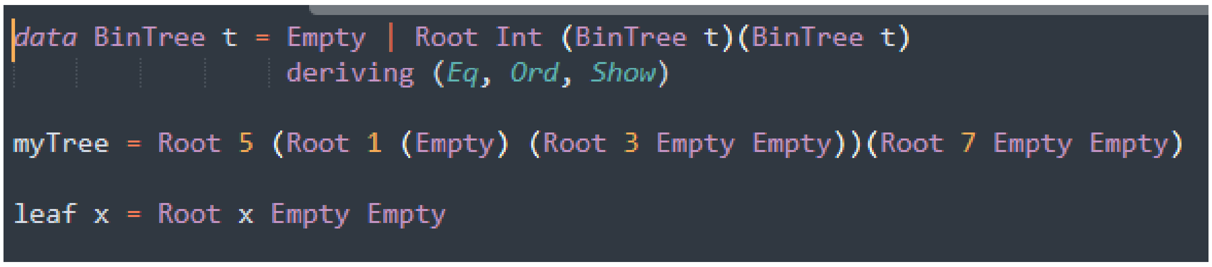
CA341 Assignment 2
Jamie Clarke
19/11/2021
18398783

## <u>Comparison of functional and Logic</u>

For this assignment, I decided to choose Haskell for my functional program and prolog for my logic. In this brief i will explain my reasoning for both as well as the advantages and disadvantages of both.  I will also walk through each of the codes, citing where necessary and showing how I got each of the answers.

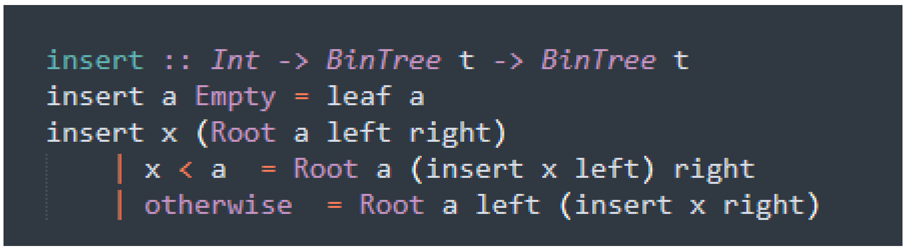
### Functional

To start with my functional programming I began by initialing the Binary tree(BinTree) data at t. I got this line of code from the lab sheet 5, trees of int.[1] It calls an Empty list and a Root integer. It also initialises myTree and a leaf x.



```haskell
data BinTree t = Empty | Root Int (BinTree t)(BinTree t)
                 deriving (Eq, Ord, Show)

myTree = Root 5 (Root 1 (Empty) (Root 3 Empty Empty))(Root 7 Empty Empty)

leaf x = Root x Empty Empty
```

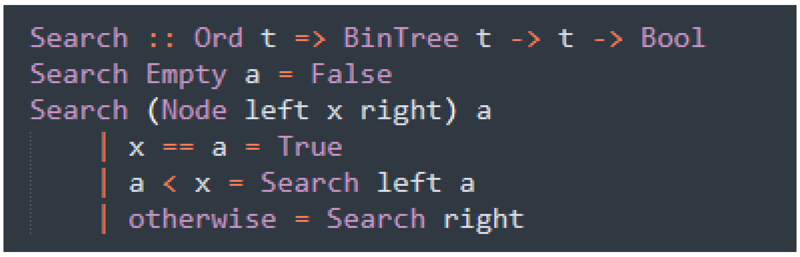
Next for this code I had to create an insert function for adding an integer x into the binary tree T. As you can see below I insert leaf a into the empty list(named Empty). I also check if leaf x is smaller than a and if so we do the following code.



```haskell
insert :: Int -> BinTree t -> BinTree t
insert a Empty = leaf a
insert x (Root a left right)
      | x < a   = Root a (insert x left) right
      | otherwise   = Root a left (insert x right)
```

