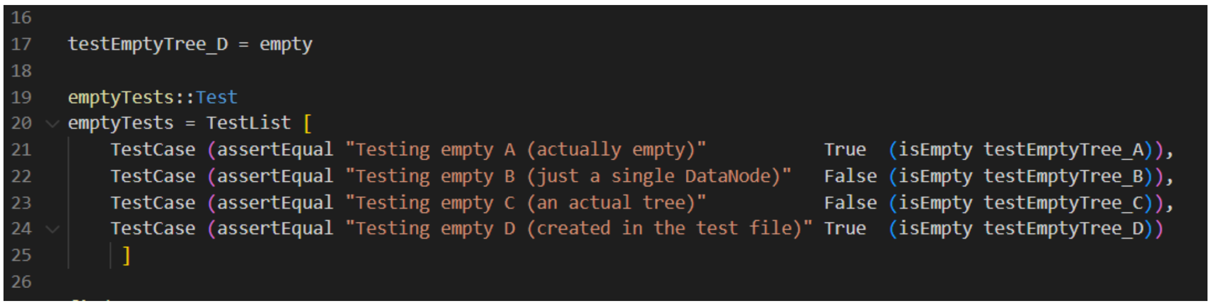
```haskell

testEmptyTree_A = empty
testEmptyTree_B = DataNode 5 "Item" empty empty
testEmptyTree_C = DataNode 5 "Item" (DataNode 4 "Left Item" empty empty) (DataNode 6 "Right Item" empty empty)

emptyTests::Test
emptyTests = TestList [
    TestCase (assertEqual "Testing empty A (actually empty)" True (testEmptyTree_A == Leaf)),
    TestCase (assertEqual "Testing empty B (just a single DataNode)" False (testEmptyTree_B == Leaf)),
    TestCase (assertEqual "Testing empty C (an actual tree)" False (testEmptyTree_C == Leaf))
     ]

```

Immediately there was a problem when the test code was compiled, which was that the DataNode and Leaf constructors could not be exported to the test file from the source code file. This led to 2 issues, the first being that the test trees could not be created in the test file, and the second being that the test trees were going to need a different method for testing if they were empty or not.

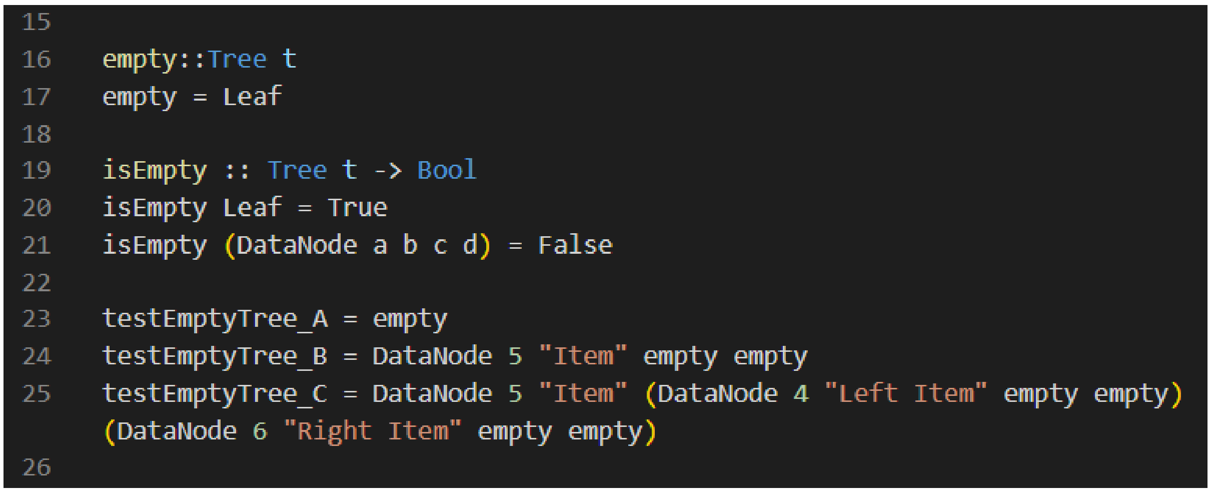
This issue was solved by simply creating the test trees in the source code file and exporting them from there. While this does mean that the binary search tree code file will always be exporting 3 random trees along with its functional code, which it shouldn't be doing, this isn't much of a big problem because those trees don't interact with anything other than the test code and can therefore, be simply be omitted at the final steps of development. The second issue was solved by creating a function in the source code file that checked whether or not a tree was empty or not. It used pattern matching to output true if the inputted tree was equal to the Leaf constructor and false if it was equal to the DataNode constructor. While this new code had no way to be tested itself, I was confident that it would work perfectly in the test due to its simplicity.

