8 Trees. Operations on trees

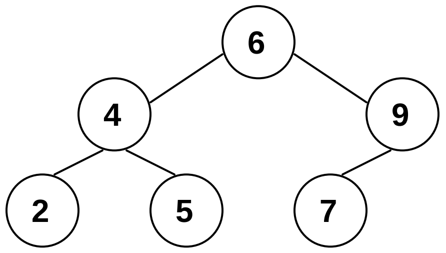
In this lesson you will explore operations on trees. Two types of trees will be addressed: binary search trees and ternary trees, with corresponding operations. In Prolog, trees are modeled as recursive structures. The empty tree is denoted through a constant, which is typically the symbol nil.

8.1 Binary Search Trees

Binary search trees can be represented in Prolog by using a recursive structure with three arguments: the *key* of the root, the *left sub-tree* and the *right sub-tree* – which are structures of the same type. The empty (null) tree is usually represented as the constant nil.

Example 8.1: The binary search tree below can be specified in Prolog using the following structure:

```
t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil)).
```



Hint: In order to avoid writing each time you want to make a query such a long construction for the input tree, you may choose to “save” a few “test” instances as predicate facts in the source file (and the predicate base), e.g.:

```
tree1(t(6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil))).
tree2(t(8, t(5, nil, t(7, nil, nil)), t(9, nil, t(11, nil, nil)))).
…
```

and instantiate a variable in the query:

```
?- tree1(T), some_useful_predicate(T, …).
```

8.1.1 Tree traversal – preorder, inorder, postorder

Perhaps the simplest operations on trees are the traversal operations. As you already know, there are three possible modes of traversing trees: *inorder*, *preorder* and *postorder*, depending on the order in which the nodes are processed.

The inorder traversal processes the left sub-tree first, then the root node, then the right sub-tree. The predicate is presented below:

```
inorder(t(K,L,R), List):-inorder(L,LL), inorder(R, LR),
                         append(LL, [K|LR],List).
inorder(nil, []).
```

You may observe that, even though the recursive call for the right sub-tree is performed before processing the root node, the correct order of the nodes is maintained when constructing the output list, in the call to append. So, the nodes on the left sub-tree appear first in the list, then the root node, then the keys in the right sub-tree.

The same observation applies for the preorder and postorder traversals, presented below:

```
preorder(t(K,L,R), List):-preorder(L,LL), preorder(R, LR),
                         append([K|LL], LR, List).
preorder(nil, []).
postorder(t(K,L,R), List):-postorder(L,LL), postorder(R, LR),
                         append(LL, LR,R1), append(R1, [K], List).
postorder(nil, []).
```

Exercise 8.1: Study (by tracing) the execution of the following queries:

1. ?- tree1(T), inorder(T, L).
2. ?- tree1(T), preorder(T, L).
3. ?- tree1(T), postorder(T, L).

8.1.2 Pretty print

Pretty printing trees in Prolog is very useful for visualizing the correctness of the other predicates on trees. The simplest strategy for pretty printing is to perform an *inorder* traversal of the tree, and print each node at a number of tabs equal to the *depth* at which the node appears in the tree. Also, each node is printed on a separate line. The root of the tree is considered to be at depth 0.

A pretty printing of the tree in *example 8.1* is presented below:

```
		2
	4
		5
6
		7
	9
```

If we study the listing above more closely we observe that the keys on the left are printed first, then the root, then the keys on the right. This suggests that an inorder traversal is suited for obtaining such a pretty print. Therefore, the predicate(s) which output the pretty printing above are:

```
% inorder traversal
pretty_print(nil, _).
pretty_print(t(K,L,R), D):-D1 is D+1, pretty_print(L, D1), print_key(K, D),
                                pretty_print(R, D1).

% predicate which prints key K at D tabs from the screen left margin and then
% proceeds to a new line
print_key(K, D):-D>0, !, D1 is D-1, write('\t'), print_key(K, D1).
print_key(K, _):-write(K), nl.
```

Hint: nl sends a newline to the standard output stream; equivalent to write('\n').

