Основни предикати със списъци

[Head|Tail]

I Member

```
member(X, [X|_]).
member(X, [_|T]) :- member(X,T).
```

```
member(X,L) :- remove(X,L,_).
```

```
member(X,L) :- append(_, [X|_], L).
```

II append

```
append([], Y, Y).
append([X|T], Y, [X|Result]) :- append(T, Y, Result).
```

III Insert

```
insert(X,L,Result) :- append(A,B,L), append(A,[X|B],Result).
```

```
insert(X,L,Result) :- remove(X, Result, N).
```

IV Remove.

```
remove(X, [X|T], T).
remove(X, [H|T], [H|N]) :- remove(X,T,N).
```

V Subset (с коректно наредени елементи).

```
sub([],[]).
sub([X|T], Res) :- sub(T, Res).
sub([X|T], [X|Res]) :- sub(T, Res).
```

VI subset (с пермутации)

```
psub(L, Res) :- sub(L,Q), perm(Q,Res).
```

VII Permutations

```
perm([X|T], Result) :- perm(T,R), insert(X,R,Res).
```

VIII First element

```
first(F, [F|_]).
```

IX Second element

- ```
  second(S, [_,S|_]).
  ```
- ```
  second(S, [H|T]):- first(S,T).
  ```

X Last element

- ```
  last(X,L):- append(_,[X],L).
  ```
- ```
  last(X, [_|T]):- last(X,T).
  last(X, [X]).
  ```

XI Prefix

```
prefix(P,L):- append(P,_,L).
```

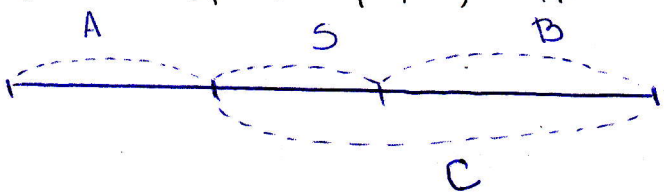
```
prefix([],_).
prefix([H|T],[H|Q]):- prefix(T,Q).
```

XII Suffix

```
suffix(S,L):- append(_,S,L).
```

XIII Sublist

```
sublist(S,L):- append(_,C,L), append(S,_,C).
```



XIV Is subset?

```
isSubset(S,L):-not(member(X,S), not(member(X,L))).
```

XV equal lists

```
equal(L1,L2):-subset(L1,L2), issubset(L2,L1).
```

XVI intersection

```
intersection(X,A,B):- member(X,A), member(X,B).
```

XVII Union

```
union(X,A,B):- member(X,A); member(X,B)
```

XVIII
```
sub(X,A,B):- member(X,A), notl member(X,B)).
```

XIX Reverse

-
```
reverse([],[])
reverse(R,[H,L]):- reverse(RL,L), append(RL,[H],R).
```

-
```
reverse(R,L):- reverse([],L,R).
reverse(S,[],S).
reverse(S,[H|T],R):- reverse([H|S],T,R).
```

XX Length

```
length([],0).
length([H|T],N):- length(T,M), N is M+1.
```

XXI Nth element

```
nth([X|_],0,X).
nth([_|T],N,X):- nth(T,M,X), N is M+1.
```

XXII Sorting

1. Insertion sort
```
sort([],[]).
sort([H|Q],L):- sort(Q,M), sort_in(H,M,L).

sort_in(X,[],[X]).
sort_in(X,[H|T],[X,H|T]):- X=<H.
sort_in(X,[H|Q],[H|L]):- H<X, sort_in(X,Q,L).
```

2. Selection sort

-
```
sort([],[]).
sort(L,[H|Q]):- min(H,L,M), sort(M,Q).

min(X,[X],[]).
min(H,[H|T],T):- min(X,T,_), H=<X.
min(X,[H|Q],[H|L]):- min(X,Q,L), H>X.
```

-
```
sort([],[]).
sort([M|S],L):- min(M,L), remove(M,L,N), sort(N,S).

min(M,[M]).
min(M,[H|T]):- min(M,T), min2(M,H,M).
min2(A,A,B):- A=<B.
min2(B,A,B):- not(A=<B).
```