However, there was one more problem with the empty function, which was that for the tests to properly compile and be checked to see if they'd work, there needed to be a basic definition of the empty function. However, the issue is that the placeholder code that should be used for the empty function is quite literally the exact same as what the actual functional code would be, which is a function definition with no input and an output of the Leaf constructor. This means that, contrary to the TDD process, the functional code of this section would have to be written before the test code could be fully debugged. While this does not break the TDD process that much, since the test code was already written before the definition for the empty function was given, not being able to properly debug the test code before moving onto the functional code feels like it is breaking the TDD process framework, which shows that there are certain situations where following the framework exactly is very difficult, or even impossible.

After writing out the definition of the function, I also decided to add another test where I create a test tree inside the test file using the empty function to check whether or not the empty function would work in a different code file than the source code file.



```haskell

testEmptyTree_D = empty

emptyTests::Test
emptyTests = TestList [
    TestCase (assertEqual "Testing empty A (actually empty)"          True  (isEmpty testEmptyTree_A)),
    TestCase (assertEqual "Testing empty B (just a single DataNode)"  False (isEmpty testEmptyTree_B)),
    TestCase (assertEqual "Testing empty C (an actual tree)"          False (isEmpty testEmptyTree_C)),
    TestCase (assertEqual "Testing empty D (created in the test file)" True  (isEmpty testEmptyTree_D))
    ]

```



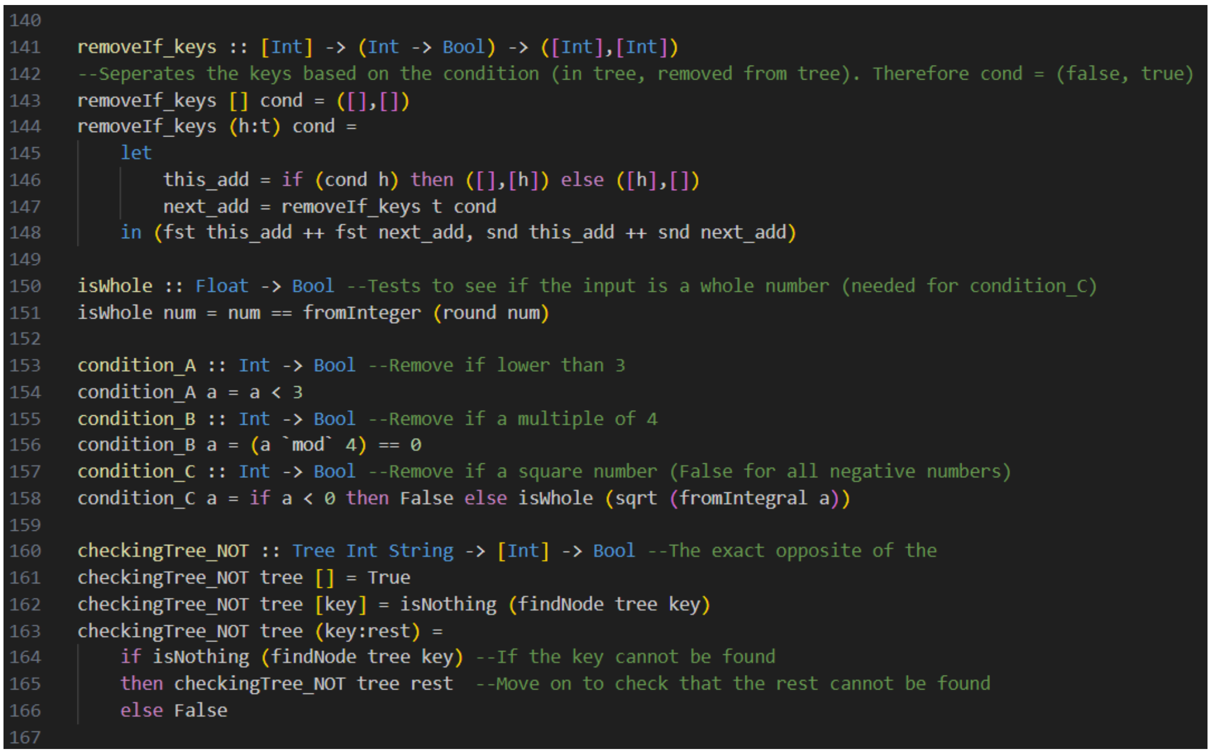
```haskell

empty::Tree t
empty = Leaf

isEmpty :: Tree t -> Bool
isEmpty Leaf = True
isEmpty (DataNode a b c d) = False

testEmptyTree_A = empty
testEmptyTree_B = DataNode 5 "Item" empty empty
testEmptyTree_C = DataNode 5 "Item" (DataNode 4 "Left Item" empty empty)
(DataNode 6 "Right Item" empty empty)