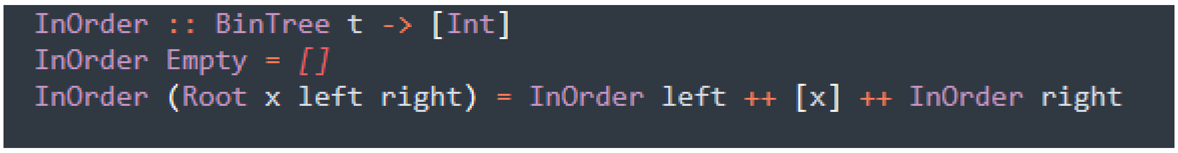
For my search function I had to use an online resource[2]. This function returns True if x is contained inside the tree. I checked both left and right nodes for x and if it was found, my function returned True else it returned False. It searches the empty list "a" to see if it is there first.

[1] https://www.computing.dcu.ie/~davids/teaching.shtml
[2] StackOverflow



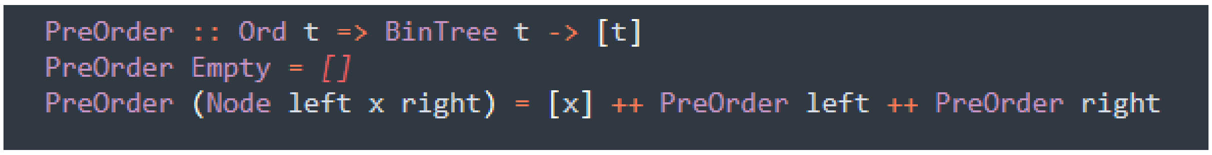
```haskell
Search :: Ord t => BinTree t -> t -> Bool
Search Empty a = False
Search (Node left x right) a
      | x == a = True
      | a < x = Search left a
      | otherwise = Search right
```

Next I had to create an InOrder  function. This had to list the nodes of the binary tree using an In order traversal. These notes were also in your lab sheet 5, trees of int. An empty list is created called "Empty". [3]



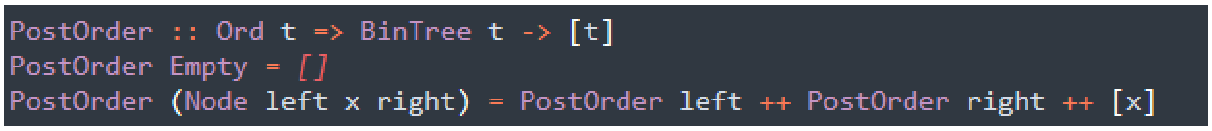
```haskell
InOrder :: BinTree t -> [Int]
InOrder Empty = []
InOrder (Root x left right) = InOrder left ++ [x] ++ InOrder right
```

For my pre order function I had to list the nodes of the binary tree using a preorder traversal. For this you visit the root or node first, followed by the left subtree and the right subtree next. I used a similar outline to the In order function, just adjusting it to suit my needs for this function. Again calling an empty list called "Empty" and just changed the bottom line of the function.



```haskell
PreOrder :: Ord t => BinTree t -> [t]
PreOrder Empty = []
PreOrder (Node left x right) = [x] ++ PreOrder left ++ PreOrder right
```

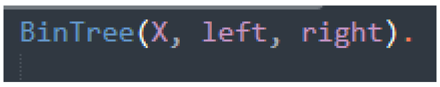
For Post Order all I did was swap the bottom lines from PreOrder, changing the position of [x] from the start of the line to the end. Apart from this the function is identical to PreOrder, calling an empty list "Empty".



```haskell
PostOrder :: Ord t => BinTree t -> [t]
PostOrder Empty = []
PostOrder (Node left x right) = PostOrder left ++ PostOrder right ++ [x]
```

