CA208 Prolog Assignment

Adrian Irwin – 20415624

I acknowledge that the work submitted in this assignment is my own work and does not breach the DCU Academic Integrity Policy.

Tree Definitions:

- nil for an empty tree,

```
nil.
```

- t(LeftSubtree, Root, RightSubtree)
  - 1 root element
  - Left subtree, elements <= root
  - Right subtree, elements > root
  - For example, root 4, 2 as root of left subtree, and 8 as root of right subtree:

```
t( t(nil, 2, nil), 4, t(nil, 8, nil) ).
```

- t(LeftSubtree, Root1, MiddleSubtree, Root2, RightSubtree)
  - 2 root elements
  - Left subtree, elements <= root1
  - Middle subtree, elements > root1 but <= to root2
  - Right subtree, elements > root2
  - For example, roots 6 and 10, 8 as root of middle subtree, and both right and left subtree have no elements:

```
t( nil, 6, t(nil, 8, nil), 10, nil).
```

Available Predicates:

member(X, T).
add(X, T1, T2).
height(T, N).
prettyPrint(T).

member(X, T):

Returns true if X is in the given tree T.

Base Case(s):

```
member(X, t(_, X, _)).
member(X, t(_, Y, _, Z, _)) :- X =:= Y; X =:= Z.
```

Check if the given element X is equal to the root elements, for either a 1 root tree or 2 root tree

Normal Case(s):

```
member(X, t(L, Y, _)) :- X =< Y, member(X, L).
member(X, t(_, _, R)) :- member(X, R).
member(X, t(L, Y, _, _, _)) :- X =< Y, member(X, L).
member(X, t(_, Y, M, Z, _)) :- X > Y, X =< Z ,member(X, M).
member(X, t(_, _, _, Y, R)) :- X > Y, member(X, R).
```

If X is not a root element of either a 1 root tree or a 2 root tree, multiple checks are done to see if X is a member in the subtrees of a 1 root or 2 root tree.
First we check a 1 root tree, we check if X is less than the root, if it is we recursively call member with X and the Left subtree.
If X is part of a 1 root tree and is not less than the root we recursively call member with X and the right subtree.
If X is not in the 1 root tree we move onto the 2 root trees.
We check if X is less than the first root, if it is we recursively call member with X and the left subtree.
If X is not less than the first root we check if X is greater than the first root and less than the second root. If it is, we recursively call member with X and the middle subtree.
Lastly, we check if X is greater than the second root, we recursively call member with X and the right subtree.

Tests:
?- member(2, t(nil, 2, nil)).
true .

?- member(7, t(nil, 2, nil, 7, nil)).
true .

?- member(8, t(nil, 5, t(nil, 8, t(nil, 9, nil)), 10, t(nil, 18, t(nil, 20, nil, 25, nil)))).
true .

add(X, T1, T2):

Returns true if adding X to the tree T1 gives you the tree T2.

Base Case(s):

```
add(X, nil, t(nil, X, nil)).
add(X, t(nil, Y, nil), t(nil, Y, nil, X, nil)) :- X > Y.
add(X, t(nil, Y, nil), t(nil, X, nil, Y, nil)) :- X =< Y.
```

Adding X to an empty tree will give a new tree with X as the root so we check for that.
Adding X to an already existing 1 root tree will create a new 2 root tree so we will have to check that X is being added in the right position. If it is larger than the existing root it will be in Root2 otherwise it will be in Root1 and the existing root will be in Root2.

Normal Case(s):

```
add(X, t(L, Y, M, Z, R), t(L1, Y, M, Z, R)) :- X =< Y, add(X, L, L1).
add(X, t(L, Y, M, Z, R), t(L, Y, M1, Z, R)) :- X =< Z, X > Y, add(X, M, M1).
add(X, t(L, Y, M, Z, R), t(L, Y, M, Z, R1)) :- X > Z, add(X, R, R1).
```

When adding X to an existing tree, we first find which subtree we should add X to by comparing it to the roots.
First we check if X is less than the first root, if it is then we recursively call add with X and the left subtree.
Then we check if X is in between the two roots, if it is we recursively call add with X and the middle subtree.
Lastly we check if X is larger than the second root, if it is we recursively call add with X and the right subtree.