```

Developing the removeIf function

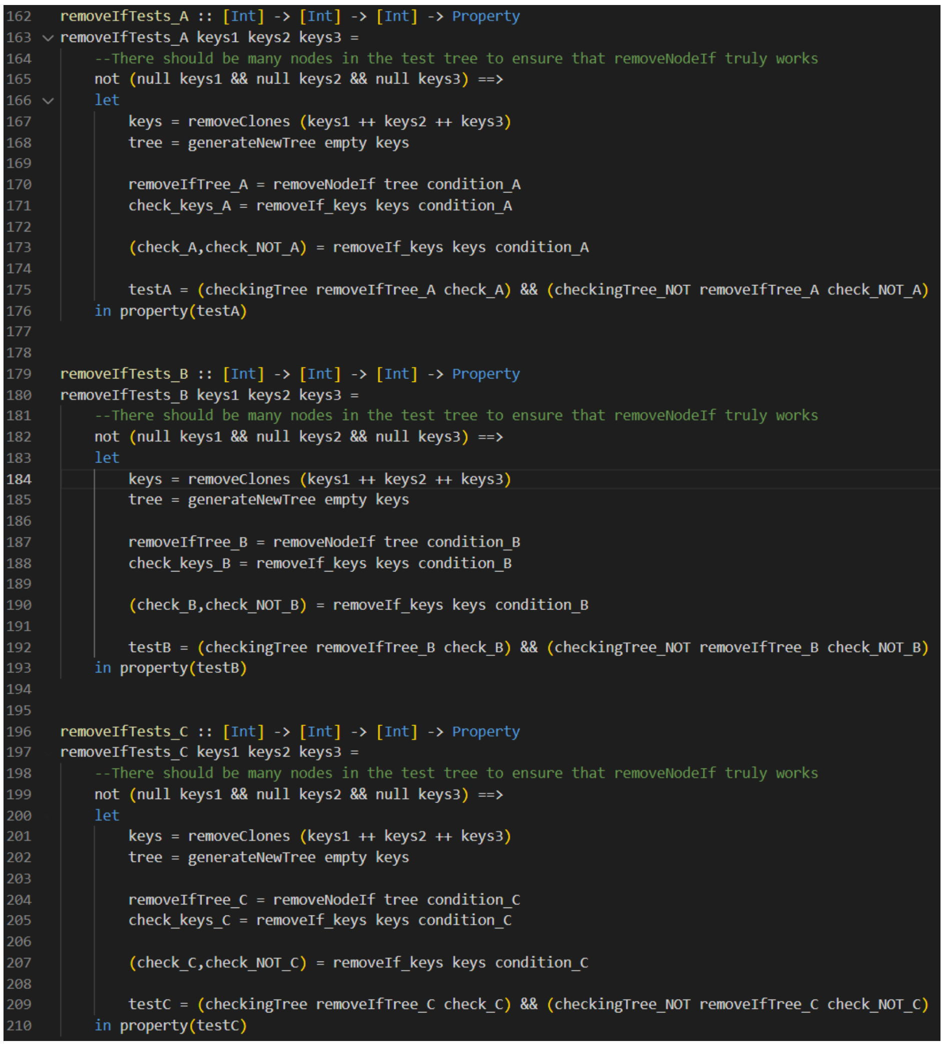
For the removeIf function I decided that I wanted to use automated testing. The tests would generate a random tree from 3 randomly generated lists of integers and then remove entries from the tree based on 3 conditions. The list of keys would be separated into a list of the keys that followed the condition and the rest would be put into a different list. Then the test code would check if all of the valid keys were in the tree and if all of the invalid keys were not in the tree. I decided to use 3 different lists of keys instead of one because I knew that it was likely a very large number of the nodes in the tree would be removed and in order to properly stress test the code, I wanted a very large number of nodes for the code to remove.

