**Harry Langford**
hjel2@cam.ac.uk



# 1 1997 Paper 12 Question 8

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y1997p12q8.pdf

The *next-highest member* of a list of integers is the second-largest member of the list. For example, for the list `[1, 4, 1, 5, 2]`, the next-highest member is `4`.

Write a Prolog program to find the next-highest member of a list of integers. For example, the goal `nexthi([1, 4, 1, 5, 2], X)` should initialise `X` to `4`. Your program may assume that the next largest member is not repeated in the list. The goal should fail if the next-highest member does not exist.

```
nexthi(X, L) :- member(X, L), count(X, L, 1).
count(X, [], 0).
count(X, [H|T], N) :- H > X, N is M + 1, count(X, T, M).
```

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
nexthi([X,Y|T], Lo) :- max(X, Y, X), largest2(T, X, Y, _, Lo).
nexthi([X,Y|T], Lo) :- max(X, Y, Y), largest2(T, Y, X, _, Lo).
largest2([], H, L, H, L).
largest2([X|Z], H, L, Hr, Lr) :- max(L, X, L), largest2(Z, H, L, Hr, Lr).
largest2([X|Z], H, L, Hr, Lr) :- max(L, X, X), max(H, X, H), largest2(Z, H, X, Hr, Lr).
largest2([X|Z], H, _, Hr, Lr) :- max(H, X, X), largest2(Z, X, H, Hr, Lr).
```

Yep — rather imperative thinking but it works!

# 2 1996 Paper 5 Question 7

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y1996p5q7.pdf

An *ordered integer binary search tree* (or OIBS tree) is either empty or a tuple $(T, N, U)$, where $T$ and $U$ are also OIBS trees and $N$ is an integer. Every node in $T$ has a value less than $N$, which in turn is less than the value of every node in $U$.

(a) Give two Prolog terms which are suitable for representing an empty OIBS tree and a node in the OIBS tree respectively.

```
leaf.
branch(L, N, R).
```

(b) Define a prolog procedure `insert(Item, T, NT)`, where `Item` is an integer being inserted into OIBS tree `T`, producing an OIBS tree `NT`. If `Item` is already present in `T`, then `NT` equals `T`.

```
insert(Item, leaf, branch(leaf, Item, leaf)).
insert(Item, branch(L, N, R), branch(L, N, R)) :- Item is N.
insert(Item, branch(L, N, R), branch(LT, N, R)) :- Item < N, insert(Item, L, LT).
insert(Item, branch(L, N, R), branch(L, N, RT)) :- Item > N, insert(Item, R, RT).
```

(c) Define a Prolog procedure `lookup(Item, T)`, where `Item` is to be looked for in OIBS tree `T`. A lookup goal will succeed if `Item` is found, or fail otherwise.

```
lookup(Item, branch(_, N, _)) :- Item is N.
lookup(Item, branch(L, N, _)) :- Item < N, lookup(Item, L).
lookup(Item, branch(_, N, R)) :- Item > N, lookup(Item, R).
```

does this work backwards — can you instantiate Item to everything that can be looked up in a specified tree?

# 3 Permutations

Write a prolog program for generating permutations of a list.

```
take([X|T], X, T).
take([H|Tl], X, [H|Tr]) :- take(Tl, X, Tr).
perm([], []).
perm([H|T], P) :- perm(T, L), take(P, H, L).
```

A "good predicate doesn't have an input or an output -- but is a mapping (relation) between its arguments"

Prolog only has one algorithm -- brute force search -- what your relations do is define HOW to search the search tree.

"Efficiency" in prolog can be
- how large the search tree is compared to the number of answers.
- time to get the first answer
    How have you described the search case -- are you searching useless corners first?

Generate and test paradigm is really usefiu! Do it!

Prolog relations are NOT functions -- they are inductive definitions of a set.

Prolog optimisations include memoisation (imagine @lru_cache at the front of every relation)

If you want to represent a tree data structure, you have to represent it as membership of a predicate -- data structures which exist are members of the predicate -- and trees which don't exist are not members of the predicate.

COMMENT PROLOG CODE FFS

In general, to make something work in both ways, use generate-and-test (to ensure arguments are instantiated)

There exist extra-logical predicates which say "if X is a variable"

Last call optimisation only happens on the LAST fact in a relation

If you cannot unify arguments to clauses then prolog will hopefully be able to tell they are orthogonal (semantically it's a tail call -- but the compiler may not realise)

Prolog uses the closed world assumption -- the absence of lines of code has meaning -- if you add code it can break it!
    This is often worth a comment when you write the code i.e. "% take ([], _, _) :- fail" [note this is commented!]

With Prolog you can "thread things together" -- just do two functions at the same time (see page 1 of live supervisions)
    Think -- you can use values you don't actually know yet!

The time complexity of unifying with an uninitialised variable is GUARANTEED to be O(1) irrelevant of how many occurences there are
    i.e unified variables become automatically dereferenced pointers