Université d'Ottawa · University of Ottawa

Faculté de Génie – EECS

CSI2520 : PARADIGMES DE PROGRAMMATION

Tutorat 5

Count the leaves of a binary tree

A leaf is a node with no successors. Write a predicate count_leaves/2 to count them.

% count_leaves(T,N) :- the binary tree T has N leaves

```
% count_leaves(T,N) :- the binary tree T has N leaves

count_leaves(nil,0).
count_leaves(t(_,nil,nil),1).
count_leaves(t(_,L,nil),N) :- L = t(_,_,_),
count_leaves(L,N).
count_leaves(t(_,nil,R),N) :- R = t(_,_,_),
count_leaves(R,N).
count_leaves(t(_,L,R),N) :- L = t(_,_,_), R = t(_,_,_),
   count_leaves(L,NL), count_leaves(R,NR), N is NL +
NR.

% The above solution works in the flow patterns (i,o)
and (i,i)
% without cut and produces a single correct result.
Using a cut
% we can obtain the same result in a much shorter
program, like this:

count_leaves1(nil,0).
count_leaves1(t(_,nil,nil),1) :- !.
count_leaves1(t(_,L,R),N) :-
    count_leaves1(L,NL), count_leaves1(R,NR), N is
NL+NR.
```

Symmetric binary trees

Let us call a binary tree symmetric if you can draw a vertical line through the root node and then the right subtree is the mirror image of the left subtree. Write a predicate symmetric/1 to check whether a given binary tree is symmetric. **Hint**: Write a predicate mirror/2 first to check whether one tree is the mirror image of another. We are only interested in the structure, not in the contents of the nodes.

```
symmetric(nil).
symmetric(t(_,L,R)) :- mirror(L,R).

mirror(nil,nil).
mirror(t(_,L1,R1),t(_,L2,R2)) :- mirror(L1,R2),
mirror(R1,L2).
```

Collect the internal nodes of a binary tree in a list

An internal node of a binary tree has either one or two non-empty successors. Write a predicate internals/2 to collect them in a list.

% internals(T,S) :- S is the list of internal nodes of the binary tree T.

```
internals(nil,[]).
internals(t(_,nil,nil),[]).
internals(t(X,L,nil),[X|S]) :- L = t(_,_,_),
internals(L,S).
internals(t(X,nil,R),[X|S]) :- R = t(_,_,_),
internals(R,S).
internals(t(X,L,R),[X|S]) :- L = t(_,_,_), R =
t(_,_,_),
   internals(L,SL), internals(R,SR), append(SL,SR,S).

% OR

internals1(nil,[]).
internals1(t(_,nil,nil),[]) :- !.
internals1(t(X,L,R),[X|S]) :-
    internals1(L,SL), internals1(R,SR),
append(SL,SR,S).
```

Exercise 1. Lists to BSTs

Define a predicate that builds a binary search tree from an ordered list. Try to make the tree as balanced as possible. Here is how such a predicate could work:

```
?- listToBST([], T).
T = nul.

?- listToBST([1], T).
T = t(1, nul, nul) .

?- listToBST([1, 2], T).
T = t(2, t(1, nul, nul), nul) .

?- listToBST([1, 2, 3, 4, 5, 6, 7], T).
T = t(4, t(2, t(1, nul, nul), t(3, nul, nul)), t(6, t(5, nul, nul), t(7, nul,
nul))) .

?- listToBST([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], T).
T = t(7, t(4, t(2, t(1, nul, nul), t(3, nul, nul)), t(6, t(5, nul, nul),
nul)), t(10, t(9, t(8, nul, nul), nul), t(12, t(11, nul, nul), nul))) .
```

- Hint:
  - find the middle element in the list.
  - **length** (List, LengthOfList).

```
listToBST([],nul).
listToBST (List, t(Root, Tleft, Tright)):-
    length (List, L1), L2 is L1/2,
    findMid (List, L2, Root, Lleft, Lright),
    listToBST (Lleft, Tleft), listToBST (Lright,
Tright).
listToBST([],nul).
listToBST (List, t(Root, Tleft, Tright)):-
    length (List, L1), L2 is L1/2,
    findMid (List, L2, Root, Lleft, Lright),
    listToBST (Lleft, Tleft), listToBST (Lright,
Tright).
```

Search

Explore the different search options given below. Use them to solve the towers of Hanoi problem and observe what happens.

```
% Only smaller disks can go on larger disks
legal(_,[]).
legal(D1,[D2|_]):- D1 < D2.
% Move from first to second pole
transition([[T|L1],L2,L3],[L1,[T|L2],L3]) :- legal(T,L2).
% Move from first to third pole
transition([[T|L1],L2,L3],[L1,L2,[T|L3]]) :- legal(T,L3).
```

```
% Move from second to first pole
transition([L1,[T|L2],L3],[[T|L1],L2,L3]) :- legal(T,L1).
% Move from second to third pole
transition([L1,[T|L2],L3],[L1,L2,[T|L3]]) :- legal(T,L3).
% Move from third pole to first pole
transition([L1,L2,[T|L3]],[[T|L1],L2,L3]) :- legal(T,L1).
% Move from third pole to second pole
transition([L1,L2,[T|L3]],[L1,[T|L2],L3]) :- legal(T,L2).

% final goal
goalState([[],[],_]).

% helper routine
showPath([]).
showPath([Node|Path]) :- showPath(Path), nl, write(Node).
```

Solution 1. Standard prolog DFS

The following program uses Prolog's depth first search strategy with the states and transitions defined above.

```
search1(Node,[Node]) :- goalState(Node).
search1(Node,[Node|Solution]) :- transition(Node,NNode),
        search1(NNode,Solution).
```

Solution 2. Prolog DFS without Loops

The following program uses again Prolog's depth first search strategy but avoids returning to previously visited states (or search nodes).

```
search2(Node,Solution) :- depthSearch([], Node, Solution),
        showPath(Solution).
depthSearch(Path, Node, [Node | Path]) :- goalState(Node).
depthSearch(Path, Node, Sol) :-
        transition(Node, NNode),
        \+member(NNode,Path),
        depthSearch([NNode | Path], NNode, Sol).
```

Solution 3. Prolog DFS to a Maximum Depth

The following program uses again Prolog's depth first search strategy but only backtracks until a certain depth in the search tree is reached.

```
search3(Node,[Node],_) :- goalState(Node).
search3(Node,[Node|Solution],Pmax) :- Pmax>0,
      transition(Node,NextNode),
      Pmax1 is Pmax-1,
      search3(NextNode,Solution,Pmax1),
      write(NextNode), nl.
```

Solution 4. Use BFS

The following program implements breadth first search with the help of the `bagof` predicate.

```
breadthSearch([[Node | Path] | _],[Node | Path]):- goalState(Node).
breadthSearch([Path | Paths], Solution):-
        transitionList(Path,NPaths),
        append(Paths,NPaths,Paths1),
        breadthSearch(Paths1,Solution).

transitionList([Node | Path],NPaths):-
        bagof([NNode,Node | Path],
              (transition(Node,NNode),
               \+member(NNode,[Node | Path])),
              NPaths),
        !.
transitionList(_,[]).


search4(Root, Solution):- breadthSearch([[Root]],Solution),
        showPath(Solution).
```