Then I designed the removeIf function which would go through every node of the using post-order traversal and if the current key followed the condition, the node would be removed with the removeNode function that had already been developed before this. Initially the tests failed, but after fixing some of the functions that were designed to aid in the testing, the tests all passed.

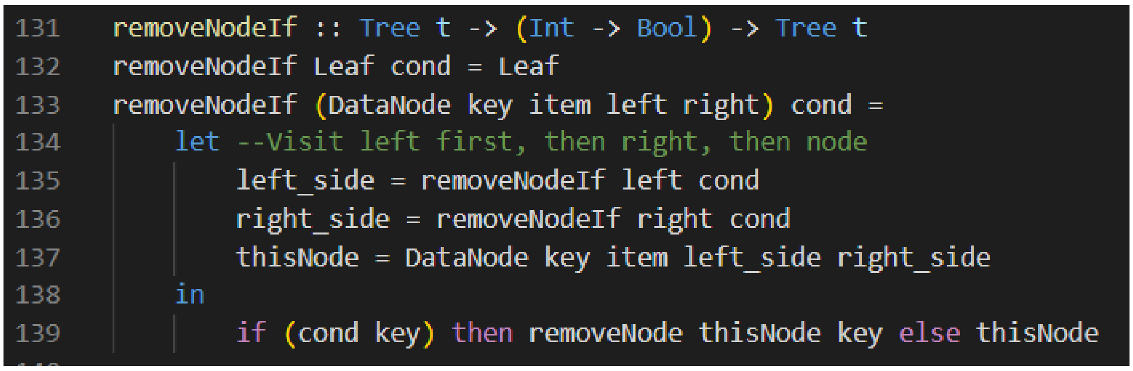
The extra code that was created specifically for the removeIf function tests, the actual test code, and the functional code is displayed respectively below.



```haskell

removeIf_keys :: [Int] -> (Int -> Bool) -> ([Int],[Int])
--Seperates the keys based on the condition (in tree, removed from tree). Therefore cond = (false, true)
removeIf_keys [] cond = ([],[])
removeIf_keys (h:t) cond =
    let
        this_add = if (cond h) then ([],[h]) else ([h],[])
        next_add = removeIf_keys t cond
    in (fst this_add ++ fst next_add, snd this_add ++ snd next_add)

isWhole :: Float -> Bool --Tests to see if the input is a whole number (needed for condition_C)
isWhole num = num == fromInteger (round num)

condition_A :: Int -> Bool --Remove if lower than 3
condition_A a = a < 3
condition_B :: Int -> Bool --Remove if a multiple of 4
condition_B a = (a `mod` 4) == 0
condition_C :: Int -> Bool --Remove if a square number (False for all negative numbers)
condition_C a = if a < 0 then False else isWhole (sqrt (fromIntegral a))

checkingTree_NOT :: Tree Int String -> [Int] -> Bool --The exact opposite of the
checkingTree_NOT tree [] = True
checkingTree_NOT tree [key] = isNothing (findNode tree key)
checkingTree_NOT tree (key:rest) =
    if isNothing (findNode tree key) --If the key cannot be found
    then checkingTree_NOT tree rest  --Move on to check that the rest cannot be found
    else False