Exercise 8.2: Study the execution of the following query:

```
?- tree2(T), pretty_print(T, 0).
```

8.1.3 Searching for a key

Because of the ordering of the keys in a binary search tree, searching for a given key is very efficient. The algorithm is sketched below:

```
if currentNode = null then return -1; //not found
if searchKey = currentNode.key then return 0; //found
else if searchKey < currentNode.key
    then search(searchKey, currentNode.left); //search left subtree
else
    search(searchKey, currentNode.right); //search right subtree
```

It is very straightforward to transform the pseudo code above in Prolog specifications. Since we want our predicate to fail in case the key is not found, we can either specify this fact explicitly, by using an explicit fail for when a nil is reached, or implicitly, by not covering the case of reaching a nil. The search_key predicate is presented below:

```
search_key(Key, t(Key, _, _)):-!.
search_key(Key, t(K, L, _)):-Key<K, !, search_key(Key, L).
search_key(Key, t(_, _, R)):-search_key(Key, R).
```

Exercise 8.3: Study the execution of the following queries:

1. ?- tree1(T), search_key(5, T).
2. ?- tree1(T), search_key(8, T).

8.1.4 Inserting a key

Each new key is inserted as a leaf node in a binary search tree. Before performing the actual insert, we must search for the appropriate position of the new key. If the key is found during the search process, no insertion occurs. When reaching a nil in the search process, we create the new node.

The insert_key predicate is presented below:

```
insert_key(Key, nil, t(Key, nil, nil)):-write('Inserted '), write(Key), nl.
insert_key(Key, t(Key, L, R), t(Key, L, R)):-!, write('Key already in tree\n').
insert_key(Key, t(K, L, R), t(K, NL, R)):-Key<K, !, insert_key(Key, L, NL).
insert_key(Key, t(K, L, R), t(K, L, NR)):- insert_key(Key, R, NR).
```

Exercise 8.4: Study the execution of the following queries:

1. ?- tree1(T), pretty_print(T, 0), insert_key(8, T, T1), pretty_print(T1, 0).
2. ?- tree1(T), pretty_print(T, 0), insert_key(5, T, T1), pretty_print(T1, 0).
3. ?- insert_key(7, nil, T1), insert_key(12, T1, T2), insert_key(6, T2, T3), insert_key(9, T3, T4), insert_key(3, T4, T5), insert_key(8, T5, T6), insert_key(3, T6, T7), pretty_print(T7, 0).

8.1.5 Deleting a key

The deletion of a key in a binary search tree also requires that the key be initially searched in the tree. Once found, we distinguish among three situations:

- *We have to delete a leaf node*
- *We have to delete a node with one child*
- *We have to delete a node with both children*

The first two cases are rather simple. For the third case we have two alternatives: either replace the node to delete with its predecessor (or successor) – by reestablishing the links correctly, or to “hang” the left sub-tree in the left part of the right sub-tree (or vice-versa).

We have implemented the first alternative in the delete_key predicate, below:

```
delete_key(Key, nil, nil):-write(Key), write(' not in tree\n').
delete_key(Key, t(Key, L, nil), L):-!. % this clause covers also case for leaf
                                                    (L=nil)
delete_key(Key, t(Key, nil, R), R):-!.
delete_key(Key, t(Key, L, R), t(Pred, NL, R)):-!, get_pred(L, Pred, NL).
delete_key(Key, t(K, L, R), t(K, NL, R)):-Key<K, !, delete_key(Key, L, NL).
delete_key(Key, t(K, L, R), t(K, L, NR)):- delete_key(Key, R, NR).
```

```
get_pred(t(Pred, L, nil), Pred, L):-!.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):-get_pred(R, Pred, NR).
```

Exercise 8.5: Study the execution of the following queries:

1. ?- tree1(T), pretty_print(T, 0), delete_key(5, T, T1), pretty_print(T1, 0).
2. ?- tree1(T), pretty_print(T, 0), delete_key(9, T, T1), pretty_print(T1, 0).
3. ?- tree1(T), pretty_print(T, 0), delete_key(6, T, T1), pretty_print(T1, 0).
4. ?- tree1(T), pretty_print(T, 0), insert_key(8, T, T1), pretty_print(T1, 0), delete_key(6, T1, T2), pretty_print(T2, 0), insert_key(6, T2, T3), pretty_print(T3, 0).