Tests:

?- add(10, t(nil, 13, nil, 15, nil), t(t(nil, 10, nil), 13, nil, 15, nil)).
true .

?- add(8, t(nil, 5, nil, 19, nil), t(nil, 5, t(nil, 8, nil), 19, nil)).
true .

?- add(25, t(nil, 8, nil, 15, nil), t(nil, 8, nil, 15, t(nil, 25, nil))).
true .

?- add(25, t(nil, 8, nil),t(nil, 8, nil, 25, nil)).
true .

height(T, N):

Returns true if adding N is the height T.
Height for the tree will always include the height of the root. So a tree with only a root and no children has a height of 1.

Base Case(s):

```
height(nil, 0).
```

If the tree is empty then the height of the tree is 0.

Normal Case(s):

```
height(t(L, _, R), N) :-
    height(L, NL), height(R, NR),
    N is max(NL, NR) + 1.

height(t(L, _, M, _, R) , N) :-
    height(L, NL), height(M, NM), height(R, NR),
    N is max(NL, max(NM, NR)) + 1.
```

To find the height of the subtree we recursively call height on the left, middle and right subtrees. We use the max function to find out which subtrees are the longest. Every time we recursively call height we add one onto the height.

Tests:
?- height(t(nil, 5, nil, 10, nil), 1).
true.

?- height(t(t(nil, 1, nil), 2, t(nil, 3, nil)), 2).
true.

?- height(t(nil, 5, t(t(nil, 3, nil), 4, t(nil, 6, nil), 7, t(nil, 8, nil)), 10, t(nil, 11, nil, t(nil, 20, nil), nil)), 3).
true.

prettyPrint(T):

Always returns true and prints out the given tree T in readable format for the user.
I tried to print out the tree in a vertical manner but was unable to correctly implement prettyPrint in a way that was readable for the user.

Base Case(s):

```
prettyPrint(nil) :- write("nil").
```

If we have an empty tree we print nil as there is no content that can be printed for empty trees.

Normal Case(s):

```
prettyPrint(t(L,X,R)) :-
    height(t(L,X,R), H),

    Num is H * 10,

    tab(Num),
    write(X),

    Right is Num - 5,
    nl,

    prettyPrint(L),

    tab(Right),
    prettyPrint(R),

    nl.
```

The above code is for printing a 1 root tree, first I find the height of the tree so that we will know how far in we will have to indent our items based on how far down the tree goes.
By creating a variable for how many spaces we want, we multiply the height by 10 and use tab() to print the amount of spaces we want.
We then print out the root of the tree.
Afterwards we create a variable named Right that equals the space count minus five so that we have a smaller space between the two roots of the left and right subtrees.
We print a new line then recursively call prettyPrint on the left subtree, we print the shorter tab and recursively call prettyPrint on the right subtree. At the end we print another new line.

```
prettyPrint(t(L, X, M, Y, R)) :-
    height(t(L, X, M, Y, R), H),

    Num is H * 10,
    tab(Num),
    write(X), write(","), write(Y),

    Mid is Num - 3,
    Right is Num - 5,
    nl,

    prettyPrint(L),

    tab(Mid),
    prettyPrint(M),

    tab(Right),
    prettyPrint(R),

    nl.
```

The above code is for printing a 2 root tree. It is almost the same as printing a 1 root tree but we also have to print the second root along with the first root. We also need to print the middle subtree, there is a smaller gap between the left and middle subtree so that the print doesn't take up all of the space in the terminal.

Tests:

Printing an empty tree:

```
?- prettyPrint(nil).
nil
true.
```

Printing a simple 1 root tree with no subtrees.

```
?- prettyPrint(t(nil, 5, nil)).
        5
nil     nil
true.
```

Printing a 2 root tree with multiple subtrees.
?-prettyPrint(t(t(nil, 3, nil), 7, t(t(nil, 8, nil), 9, nil, 12, t(nil, 13, nil)), 14, nil)).

The diagram above is the output from the above command.
As you can see I was not able to print the output properly but have outlined where the outputs ended up printing in relation to each other.
Below is the output without any of the red outlines.