```



```haskell
removeIfTests_A :: [Int] -> [Int] -> [Int] -> Property
removeIfTests_A keys1 keys2 keys3 =
    --There should be many nodes in the test tree to ensure that removeNodeIf truly works
    not (null keys1 && null keys2 && null keys3) ==>
    let
        keys = removeClones (keys1 ++ keys2 ++ keys3)
        tree = generateNewTree empty keys

        removeIfTree_A = removeNodeIf tree condition_A
        check_keys_A = removeIf_keys keys condition_A

        (check_A,check_NOT_A) = removeIf_keys keys condition_A

        testA = (checkingTree removeIfTree_A check_A) && (checkingTree_NOT removeIfTree_A check_NOT_A)
    in property(testA)


removeIfTests_B :: [Int] -> [Int] -> [Int] -> Property
removeIfTests_B keys1 keys2 keys3 =
    --There should be many nodes in the test tree to ensure that removeNodeIf truly works
    not (null keys1 && null keys2 && null keys3) ==>
    let
        keys = removeClones (keys1 ++ keys2 ++ keys3)
        tree = generateNewTree empty keys

        removeIfTree_B = removeNodeIf tree condition_B
        check_keys_B = removeIf_keys keys condition_B

        (check_B,check_NOT_B) = removeIf_keys keys condition_B

        testB = (checkingTree removeIfTree_B check_B) && (checkingTree_NOT removeIfTree_B check_NOT_B)
    in property(testB)


removeIfTests_C :: [Int] -> [Int] -> [Int] -> Property
removeIfTests_C keys1 keys2 keys3 =
    --There should be many nodes in the test tree to ensure that removeNodeIf truly works
    not (null keys1 && null keys2 && null keys3) ==>
    let
        keys = removeClones (keys1 ++ keys2 ++ keys3)
        tree = generateNewTree empty keys

        removeIfTree_C = removeNodeIf tree condition_C
        check_keys_C = removeIf_keys keys condition_C

        (check_C,check_NOT_C) = removeIf_keys keys condition_C

        testC = (checkingTree removeIfTree_C check_C) && (checkingTree_NOT removeIfTree_C check_NOT_C)
    in property(testC)
```



```haskell
removeNodeIf :: Tree t -> (Int -> Bool) -> Tree t
removeNodeIf Leaf cond = Leaf
removeNodeIf (DataNode key item left right) cond =
    let --Visit left first, then right, then node
        left_side = removeNodeIf left cond
        right_side = removeNodeIf right cond
        thisNode = DataNode key item left_side right_side
    in
        if (cond key) then removeNode thisNode key else thisNode
```

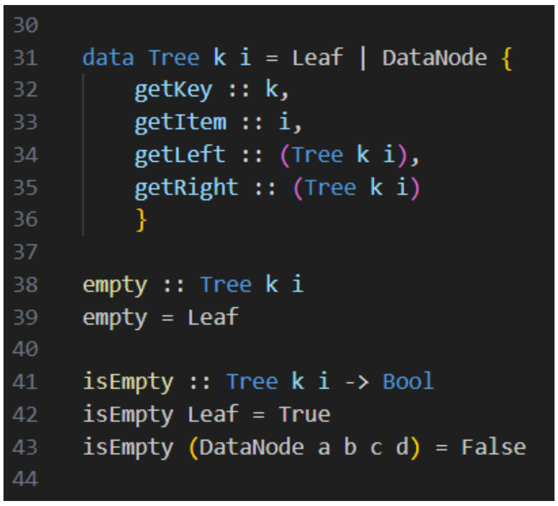
Something I noticed while creating the test code was how little control I had over the randomly generated lists of integers, because I couldn’t control the range of the integers that were generated or the range of sizes of the lists, which is why I used 3 randomly generated lists.

But there was another issue with the tests, which was that I had included a line in the test code where if any of the randomly generated lists were empty, the test would be discarded, and another set of lists would be generated for testing. When the tests were run, roughly between 30 and 40 of the tests were discarded because at least one of the lists was empty. If the lists were being generated to avoid repeated lists, then a maximum of 3 tests would have been discarded, but the number of discarded tests tells me that the generated lists are not only not ignoring repeat randoms, but they are likely to repeat many of the same lists during the random generation.