3. Quick Sort

```
quicksort([],[]).
quicksort([X|Xs],Res):- part(X, Xs, L, R), quicksort(L,SL), quicksort(R,SR),
                        append(SL, [X|SR], Res).

part(_,[],[],[]).
part(X, [H|T], [H|L], R):- H<X, part(X, T, L, R).
part(X, [H|T], L, [H|R]):- H>=X, part(X, T, L, R).
```

4. Is sorted?

```
sorted(X):- not(append(_, [A,B|_], X), A>B).
// sorted      increasingly
```

XXIII Sum of elements

```
sum([],0).
sum([H|T],N):- sum(T,M), N is M+H.
```

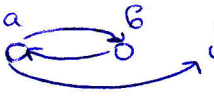
Графи.

Нека G е представен със списък от ребра
[[v0, u0], [v1, u1], ..., [vn, un]]

I Път в граф

```
path(G, X, Y):- member([X,Y], G).
path(G, X, Y):- member([X, Z], G), path(G, Z, Y).
```

Проблем със зацикляне



G = [[a,b], [a,c], [b,a]]
безкраен.

⇓

visited

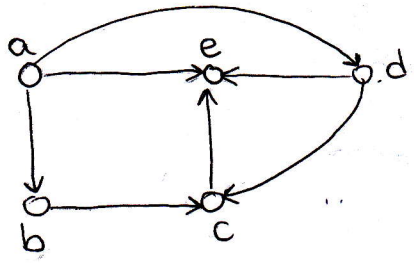
```
path(G, X, Y):- path(G, X, Y, [X], _).

path(G, X, X, [X], [X]).
path(G, X, Y, V, [X|P]):- member([X, Z], G), X \= Y,
                          not(member(Z, V)),
                          path(G, Z, Y, [Z|V], P).
```

II Цикъл в граф?

```
cycle(G):- member([X,Y], G), path(G, Y, X, [], _).

cycle(G, [X|T]):- member([X,Y], G), path(G, Y, X, [], T).
```

III DFS

```
dfs(G, X, T):- dfs(G, [X], [], T).

next(G, U, S, F, V):- member([U,V], G),
                      not(member(V, S)),
                      not(member(V, F)).

dfs(G, [], _, []).                          visited
dfs(G, [S|Ss], F, [[S,N]|R]):-
        стек        next(G, S, [S|Ss], F, N),
                    dfs(G, [N,S|Ss], F, R).

dfs(G, [S|Ss], F, R):- not(next(G, S, [S|Ss], F, _)),
                       dfs(G, Ss, [S|F], R).
```



G = [[a,b], [a,e], [a,d], [b,c], [c,e], [d,c], [d,e], [e]]

dfs(G, a, T) :- dfs(G, [a], [], T)

dfs(G, [b, a], [], R) R = ([a,b] | R)

dfs(G, [c, b, a], [], R) R = ([b,c], [a,b] | R)

dfs(G, [e, c, b, a], [], R) R = ([c,e], [b,c], [a,b] | R)

dfs(G, [c, b, a], [e], R)

dfs(G, [b, a], [c, e], R)

dfs(G, [a], [b, c, e], R)

dfs(G, [d, a], [b, c, e], R) R = ([a,d], [c,e], [b,c], [a,b] | R)

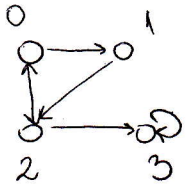
dfs(G, [a], [d, b, c, e], R)

dfs(G, [], [a, d, b, c, e], R)

T = [a,b], [b,c], [d,e], [a,d]

DFS

*using stack



1. Push 0 on the stack
2. Pop the last element of the stack
3. See which elements it's connected to.
4. Put them on the stack (1, 2).
5 Remove 2 from the stack.
6. See which element it's connected to (0, 3).
7. Put 3 on the stack. (0 is already visited).
8. Remove 3 from the stack. // Just 1 left.
9. Remove 1 from the stack
10. Done.

