Introduction to The Problem

- My understanding of Binary Search Trees (BST) mainly originated from my Python Programming Studies with Charlie Daly.
    - CA268: Computer Programming 3: Algorithms and Data Structures.

- This module had already made me familiar with the structure of Binary Search Trees.
    - mainly their root node, left subtree and right subtree.

- Initially I used the google sites link cited below to familiarise myself with the idea of a Binary Search Tree as implemented through Prolog.
    - Sites.google.com. (2018). *4. Binary Trees - Prolog Site*. [online] Available at: https://sites.google.com/site/prologsite/prolog-problems/4 [Accessed 11 Dec. 2018].
    - This website also allowed me to compile many of my test cases to ensure I was on the right track.

- I found many of the methods of BST implementation in Python to be transferable to the task in Prolog given here.
- My prolog notes were useful for implementing the search function, they showed how to apply a similar function which I was able to work from and edit.
- To understand tree traversals better I referred to this article from the follow link.
    - GeeksforGeeks. (2018). *Tree Traversals (Inorder, Preorder and Postorder) - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ [Accessed 11 Dec. 2018].

Explanation of Thinking Behind My Prolog Code

1 – emptyBT

```
emptyBT(nil) :- bTree(nil).
```

The emptyBT function defines an empty binary tree and is initiated as nil as it consists of no root and no left subtree or right subtree.

Test Code for emptyBt

emptyBT function	**Result**
?- emptyBT(nil).	True.
?- emptyBT(bTree(6,nil,nil)).	False.

2 - bTree(N,T1,T2)

```
bTree(nil).
```

The empty binary tree is a tree with no root, left subtree or right subtree as explained above.

N is the root - a singleton variable (_), T1 is the left root its own subtree and the same applies for T2.

```
bTree(bTree(_,T1,T2)) :- bTree(T1),bTree(T2).
```

Test Code for bTree:

bTree function	**Result**
?- bTree(bTree(3,nil,nil)).	True.

3 - insert(I,T1,T2)

If the binary search tree is empty (nil), the new resulting BST will consist of a root that is the item that is being inserted.

```
insert(I,nil,bTree(I,nil,nil)).
```

- If the item being inserted (I) is less than our root N the item will be inserted into the left subtree.
- Inserting I into BST with root N, Left subtree T1 and Right Subtree T2 will result in the new BST consisting of the original root N, new left subtree L1 with item added and original right subtree T2.

```
insert(I,bTree(N,T1,T2),bTree(N,L1,T2)) :-
   I < N,
   insert(I,T1,L1).
```

- If the item being inserted I is greater than our root N the item will be inserted into the right subtree.
- Inserting I into BST with root N, Left subtree T1 and Right Subtree T2 will result in the new BST consisting of the original root N, original left subtree T1 and updated right subtree L2.

```
insert(I,bTree(N,T1,T2),bTree(N,T1,L2)) :-
   I > N,
   insert(I,T2,L2).
```

Test Code for insert

Insert Function	**Result**
?- insert(5, nil, X).	X = bTree(5, nil, nil) .
?- insert(7,bTree(5,nil,nil),X).	X = bTree(5, nil, bTree(7, nil, nil)) .
?- insert(14,bTree(26,nil,nil),bTree(26,bTree(14,nil,nil),nil)).	True

4 - preorder(T,L)

A preorder traversal of an emptyBT (nil) would return an empty list ([]) due to no elements being present in the BST.

From research I found that:

- a preorder traversal visits the BST root first (N),
- the left tree (T1) next and
- finally, the right subtree (T2) is traversed.

- The preorder traversal of the left sub-tree would be stored in L1 and the preorder traversal of the right sub-tree would be stored in L2.
- To combine all the traversals together the root node N would be set as the list head with L1 being the tail of the list, the two lists L1 and L2 are appended together and the resulting combined list being stored in the variable L.

```
preorder(nil, []).
preorder(bTree(N,T1,T2), L):-
   preorder(T1,L1), preorder(T2,L2), append([N|L1],L2,L).
```

Test Code:

?- preorder(nil, X). **Result:** X = [].
?- preorder(bTree(6, bTree(4, bTree(2,nil,nil), bTree(5,nil,nil)), bTree(9, bTree(7,nil,nil), nil)), A).
Result: A = [6, 4, 2, 5, 9, 7].