**Logic**

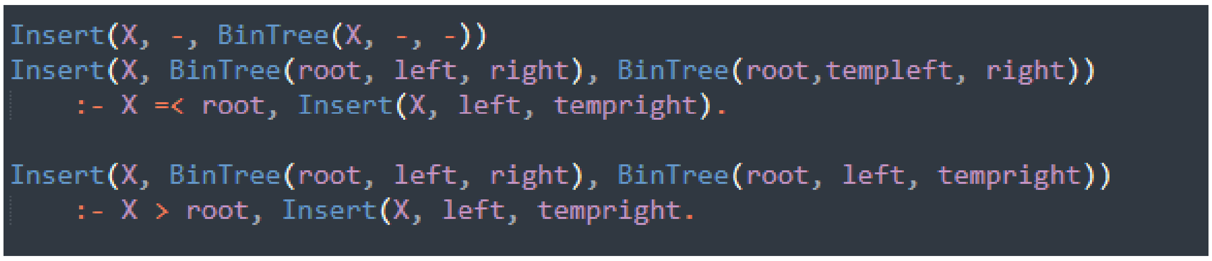
To start off in my prolog code, I created "BinTree" with left and right nodes as well as X, an integer. This was quite a simple start to the code.



```prolog
BinTree(X, left, right).
```

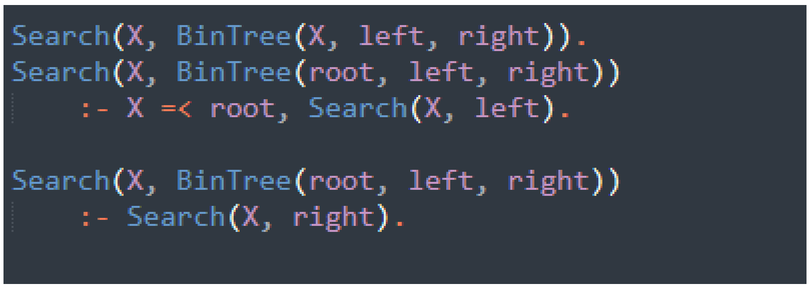
[3] https://www.computing.dcu.ie/~davids/teaching.shtml

Next I had to create an insert function to add integer X to the “BinTree”. I also created “-” which was an Empty list. I checked to see if the integer X was less than or equal to the root node and if so inserted X into the left node. If X was greater than the root node then we did the same thing. I am not sure if i did this correctly but couldn’t figure out what was going wrong.



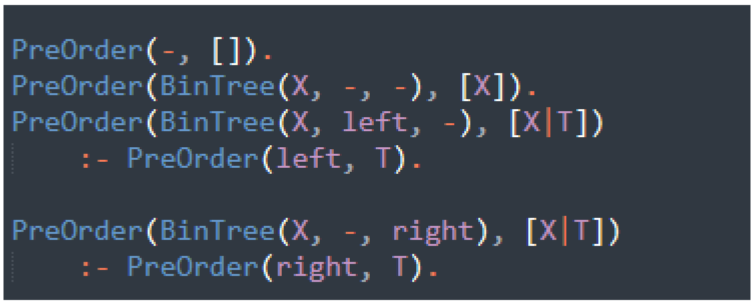
```
Insert(X, -, BinTree(X, -, -))
Insert(X, BinTree(root, left, right), BinTree(root,templeft, right))
    :- X =< root, Insert(X, left, tempright).

Insert(X, BinTree(root, left, right), BinTree(root, left, tempright))
    :- X > root, Insert(X, left, tempright.
```

Next came my search function, which had to return true if X is in “BinTree”. It does this by searching to see if X is contained in the left and right nodes of the tree. If X is less than or equal to the root node, it searches the left node else then it checks the right node. I watched a very good youtube video to help with this function.



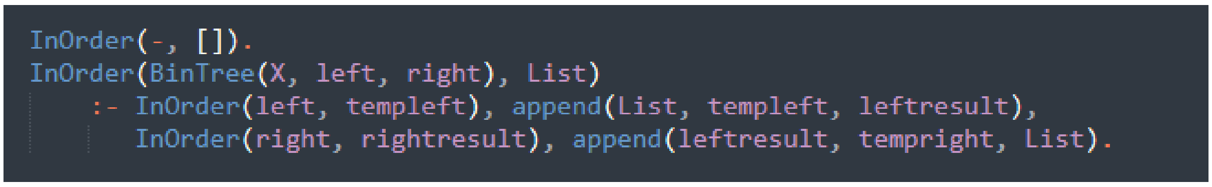
```
Search(X, BinTree(X, left, right)).
Search(X, BinTree(root, left, right))
    :- X =< root, Search(X, left).

Search(X, BinTree(root, left, right))
    :- Search(X, right).
```

For my PreOrder function I did something similar to the Insert function, calling an empty list “[]”.  For this you visit the root or node first, followed by the left subtree and the right subtree next. Unfortunately this function wasn’t working for me and I couldn't understand why.