Stack: 0
Visited: -

Stack: 1, 2
Visited: 0

Stack: 1, 3
Visited: 0, 2

Stack: 1
Visited: 0, 2, 3

Stack -
Visited: 0, 2, 3, 1

So, we need:
1) Stack
2) Visited
3) neighbours which are neither on the stack nor in the visited.

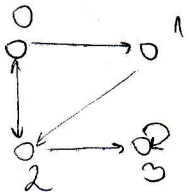
Prolog:

```
neighbour(G, U, V, Stack, Visited):-
    member([U,V], G),
    not(member(V, stack)),
    not(member(V, Visited)).
```

```
dfs(G, X, T):- dfs(G, [X], [], T).
dfs(G, [], _, []).

dfs(G, [Current|Rest], Visited, ([Current, Next]|R):-
        next(G, Current, Next, [Current|Rest], Visited),
        dfs(G, [Next, Current|Rest], Visited, R).
dfs(G, [Current|Rest], Visited, R):-
        not(next(G, Current, _, [Current|Rest], Visited),
        dfs(G, Rest, [Current|Visited], R).
```

BFS

* uses a queue



Queue: 0
Visited: -

Queue: 1, 2
Visited: 0

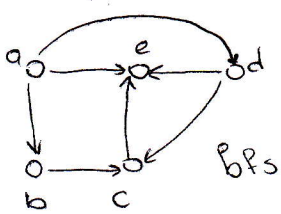
Queue: 2
Visited: 0, 1

Queue: 3
Visited: 0, 1, 2

Queue: -
Visited: 0, 1, 2, 3

Prolog:

```
bfs(G, X, T):- bfs(G, [X], [], T).
bfs(G, [], _, []).
bfs(G, [Current|Queue], Visited, [[Current|Next]|R]):-
    next(G, Current, Next, [current|Queue], Visited),
    append([Current|Queue], [Next], NewQueue),
    bfs(G, NewQueue, Visited, R).

bfs(G, [Current|Queue], Visited, R):-
    not(next(G, Current, _, [current|Queue], Visited),
    bfs(G, Queue, [Current|Visited], R).
```



G = [[a,b], [a,e], [a,d], [b,c], [c,e], [d,c], [d,e]]

bfs(G, a, T) :- bfs(G, [a], [], T)

bfs(G, [a,b], [], R=[[a,b]])

bfs(G, [a,b,e], [], [[a,b],[a,e]])

bfs(G, [a,b,e,d], [], [[a,b], [a,e], [a,d]])

bfs(G, [b,e,d], [a], [[a,b], [a,e], [a,d]])

bfs(G, [b,e,d,c], [a], [[a,b], [a,e], [a,d], [b,c]])

bfs(G, [e,d,c], [b,a], R)

bfs(G, [d,c], [e,b,a], R)

bfs(G, [c], [d,e,b,a], R)

bfs(G, [], [c,d,e,b,a], R)

Работа с числа и аритметични операции

I Генератор на естествени числа

```
nat(0).
nat(X):- nat(Y), X is Y+1.
```

Не става за разпознавател

Операция за присвояване

II Най-голям общ делител

```
gcd(X,Y,Y):- X mod Y=:=0.
gcd(X,Y,Z):- R is X mod Y, gcd(Y,R,Z).
```

III Генератор на естествени четни числа

```
evenNumber(0).
evenNumber(X):- evenNumber(Y), X is Y+2.
```

IV Разпознавател на естествени четни числа

```
evenNumber(0).
evenNumber(X):- X>=1, Y is X-2, evenNumber(Y).
```

V Разпознавател / Генератор на естествени четни числа

```
evenNumber(X):- nat(X), X mod 2=:=0.
```

VI Фибоначи

```
fibonacci(0,1).
fibonacci(A,B):- fibonacci(B,C), A is B+C.
fibonacci(X):- fibonacci(X,_).

fibonacci(X,X,_).
fibonacci(X,A,B):- C is A+B, fibonacci(X,C,A).
fibonacci(X):- fibonacci(X,0,1).
```

VII Between

```
between(A,B,A):- A=<B.
between(A,B,X):- A<B, C is A+1, between(C,B,X).
```

VIII Prime

```
isPrime(N):- N1 is N//2, not(between(2,N1,X), N mod X=:=0).
prime(N):- nat(X), isPrime(N).
```

IX

Двойки естествени числа
pair(X,Y) :- nat(N), between(0,N,X), Y is N-X.

