Trees

1 Theoretical Considerations

In this lesson, we will explore operations on Trees. Two types of trees will be addressed: binary search trees and ternary trees. In Prolog, trees are modeled as recursive structures. An empty tree is represented by a constant, typically by the symbol *nil*.

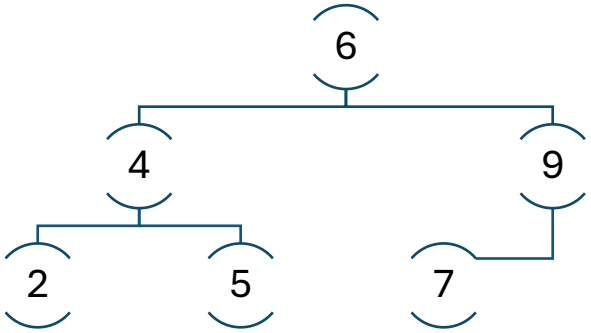
1.1 Representation

The Binary Search Tree is represented through a recursive structure with 3 arguments:

- the key of the current node,
- the left sub-tree (structure of the same type) and
- the right sub-tree (structure of the same type).

Example:

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



To avoid writing the tree every time in the Prolog interpreter, you can „save" the structure as a fact.

Examples:

Code

```
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)))).
...
```

Query

```
% through this query, the T variable unifies with the tree in the tree1/1 fact.
?- tree1(T), operation_on_tree(T, ...).
```

1.2 Tree Traversal

There are 3 ways to traverse a tree depending on the order in which the nodes are processed: inorder, preorder and postorder. For example, the inorder traversal collects the nodes of the left sub-tree, then the current node and the right sub-tree.

Code

```
% left sub-tree, key, right sub-tree (order in append)
inorder(t(K,L,R), List):-
   inorder(L,LL),
   inorder(R,LR),
   append(LL, [K|LR], List).
inorder(nil, []).

% key, left sub-tree, right sub-tree (order in append)
preorder(t(K,L,R), List):-
   preorder(L,LL),
   preorder(R, LR),
   append([K|LL], LR, List).
preorder(nil, []).

% left sub-tree, right sub-tree, key (order in appends)
postorder(t(K,L,R), List):-
   postorder(L,LL),
   postorder(R, LR),
   append(LL, LR,R1),
   append(R1, [K], List).
postorder(nil, []).
```

Observation1. Even though the recursive call of 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 the *append* predicate. Thus, for the case of the *inorder/2* predicate, the nodes on the left sub-tree appear first in the list, then the root node, then the keys in the right sub-tree.

Observation2. Observe that only the position of the current node changes, and thus the only difference between the predicates is where the current node (*K*) is appended.

Follow the execution of:

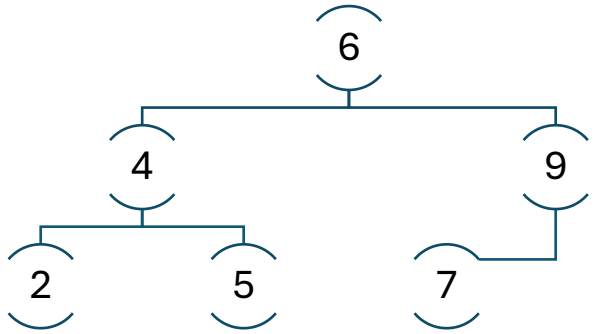
Query

```
?- trace, tree1(T), inorder(T, L).
?- trace, tree1(T), preorder(T, L).
?- trace, tree1(T), postorder(T, L).
```

1.3 „Pretty” Print

We will use the „pretty” print to visualize the correctness of our tree operations. The easiest option of printing a tree is to traverse it in inorder and print each node at an indent (number of spaces) equal to its depth in the tree. The root is always at depth 0, with each node being printed on a separate line.

Example:




```
|          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:

Code

```
% wrapper
pretty_print(T):- pretty_print(T, 0).

% predicate that prints the tree
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).