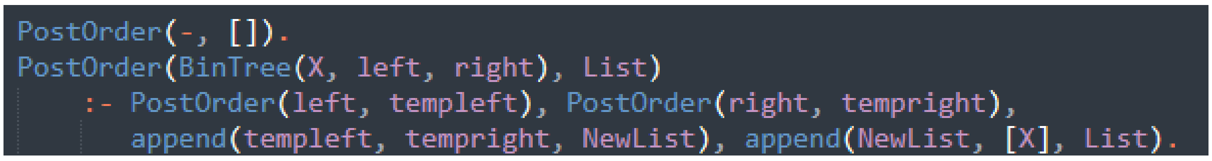
```
PreOrder(-, []).
PreOrder(BinTree(X, -, -), [X]).
PreOrder(BinTree(X, left, -), [X|T])
    :- PreOrder(left, T).

PreOrder(BinTree(X, -, right), [X|T])
    :- PreOrder(right, T).
```

For my inorder function I used append to add the left and right nodes to “List”.  I first visited the left child, then visited the root, and finally visited the right child. I called an empty list “[]” again. I created a templeft and tempright for checking the left and right of the list.



```
InOrder(-, []).
InOrder(BinTree(X, left, right), List)
    :- InOrder(left, templeft), append(List, templeft, leftresult),
       InOrder(right, rightresult), append(leftresult, tempright, List).
```

Finally for PostOrder i copied most of In Order with a few exceptions. For Post order we check the root node last. Again I created an empty list “[]”. Also I used append to add nodes to “NewList”.



```
PostOrder(-, []).
PostOrder(BinTree(X, left, right), List)
    :- PostOrder(left, templeft), PostOrder(right, tempright),
       append(templeft, tempright, NewList), append(NewList, [X], List).
```

**Comparison**

First thing to note when comparing the two languages

Functions:
Firstly in prolog we do not use functions. We use predicates, which can not return a value, they can only answer yes/no questions.

Memory Management
Haskell produces a lot of memory garbage, this is because in haskell it creates new values to store the data every time. Every iteration of a recursive computation rates a new value. This is not the case in Prolog. In prolog it uses a WAM system, which is inbuilt around a set of stacks. There is very little data on how modern prolog systems perform memory management.

Pointers
A pointer is represented in Prolog as a mangled integer, this means they try to make pointers fit into a tagged-integer. However in haskell pointers are “without state”, meaning they do not fit into the language design.

Data Types
In prolog we can use tuples, lists, numbers, atoms, and patterns as data types. When we compare this to haskell, it has the following data types: Integers, Characters, Booleans, Ordering, Tuples. Haskell has three basic ways to declare a

new type: The data declaration, which defines new data types. The type declaration for type synonyms, that is, alternative names for existing types.[4]

## Haskell

**Advantages**

- Very fast
- System that provides extra safety
- Actively being developed/improved

**Disadvantages**

- High learning time
- Lacks widespread implementation
- Very limited use
- Written along time ago
- Not much use in today's work
- Hard to understand

## Prolog

**Advantages**

- Built in list handling
- Easy to build tables/databases
- Easy enough to read

**Disadvantages**

- Limited use
- Poor I/O features
- Very slow
- Limited use
- Not a popular language

## References

- https://en.wikibooks.org/wiki/Haskell/Type_declarations
- https://www.computing.dcu.ie/~davids/teaching.shtml
- https://www.computing.dcu.ie/~davids/teaching.shtml
- StackOverflow

4