Declaration on Plagiarism and Assignment Submission Form

This form must be filled in and completed by the student(s) submitting an assignment

|  |  |
| --- | --- |
| Student Name(s): | Tomas Baltrunas |
| Student Number(s): | 17350793 |
| Programme: | BSc. Computer Applications |
| Assignment Title: | Logic Assignment 2 Binary Trees |
| Module Code: | CA208 |
| Module Coordinator: | David Sinclair |
| Submission Date: | 2018/12/17 |

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at

<http://www.dcu.ie/info/regulations/plagiarism.shtml>, <https://www4.dcu.ie/students/az/plagiarism>

and/or recommended in the assignment guidelines

Name(s): \_\_\_\_\_\_\_Tomas Baltrunas Date: \_\_\_\_\_\_\_2018/12/15

Implementing Binary Trees in Prolog Description of the Design of the Application

# Implementation of Required Predicates

## Binary Tree (BT) definition

BT representation in Prolog using structures:

* The atom **emptyBT** represents an empty BT with no nodes
* The structure **bTree(N, T1, T2)** represents a non-empty BT, with arguments **N** as the root node’s item, **T1** as the left sub-tree (an empty or non-empty BT), and **T2** as the right sub-tree.
  + The BT is a binary *search* tree (BST). It satisfies the *Binary Search property*: Each node on the left sub-tree is less than or equal (this is not a set) to the root, and each node on the right sub-tree is greater than the root.
  + Example: The expression ‘bTree(2, bTree(1, emptyBT, emptyBT), bTree(3, emptyBT, emptyBT).’ represents a tree with value 2 at its root node, node with value 1 on the left, and node with value 3 on the right.

I assume that if the user writes a binary tree ‘expression’ by themselves, it is up to them to make sure that the *Binary Search property* holds and that the binary tree is written using correct syntax. Instead of messing with the ‘internals’ of the tree representation, I recommend that the user instead makes use of the **insert** predicate, eg: ‘insert(10, emptyBT, T)’, to get started.

In the *helper predicates* section I have provided the **bst** predicate to check that a binary tree is a binary search tree, and the **bt** predicate to check the syntactic validity of a binary tree expression. I have tried to make **bTree** a predicate in itself that gives true/false based on if it‘s a valid BST, but the solutions ended up not being very elegant due to the many combinations possible from using bTree with emptyBT.

Note that in this implementation binary tree items can only be numbers (integers or floats). Character literals / constants are not supported (only numerical ordering is supported, not lexicographical).

Each predicate defined below is deterministic (produces one and only one answer).

## 

## Insert

The predicate **insert(I, T1, T2)** appropriately inserts a single node value **I** into an old tree **T1**, resulting in a new tree **T2**. The *Binary Search* property still holds after the insertion.

The predicate operates as follows:

* The predicate starts at the root of the old tree and picks either the left or the right branch/subtree, based on whether **I** is smaller/equal/greater to the root value.
* It recurses on the branch picked. Recursion results in a new sub-tree with **I** inserted.
* The base case is insertion into an empty tree. The resulting tree is a tree with one node only, with **I** as its value.
* The new tree is instantiated with one subtree being the result of recursion on the branch picked and another sub-tree being the old unmodified subtree of the branch not picked.

Example:

‘insert(4,

bTree(10, bTree(5, emptyBT, emptyBT), bTree(15, emptyBT, emptyBT)),

T).’

Results in

‘T =

bTree(10, bTree(5, bTree(4, emptyBT, emptyBT), emptyBT), bTree(15, emptyBT, emptyBT))’

## 

## Traversal

**preorder(T, L)**, **inorder(T, L)** and **postorder(T, L)** all result in a traversal (visit to each node) of the tree **T**. The order in which the items were visited is stored in the Prolog list structure **L** and depends on the traversal used. All traversals are assumed to be *Depth First Search*, and from *left to right (LR)*. Each traversal consists of three parts: traversing the root (N), the left subtree (L) and the right subtree (R). The pre/in/post prefixes simply indicate when N is traversed.

Each design uses the **‘my\_concat’** supporting predicate, defined inside the file. This concatenates two lists together in the order they were passed into a resulting third list.

For each predicate the base case for recursion is the empty tree, resulting in an empty ‘traversal list’.

The **preorder** predicate (NLR) is done as follows:

* The traversal list **L** is split into the head and the tail.
* The head is made equal to the root of **T**.
* The subtrees are preorder traversed recursively, resulting in their own traversal lists.
* The tail of **L** is then the concatenation of the left traversal list firstly and the right traversal list secondly.
* This gives a Node-Left-Right order

The **inorder** predicate (LNR) is designed like this:

* Recursively find the inorder traversal of the left and right subtrees, resulting in left and right traversal lists.
* Concatenate the left traversed list to a singleton list with root as the only item. This is the intermediate list.
* Concatenate the intermediate list to the right traversed list. The full list **L** is then obtained.
* This Left-Node-Right order gives a sorted list (due to the *Binary Search* *property*).