% print_key/2 prints key K at D tabs from the left margin
% and inserts a new line (through nl)
print_key(K, D):-D>0, !, D1 is D-1, tab(8), print_key(K, D1).
print_key(K, _):-write(K), nl.
```

Follow the execution of:

Query

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

1.4 Search of a Key

The *search_key(Key, T)* predicate checks if a node with key *Key* exists in the tree *T*. Because of the ordering of the keys in a binary search tree, searching for a given key is very efficient. The mathematical recurrence can be formulated as:

$$search(key, T) = \begin{cases} false & T = null \\ true & key = T.root.key \\ search(key, T.left) & key < T.root.key \\ search(key, T.right) & key > T.root.key \end{cases}$$

It can also be sketched as below:

```
if currentNode = null then return -1; //not found
if searchKey = currentNode.key then return 0; //found (clause 1)
else     if searchKey < currentNode.key (clause 2)
                  //search left subtree
then search(searchKey, currentNode.left);
         else (clause 3)
                  //search right subtree
search(searchKey, currentNode.right);
```

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.

In Prolog, it will be written as:

Code

```
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).
```

Follow the execution of:

Query

```
?- trace, tree1(T), search_key(5,T).
?- trace, tree1(T), search_key(8,T).
```

1.5 Insertion of a Key

The first step consists in searching the position in the tree where the node with the given key should be inserted (the node is of leaf-type). 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.

Code

```
insert_key(Key, nil, t(Key, nil, nil)). % insert key in tree
insert_key(Key, t(Key, L, R), t(Key, L, R)):- !. % key already exists
```

```
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).
```

Follow the execution of:

Query
?- tree1(T),pretty_print(T),insert_key(8,T,T1),pretty_print(T1). ?- tree1(T),pretty_print(T),insert_key(5,T,T1),pretty_print(T1). ?- 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).

1.6 Deletion of a Key

In the case of key deletion, we need to search for the node *(clause 4 and 5)* with that key and then we need to ensure that after the removal of that node, the tree remains a Binary Search Tree.

Once the key is found, we distinguish among three situations:

- We have to delete a leaf node (**clause 1 and 2**, when L or R = nil)
- We have to delete a node with one child **(clause 1 and 2)**
- We have to delete a node with both children **(clause 3)**

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).

The first alternative is implemented in the *delete_key*/3 predicate, below:

Code

```
delete_key(Key, t(Key, L, nil), L):- !.
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).

% search for the predecessor node
get_pred(t(Pred, L, nil), Pred, L):-!.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Follow the execution of:

Query

```
?- tree1(T), pretty_print(T), delete_key(5, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), delete_key(9, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), delete_key(6, T, T1), pretty_print(T1).
?- tree1(T), pretty_print(T), insert_key(8, T, T1), pretty_print(T1), delete_key(6, T1,
T2), pretty_print(T2), insert_key(6, T2, T3), pretty_print(T3).
```

1.7 Tree Height

The *height(T,R)* predicate computes the distance between the root and a leaf. The mathematical recurrence can be formulated as:

$$height(T) = \begin{cases} 0 & T = null \\ \max\big(height(T.left), height(T.right)\big) + 1 & otherwise \end{cases}$$

Or it can be formulated as:

- 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{T.Left, T.Right} + 1)

In Prolog, it will be written as:

Code

```
height(nil, 0).
height(t(_, L, R), H):-
    height(L, H1),
    height(R, H2),
    max(H1, H2, H3),
    H is H3+1.

max(A, B, A):-A>B, !.
max(_, B, B).
```

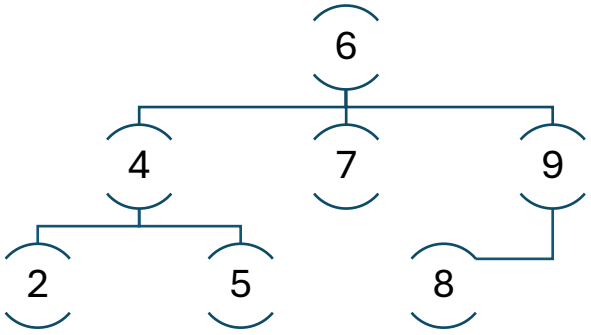
Follow the execution of:

Query

```
?- trace, tree1(T), height(T, H).
?- tree1(T), height(T, H), pretty_print(T), insert_key(8, T, T1), height(T1, H1), pretty_print(T1).
```

1.8 Ternary Trees

In a ternary tree, a node can contain at most 3 children. Although it is not as easy as for binary trees, ordering relations can be established for ternary trees as well. The tree can be represented using a structure of type *t(Key, Left, Middle, Right)*.