X

Генератор на цели числа

- z(Y) :- nat(X), s(X,Y).

s(X,X).
s(X,Y) :- X>0, Y is -X.

- z(X) :- nat(Y), decode(Y,X).

decode(X,Y) :- X mod 2 =:= 0,
Y is -X//2.
decode(X,Y) :- X mod 2 =:= 1
Y is X//2+1

0 1 -1 2 -2 3 -3
0 1 2 3 4 5 6

Задачи

1. Да се реализира предикат, който по дадено число, създава списък с разлагането му на прости делители във възходящ ред.

Р-ие:

```
primeDivisor(N,P):- N1 is N//2,
                    between(2,N1,P),
                    N mod P =:= 0, isPrime(P).
minPrimeDivisor(N,P):- primeDivisor(N,P),
                       not(primeDivisor(N,Q), Q<P).

decomposition(0,[]).
decomposition(1,[]).
decomposition(N,[P|T]):- minPrimeDivisor(N,P),
                         N1 is N//P,
                         decomposition(N1,T).
```

2. Да се реализира предикат, който по дадено число създава списък с простите делители (във възходящ ред) и техните степени от каноничното му представяне.

Р-ие:

```
degrees(0,[]).
degrees(1,[]).
degrees(N,D):- minPrimeDivisor(N,P), N1 is N//P,
               degrees(N1,T),
               add(T,P,D).

add(T,P,D):- append(B,[[P,C]|E],T), C is C+1,
             append(B,[[P,C]|E],D).
add(T,P,[[P,1]|T]):- not(member([P,_],T)).
```

3. Да се генерират всички числа между 0 и 1000, които са от следния вид $3*(n+2)^5*(k+3)^2$.

Р-ие:

```
format(X):- between(0,1000,Y), between(0,1000,Z),
            X is 3*(Y+2)^5*(Z+3)^2, X>=0, X=<1000.
```

Зад 4. Да се генерират всички точки с цели координати, които се намират във вътрешността на елипса с полуоси 113 и 71.

Р-ие:

```
elipse(X,Y):- between(-113,113,X),
              between(-71,71,Y),
              (X*X)/(113*113) + (Y*Y)//(71*71) < 1
```

// е целочислено деление

5//2 = 2 5/2 = 2,5

-5-

Р-ие

Предикат, генериращ K числа със сума S.

```
genKS(1,S,[S]).
genKS(K,S,[X|T]):- K>1, between(0,S,X), K1 is K-1,
                   S1 is S-X, genKS(K1,S1,T).
```

а) предикат, генериращ всички четворки естествени числа

```
gen4(X):- nat(N), genKS(4,N,X).
```

б) предикат, генериращ всички двойки естествени числа

```
pair(X,Y):- nat(N), genKS(2,N,[X,Y]).
```

в) предикат, генериращ всички крайни редици от естествени числа

```
gen_all([]).
gen_all(X):- nat(N), between(1,N,K), S is N-K,
             genKS(K,S,X).
```

6 $\{a_n\}_0^\infty$ $a_0=1$, $a_n = 2a_{n-1}+n$

$\{b_n\}_0^\infty$ $b_0=1$, $b_n = 3b_{n-1} + n^2 - 1$

Да се напише предикат p(N), генериращ всички числа от вида $a_i + b_j$.

