**CA208: Assignment 2**

**Student Number:** 20410974

**Student Name:** Duarte Martinho

**Files:** 204410974.pl

**STATEMENT OF NON-PLAGIARISM**

I hereby declare that all information in this assignment has been obtained and presented in accordance with academic rules and ethical conduct, the work that I am submitting in this document, except where indicated, is my own work.

**Predicates**

***add(X, T1, T2).***

**Description**

This predicate is true if it adds an element to the tree T1 and therefore T1 will be the exact same as the T2.

**Approach**

My approach to this predicate was starting by trying to add the element to the tree, so for this I needed to check where I would add it, if it was a 2 node it checked the left (L) or right (R) using the head (H), and if it was a 3 node it looked through left (L), middle (M), right (R) using H1 and H2, after it looked and found a nil/empty node it would add it at this position and after checking with the node at the same position in T2, returning **TRUE** if it is equal, **FALSE** otherwise.

**Test cases**

**Input 1**: add(9, tree(3, 4, tree(1, nil, nil), nil, nil), tree(3, 4, tree(1, nil, nil), nil, tree(9, nil, nil))).

**Output 1**: true.

**Input 2**: add(5, nil, tree(4, nil, tree(5, nil, nil))).

**Output 2**: false.

**Input 3**: add(10, tree(5, nil, nil), tree(5, tree(10, nil, nil), nil)).

**Output 3**: false.

***member(X, T)******.***

**Description**

This predicate is true if it finds the element X in the tree T.

**Approach**

My approach to this predicate was starting by making a recursive function that would through the heads and printing them out making sure to not forget any head even for 2 node trees or 3 nodes trees. After looping through all numbers I would check if any of the heads is equals to the element X and if it is returns **TRUE**, else it would search the subtrees and checking for the element X and returning **TRUE** if it is found and equal, otherwise **FALSE** if it not found and reached the end of the tree.

**Test cases**

**Input 1**: member(1, tree(2, tree(1, nil, nil), nil)).

**Output 1**: true.

**Input 2**: member(1, tree(2, nil, nil)).

**Output 2**: false.

**Input 3**: member(25, tree(5, 13, tree(3, tree(1, nil, nil), tree(4, nil, nil)), tree(8, nil, nil), tree(19, tree(16, nil, nil), tree(20, nil, tree(25, nil, nil))))).

**Output 3**: true.

***height(T, N).***

**Description**

This predicate is true if the height N is the height of the tree T.

**Approach**

My approach to this predicate was by measuring the height of each branches in case of 3 nodes trees checks left (L), middle (M) and right (R), and for 2 nodes trees check for left (L) and right (R), and then we would use our custom predicate max(), where we can pass up to 3 values and it would find the max value, this means that it would count the number of sub branches in that node. It would then return the max value, the output can return **TRUE** if it is the height of the tree is equal to the value supplied, otherwise it would print **FALSE** meaning that the height of the tree (T) and the height (N) supplied does not equal, another way to get the output is give a variable as an argument this will “query” the predicate what is the height of the tree (T).

**Test cases**

**Input 1**: height(tree(2, nil, nil), 0).

**Output 1**: false.

**Input 2**: height(tree(5, 7, tree(2, nil, nil), nil, nil), 2).

**Output 2**: true.

**Input 3**: height(tree(5, 7, tree(2, tree(1, nil, nil), nil), tree(6, nil, nil), tree(9, nil, tree(13, nil, tree(19, nil, nil)))), H).

**Output 3**: H = 4.

***prettyPrint(T).***

**Description**

This predicate is always true, and it displays to the user screen the 2-3 tree.

**Approach**

My approach to this predicate was starting by trying to make a recursive function, I started by printing horizontal and then I changed it to vertical, where I was able to print it, but then it was all in the same “column” so what I had to do was create a variable that kept track of which column it has in other words the column means DEPTH, I had the variables as “D” for “depth” but it was confusing me, so it was better to change the variable not only for code readability but also for me, this depth would print out the value after reaching 0, after this it returns the tree on the users screen and returning **TRUE** as it always returns **TRUE**.

**Test cases**

**Input 1**: prettyPrint(tree(3, tree(1, nil, nil), tree(5, nil, nil))).

**Output 1**:

nil

1

nil

3

nil

5

nil

true.

**Input 2**: prettyPrint(tree(5, 10, tree(1, nil, nil), tree(7, tree(5, nil, nil), tree(8, nil, nil)), tree(18, nil, nil))).

**Output 2**:

nil

1

nil

5,10 nil

5

nil

7

nil

8

nil

nil

18

nil

true.

**Input 3**: prettyPrint(tree(4, 10, tree(2, tree(1, nil, nil), tree(3, nil, nil)), tree(6, tree(5, nil, nil), tree(9, nil, nil)), tree(15, tree(12, nil, nil), tree(20, tree(18, nil, nil), tree(28, nil, nil))))).

**Output 3**:

nil

1

nil

2

nil

3

nil

4,10 nil

5

nil

6

nil

9

nil

nil

12

nil

15

nil

18

nil

20

nil

28

nil

true.