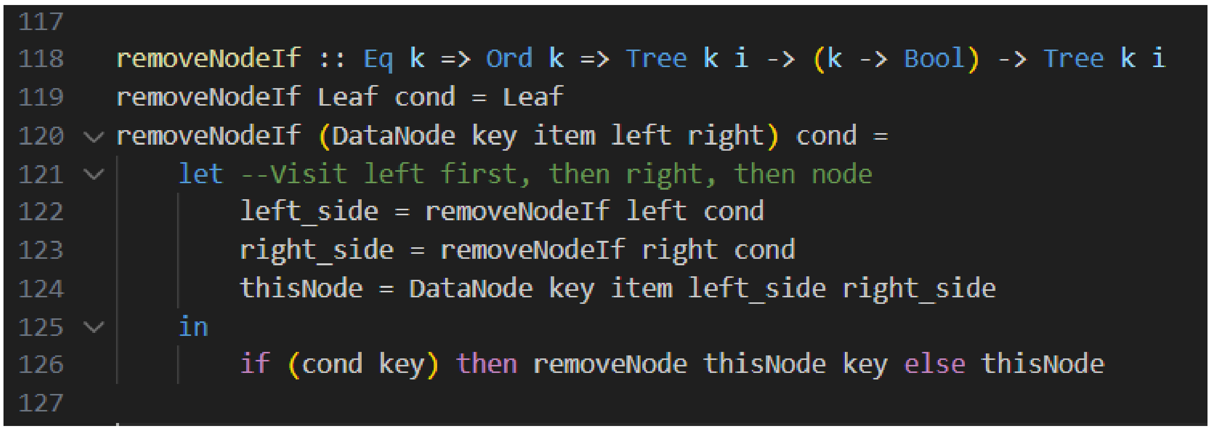
Developing the dictionary class

The first part of the dictionary class was to make sure that the binary search tree could work with keys and items of any data type. As you can see in the code screenshots in this report, while the items could be any data type, the keys were hard coded to have an integer key only. In order to fix this, I had to go back to all of the tree test code and tree functional code and refactor it to allow the key to be any data type. However, the only types that should be allowed were ones that you can use the ==, < and > operators on since those operators were used in the tree functions. The solution to this was to use type classes on the tree functions in order to specify that the key needs to be of an appropriate type.

Since the TDD process allows going back to previously developer code for the purposes of refactoring, doing this did not break the framework. The images below show the refactored tree declaration, empty function, and removeIf function.



```haskell

data Tree k i = Leaf | DataNode {
    getKey :: k,
    getItem :: i,
    getLeft :: (Tree k i),
    getRight :: (Tree k i)
    }

empty :: Tree k i
empty = Leaf

isEmpty :: Tree k i -> Bool
isEmpty Leaf = True
isEmpty (DataNode a b c d) = False

```



```haskell

removeNodeIf :: Eq k => Ord k => Tree k i -> (k -> Bool) -> Tree k i
removeNodeIf Leaf cond = Leaf
removeNodeIf (DataNode key item left right) cond =
    let --Visit left first, then right, then node
        left_side = removeNodeIf left cond
        right_side = removeNodeIf right cond
        thisNode = DataNode key item left_side right_side
    in
        if (cond key) then removeNode thisNode key else thisNode

```

Developing the next part of the code presented a unique issue in that, in reading the coursework specification, I had misunderstood the actual functionality of the dictionary class. I originally thought that this was supposed to be a singly linked list that inherited the binary search tree, when instead it is supposed to be a general dictionary class that inherits the binary search tree and makes use of the methods for the tree. However, by the time I had discovered this, I had already written half of the methods of this class and the tests that go with them.

I decided that the only way to fix this issue while keeping to the TDD process was to stop working on all of the current code and start another from the beginning on the proper test code and functional code. However, a few bits of the original functional code and a lot of the original test code could just be copied over because they worked the same for the new functionality, which means that developing the new functionality was easier than it normally would be. Although, if I had misunderstood the functionality more than I actually had, then this would have been a much more difficult task.

After implementing these fixes and running the tests, not only did the new dictionary tests run without issue, but the original tree test code also ran without issue with the newly refactored code.