5 - inorder(T,L)

is true if L is a list of nodes generated by a inorder traversal of the binary tree T.

An inorder traversal of an emptyBT (nil) would return an empty list ([]) due to no elements being present in the BST.

From research I learned that an inorder traversal:

- traverses the left subtree (T1) first,
- the root (N) is visited next and
- finally, the right subtree (T2) is traversed.

- The inorder traversal of the left subtree is stored in L1.
- The inodrer traversal of the right subtree is stored in L2.
- The left subtree traversal is then appended to the right subtree traversal with the root node being the head of the second list.

```
inorder(nil, []).
inorder(bTree(N,T1,T2), L):-
   inorder(T1,L1), inorder(T2,L2),
append(L1,[N|L2],L).
```

Test Code for inorder

Inorder function
?- inorder(nil, X). **Result:** X = [].
?- inorder(bTree(6, bTree(4, bTree(2, nil, nil), bTree(5, nil, nil)), bTree(9, bTree(7, nil, nil), nil)), A). **Result:** A = [2, 4, 5, 6, 7, 9].

6 - postorder(T,L)

A postorder traversal of an emptyBT (nil) would return an empty list ([]) due to no elements being present in the BST.

From the geeksforgeeks page I learned in a postorder traversal:

- The left subtree is traversed first.
- The right tree is then traversed.
- The root node is last visited.

- I implemented this by doing a postorder traversal on T1 (the left subtree) and storing this in L1 then moving onto the right subtree (T2) and storing this traversal in L2.
- These two list were then appended together to form R1 and finally append the root node N to the end of the list to form L.

```
postorder(nil, []).
postorder(bTree(N,T1,T2), L):-
   postorder(T1,L1), postorder(T2,L2), append(L1,L2,R1), append(R1,[N],L).
```

Test Code for postorder

?- postorder(nil, X). **Result:** X = [].
?- postorder(bTree(6, bTree(4, bTree(2, nil, nil), bTree(5, nil, nil)), bTree(9, bTree(7, nil, nil), nil)), A).
Result: A = [2, 5, 4, 7, 9, 6]

7 - search(T,I)

If the root matches the element I, then stop our element is in the BST.

```
search(bTree(I,_,_), I).
```

If the root is greater than or equal to the element being searched, then we look in the left subtree for our element.

```
search(bTree(N,T1,_), I) :-
        N >= I, search(T1,I).
```

If the root is less than the element, then the right subtree is searched the element I.

```
search(bTree(N,_,T2), I) :-
        I > N, search(T2,I).
```

The search function will return a value True or False whether the value is in the BST or not.

Test Code for search

? - search(bTree(3,bTree(2,nil,nil),bTree(4,nil,nil)), 3).	**True**
?- search(bTree(3,bTree(2,nil,nil),bTree(4,nil,bTree(5,nil,nil))), 5).	**True**
?- search(bTree(7,nil,nil),7).	**True**

8 - height(T,H)

The height function works by returning H the height of the binary tree T.

The height of an empty tree is 0.

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

The height of a non-empty tree is computed from examining the left subtree T1 and the right subtree T2.

```
height(bTree(_,T1,T2), H) :-
```

Computing the depth of the left subtree T1 and store the value in the H1 variable

```
height(T1, H1),
```

Compute the depth of the right subtree T2 and use the variable H2 to store that value.

```
height(T2, H2),
```

The overall depth is 1 more than the maximum depth of both subtrees to include the root node.

```
H is 1 + max(H1, H2).
```

Test Code for height

```
?- height(nil,X).
X = 0
```

```
?- height(bTree(6, bTree(4, bTree(2, nil, nil), bTree(5, nil, nil)), bTree(9, bTree(7, nil, nil), nil)), A).
A = 3
```

```
?- height(bTree(3,nil,nil), X).
X = 1.
```