The **postorder** predicate (LRN) is modified like this:

* Repeat the steps of **inorder**, but concatenate the left traversal list to the right traversal list first. Then concatenate this intermediate list to the root node singleton list.

Do not use this to build trees out of traversal lists (doesn’t work).

Example:

Given the tree: ‘bTree(10, bTree(4.5, emptyBT, emptyBT), bTree(13, bTree(11, emptyBT, emptyBT), emptyBT))’

The resulting traversals are:

‘PRE = [10, 4.5, 13, 11],

IN = [4.5, 10, 11, 13],

POST = [4.5, 11, 13, 10].’

## Search

**search(T, I)** results in true/false based on whether the value **I** is contained in / not contained in the BT **T**.

The predicate works as follows:

* If the **T** empty, stop and fail (an empty tree has no items).
* If the root of **T** has the same value as **I**, stop and do not backtrack (success, found the element).
* Pick the left / right branch based on if **I** is less / equal / greater than the **T**’s root, and recurse on the picked branch subtree.
  + The fact that the BT upholds the *binary search property* is used (this makes the search very fast).
  + Recursion is terminated by the above two base cases.

The predicate gives true or false only. It should not be used to get all the elements of a BT (by passing **I** as a variable), as the arguments won’t be sufficiently instantiated to compare the item to the current root node when picking a branch. To get all the nodes of a tree refer to the traversal predicates instead.

Example:

‘search(bTree(10, bTree(4, emptyBT, emptyBT), bTree(11, emptyBT, emptyBT)), 4).’

Results in

‘true.’

## 

## Height

The **height(T,H)** predicate gives a non-negative integer **H** representing the ‘height’ of the tree **T**. The following ‘height’ ‘base values’ are used: an empty tree is height 0, a tree with one node is height 1. Intuitively height is the greatest number of nodes one can visit going from the root to the bottom of the tree downwards.

I have implemented the predicate like this:

* Recursively get the height of the left and right subtrees.
* The base case for the recursion is that the empty tree has height 0.
* Pick the taller (greater height) of the two subtrees.
* The height of the taller sub-tree added to the height of the root node (height 1) is the overall height **H** of the tree.

This predicate should be used only to get or check the integer height of a known tree, not to generate trees of a certain height (eg: calling ‘height(T, 10).’) (results in stack error).

Example:

Having a tree ‘T = bTree(2, bTree(1, emptyBT, emptyBT), bTree(3, emptyBT, bTree(4, emptyBT, emptyBT)))‘

‘height(T, H).’

Assigns

‘H = 3.’

# 

# 

# Helper Predicates

**bt(T)** checks that the data object **T** represents a valid binary tree. Produces true or false.

* If **T** is **emptyBT** then T is a binary tree (base case).
* If **T** is a structure **bTree(N, T1, T2)** then it is a binary tree if **N** is a number (integer or float), and **T1** and **T2** are binary trees themselves (recursive case).

**bst(T)** checks that a binary tree **T** is a binary search tree, giving true or false.

The **all\_left(N, L)** and **all\_right(N, L)** supporting predicates are used. These check that each item in list **L** is less/equal/greater than the item **N**, based on whether the left or the right version is called. Essentially the predicates check for the *Binary Search property*.

* If **T** is **emptyBT**, it is a BST (base case)
* Else **T** is a **bTree(N, T1, T2)** with a root node **N** and two children subtrees **T1** and **T2**.
* Recursively check that each subtree is a BST.
* Get all the nodes of each of the subtrees.
  + Use one of the traversal predicates. The order of traversal does not matter.
* Check that the root node and the subtree nodes together satisfy the *Binary Search property*.
  + Use the **all\_left**/**all\_right** support predicates.

**insert\_list(L, T1, T2)** inserts a list of items **L** into the old tree **T1**, resulting in a new tree **T2** with all the items inserted. Uses the **insert** predicate for each insertion.

* Use a standard Prolog pattern to go through each item in a list recursively.
* On each item apply the **insert** predicate, ‘carrying forward’ the resulting tree onto the next insertion.

# References

<https://en.wikipedia.org/wiki/Binary_search_tree>

<https://en.wikipedia.org/wiki/Tree_traversal>

<https://www.computing.dcu.ie/%7Edavids/courses/CA208/CA208_Prolog_2p.pdf>

END