Task 3: Critically evaluate TDD

One thing that is true for test-driven development is that, assuming the tests that are written by the developer are good enough to detect the bugs in a piece of code, then that means that by the end of developing each section of code, the code should not need any debugging that requires the developer to go back through large clumps of code. That process can take a lot of time, which means that test-driven development can potentially save time for the developer in terms of bug testing.

However, this is only assuming that the tests that are written by the developer are thorough enough to find all of the potential issues with a program. If the developer writes insufficient test code that misses a major error, they are most likely not going to discover this issue until much later in development and by that point, they will have to do a lot of work going backwards through their code to find the issue.

And on the opposite side of that, there is also the issue that the developer might waste time writing unneeded tests for the code. In larger projects, this is more likely to happen, which means that a lot of development time could be wasted on the test code, which is a big issue for larger projects.

There are also some types of coding projects where using test-driven development is next to impossible because of the nature of that type of project. A prime example of this that I have personal experience in is video game development. In game development, pretty much every major system interacts with so many other systems that it becomes extremely difficult to keep track of every single input in a system, which is essential for testing specific cases.

For example, if you were testing a movement system for the player character, you need to make sure that the controller works, that the game can place the sprite/model of the character correctly to where the position variable says it should be, and then the actual movement code would need current position and current button inputs, but also maybe things like the friction of the surface the player is on, their current velocity and acceleration, current power ups, and anything else the game might have. Trying to properly test this would be almost impossible not just due to the sheer number of inputs that need regulating, but also the fact that the output is a visual one, which means that it’s extremely difficult to actually test the output of the code with test code and the developer would most likely have to do it by eye.

One more thing that makes test-driven development very difficult is the fact that most languages don’t have an integrated testing suite. This project was much easier to apply test-driven development to since Haskell already has certain tools integrated to test the code. But for a language like C++, if the developer wants to test their code with a testing suite, they are going to need to create the testing suite themselves. It would be really stressful and time consuming to try and create a testing suite from scratch in a different programming language for a project, especially if that project was under a strict time limit.

<u>Task 4: Reflect on your own experiences</u>

One thing that I personally feel went well with my implementation of test-driven development is that I never felt like I had to go back through previously written code to make fix bugs. Numerous functions and tests for those functions in the binary search tree make use of functions like empty, findNode, and insertNode, and there was never an instance in the testing phase where the code from those sections caused any issues with the tests or the functional code, and this is in no small part due to the fact that those functions were tested before I moved onto other parts of the code.

However, I personally really disliked the rigidness of the process. There were many times when I started on a new section of code that I’d have an idea for how the code could be implemented, but instead of being able to work on my idea, I had to put it on pause and work on the tests first. This wasn’t just annoying because I personally like to have a lot more freedom in my development, but leaving the idea untouched while I wrote the test code meant that there was a much greater chance that I would forget the idea. However, an easy solution is that I could make a note of the idea in the functional code file and not implement it yet, which leaves a much greater chance that I will remember my idea when it comes to writing the functional code.

One thing that I personal disliked about the automated testing suite that Haskell provided was the lack of control I had over the automated values. For the removeIf tests, I wanted to not only regulate the size of the randomly generated lists, but I also wanted to avoid repeated instances, and Haskell’s property testing didn’t allow me to do either. I would much prefer to have much more control over the randomly generated lists, like being able to set a size limit, a limit for the numbers in the lists, and even the ability to avoid repeat lists, which can be avoided by using the “not” keyword in the test, but it would be much more efficient to just avoid generating another empty list in the first place.

Despite my complaints, I do see myself using the practice in my own projects. However, it does depend on if the type of project would be best suited to use the practice of test-driven development, and while other languages don’t have the tools built in, I wouldn’t be opposed to developing certain tools for the language in my own time if I had the freedom in terms of time. I really like how the test code allows me to find the bugs in my code before they get so deeply engrained in the program that they become very difficult to remove, and I like that the process encourages decomposing the functionalities of the program as well because that is a massive help for larger projects.

And as well as that, there is nothing to say that I can’t create my own process based on TDD that takes inspiration from it but is different according to what I’d prefer to do. Maybe I could use a process that allows me to properly check code with visual outputs like an image or printed text. It’s up to me what I do and how I take inspiration from TDD.