8.1.6 Height of a binary tree

The height of a binary tree can be computed using the following idea:

- *the height of a nil node is 0*
- *the height of a node other than nil is the maximum between the height of the left sub-tree and the height of the right sub-tree, plus 1 (max{hLeft, hRight} + 1)*

Therefore, the predicate which computes the height of a binary tree can be specified in Prolog as:

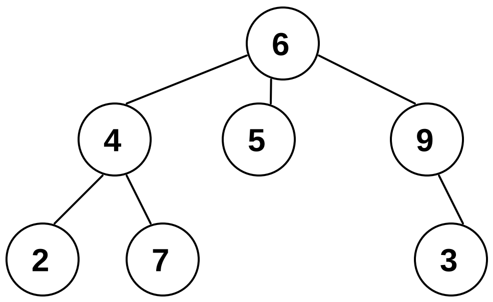
```
% predicate which computes the maximum between 2 numbers
max(A, B, A):-A>B, !.
max(_, B, B).

% predicate which computes the height of a binary  tree
height(nil, 0).
height(t(_, L, R), H):-height(L, H1), height(R, H2), max(H1, H2, H3),
                        H is H3+1.
```

Exercise 8.6: Study the execution of the following queries:

1. ?- tree1(T), pretty_print(T, 0), height(T, H).
2. ?- tree1(T), height(T, H), pretty_print(T, 0), insert_key(8, T, T1), height(T1, H1), pretty_print(T1, 0).

8.2 Ternary Trees

In a ternary tree, each node can have up to three children. Although it is not as easy as for binary trees, ordering relations can be established for ternary trees as well. For the sake of simplicity, we will not impose any ordering relation for the keys in a ternary tree. Below you have an example of a ternary tree:



We shall study a few operations for ternary trees. In Prolog, they are represented in the same manner as binary trees – through recursive structures.

8.2.1 Pretty print

Since a node in a ternary tree can have up to three children, we need a different strategy for pretty printing such structures than the one employed for binary trees. A solution would be to print each node at *depth* tabs from the left screen margin (as before); moreover a node's sub-trees should appear bellow the node (should be printed after the node is printed), but above the next node at the same depth:

```
6
        4
                2
                7
        5
        9
                3
```

Such a pattern could be achieved through a pre-order traversal of the tree (*Root, Left, Middle, Right*), and printing each key on a line, at *depth* tabs from the left margin.

Exercise 8.7: Implement pretty printing for a ternary tree. Study the execution of one or two queries for your predicate.

8.2.2 Tree traversal

Tree traversal operations can be performed on ternary trees as well. The order of visiting the nodes in each of the tree walks is the following:

- inorder: *Left->**Root**->Middle->Right*

- preorder: ***Root****->Left->Middle->Right*
- postorder: *Left->Middle->Right->****Root***

Exercise 8.8: Implement the tree traversal operations for a ternary tree. Study the execution of different queries for the resulting predicates.

8.2.3 Tree height

The height of a ternary tree can be computed using the same idea from binary trees; the only difference is that you have to consider three branches, instead of two.

Exercise 8.9: Write a predicate which computes the height of a ternary tree. Study the execution of different queries for your predicate.

8.4 Quiz exercises

q8-1. Alter the predicate for the inorder traversal of a binary search tree such that the keys are printed on the screen instead of collecting them in a list.

q8-2. Alter the delete_key predicate for deleting a key from a binary search tree, such that when the key is in a node with two children you apply the second solution: “hang” the left sub-tree to the right sub-tree, or vice-versa.

q8-3. Write a predicate which collects, in a list, all the keys found in leaf nodes of a binary search tree.

8.5 Problems

p8-1. Write a predicate which computes the diameter of a binary tree (*diam(Root) = max{diam(Left), diam(Right), height(Left)+height(Right)+1}*).

```
?- tree1(T), diameter(T, D).
D =  4 ,
T = (6, t(4, t(2, nil, nil), t(5, nil, nil)), t(9, t(7, nil, nil), nil)) ? ;
no
```

p8-2. (**) Write a predicate which collects, in a list, all the nodes at the same depth in a ternary tree.

p8-3. (**) Let us call a binary tree *symmetric* if you can draw a vertical line through the root node and then the right sub-tree is the mirror image of the left sub-tree. Write a predicate symmetric(T) to check whether a given binary tree T is symmetric. We are only interested in the structure, not in the contents of the nodes.

Hint: Write a predicate mirror(T1, T2) first to check whether one tree is the mirror image of another.

```
?- tree1(T), symmetric(T).
no

?- tree1(T), delete_key(2, T, T1), symmetric(T1).
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)),
T1 = t(6,t(4,nil,t(5,nil,nil)),t(9,t(7,nil,nil),nil)) ? ;
no
```