Example:

Code

```
ternary_tree(t(6, t(4, t(2, nil, nil, nil), nil, t(5, nil, nil, nil)), t(7, nil, nil, nil), t(9, t(8, nil, nil, nil), nil, nil) ) ).
```

1.8.1 Pretty Print for ternary trees

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 below the node (should be printed after the node is printed), but above the next node at the same depth:

```
|6
|       4
|               2
|               5
|       7
|       9
|               8
```

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.

2 Exercises

Before beginning the exercises, add the following facts which define trees (binary and ternary) into the Prolog script:

Code

```
% Trees:
binary_tree(t(6, t(4,t(2,nil,nil),t(5,nil,nil)), t(9,t(7,nil,nil),nil))).
ternary_tree(t(6, t(4, t(2, nil, nil, nil), nil, t(5, nil, nil, nil)), t(7, nil, nil, nil), t(9, t(8, nil, nil, nil), nil, nil))).
```

1. Write the predicates that iterate over the elements of a ternary tree in:

1.1.*preorder*=Root->Left->Middle->Right
1.2.*inorder*=Left->Root->Middle->Right
1.3.*postorder*=Left->Middle->Right->Root

Query

```
?- ternary_tree(T), ternary_preorder(T, L).
L = [6, 4, 2, 5, 7, 9, 8], T= ...

?- ternary_tree(T), ternary_inorder(T, L).
L = [2, 4, 5, 6, 7, 8, 9], T= ...

?- ternary_tree(T), ternary_postorder(T, L).
L = [2, 5, 4, 7, 8, 9, 6], T= ...
```

2. Write the pretty print predicate called *pretty_print_ternary/1* for a ternary tree.

Query

```
?- ternary_tree(T), pretty_print_ternary(T).
6
    4
        2
        5
    7
    9
        8
T= ... ;
```

3. Write a predicate that computes the height of a ternary tree.

Query
?- ternary_tree(T), ternary_height(T, H). H=3, T= ... ; false.

4. Write a predicate that collects into a list, all the keys of the leaves of a Binary Search Tree.

Query
?- binary_tree(T), leaf_list(T, R). R=[2,5,7], T= ... ; false.

5. Write a predicate that collects into a list, all the keys of the internal nodes (non-leaves) of a Binary Search Tree.

Query
?- binary_tree(T), internal_list(T, R). R = [4, 6, 9], T = ... ; false.

6. Write a predicate that collects into a list, all the nodes from the same depth (inverse of height) in a Binary Tree.

Query
?- binary_tree(T), same_depth(T, 2, R). R = [4, 9], T = ... ; false.

7. Write a predicate that computes the diameter of a Binary Tree.

$$diam(T) = \max\{diam(T.left), diam(T.right), height(T.left) + height(T.right) + 1\}$$

Query
?- binary_tree(T), diam(T, D). D = 5, T = ... ; false.

8. Write a predicate that checks if a binary tree is symmetric. A binary tree is symmetric if the left sub-tree is the mirror of the right sub-tree. We are interested in the structure of the tree, not the values (keys) of the nodes.

Query

```
?- binary_tree(T), symmetric(T).
false.

?- binary_tree(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));
false.
```

9. Rewrite the *delete_key* predicate using the *successor* node (binary search trees).

Query

```
?- binary_tree(T), delete_key(5, T, T1), delete_key_succ(5, T, T2).
T1 = T2, T2 = t(6,t(4,t(2,nil,nil),nil),t(9,t(7,nil,nil),nil)),
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)).

?- binary_tree(T), delete_key(6, T, T1), delete_key_succ(6, T, T2).
T = t(6,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,t(7,nil,nil),nil)),
T1 = t(5,t(4,t(2,nil,nil),nil),t(9,t(7,nil,nil),nil)),
T2 = t(7,t(4,t(2,nil,nil),t(5,nil,nil)),t(9,nil,nil))
```