Р-ие: •

```
a(0,1)
a(N,X):- N>0, N1 is N-1, a(N1,Y), X is 2*N1+N.
b(0,1)
b(N,X):- N>0, N1 is N-1, b(N1,Y), X is 3*N1 + N*N-1.

p(N):- nat(X), between(0,X,I), J is X-I, a(I,A), b(J,B),
       N is A+B.
```

•

```
s(_,[A|As],[B|Bs],Res):- (member(X,Bs), Res is A+X);
                         (member(X,[A|As]), Res is B+X).
s(N,[A|As],[B|Bs],R):- NA is 2*A+N,
                       NB is 3*B + N*N-1,
                       N1 is N+1
                       s(N1,[NA,A|As],[NB,B|Bs],R).

p(N):- s(1,[1],[1],N).
```

// неефективно

7. Дефиниция на Δ-списък:
[] е Δ-списък
Ако A и B са Δ списъци, то [A,[A,B]] също е Δ списък.
Да се напише предикат, генериращ всички Δ списъци.

[[], [[], [[], [[], []]]]]
A A B

rang([]) = 1
rang([A, [A, B]]) = rang(A) + rang(B)

Р-ие:
```
r([],1).
r([A,[A,B]],N):- N>1, M is N-1, between(1,M,RA),
                 RB is N-RA, r(A,RA), r(B,RB).

alpha(L):- nat(N), r(L,N).
```

[[], [[], []]] r=2
[] r=1 A
[] r=1 B

8. Да се дефинира предикат s(M,L), който генерира в M всички списъци с елементи от L.
L=[1,2]
M=[1,1,1,2]
M=[2,1,2,1,2].

Р-ие:
```
s([],L).
s([H|T],L):- s(T,L), member(H,L).
```

```
s(M,L):- nat(N), s(N,M,L).
s(0,[],L).
s(N,[H|T],L):- N>0, member(H,L), N1 is N-1, s(N1,T,L).
```

9. Дефиниция на двоично дърво:
- [] е дърво.
- Ако A и B са дървета, [A,B] също е дърво.

Да се напише предикат, генериращ всички дървета.

Р-ие:
```
t(T):- nat(N), t(N,T).
t(0,[]).
t(N,[A,B]):- N>0, N1 is N-1, between(0,N1,NA), NB is N-NA,
             t(NA,A), t(NB,B).
```

10. Да се напише предикат st(G,T), който по даден граф G=[V,E], генерира покриващо дърво T=[V,E1]

Р-ие:
```
st([V,E],[V,E1]):- V=[X|V1], st([V,E],[X],V1,E1).
st(G,V,[],[]).
st(G,V,N,[[X,Y]|R]):- edge(G,X,Y), member(X,V), member(Y,N),
                      delete(Y,N,N1), st(G,[Y|V],N1,R).
```
посетени ↓ Непосетени

8 Нека G е представен като списък от ребра, т.е
G = { [A,B] | има ребро от A до B }.
Да се дефинира предикат p(G), който по даден ориентиран граф G без изолирани върхове проверява дали върховете на G могат така да се оцветят в 2 цвята, че никои три ребра да не образуват едноцветен триъгълник.

P-ие:

```
% vertices(E,V) - gathers into V all vertices from list of edges E
vertices([],[]).
vertices([[A,B]|Tail],V):-
        vertices(Tail,TV), add(A,TV,VA), add(B,VA,V).

% add(X,L,Res) - adds X in L if it's not already stored.
add(X,L,L):- member(X,L).
add(X,L,[X|L]):- not(member(X,L)).

% subset(L,S)
subset(_,[]).
subset([H|T],S):- subset(T,S).
subset([H|T],[H|S]):- subset(T,S).

% triangle(G,A,B,C):- find 3 vertices forming a triangle.
triangle(G,A,B,C):-
        member(G,[A,B]), member(G,[B,C]), member(G,[C,A]),
        not(A=B), not(B=C), not(C=A).

% sameColour(A,B,C,Coloured) - determines if A,B,C are the same
                               colour.
sameColour(A,B,C,Coloured):- member(Coloured,A),
                             member(Coloured,B),
                             member(Coloured,C).

sameColour(A,B,C,Coloured):- not(member(Coloured,A)),
                             not(member(Coloured,B)),
                             not(member(Coloured,C)).

% solution.
p(G):- vertices(G,V), subset(V,Colored),
       not(triangle(G,A,B,C), sameColor(A,B,C,Colored)).
```

9 Нека G е граф без изолирани върхове, представен чрез списък от ребра.
Да се дефинира предикат p(G), който по даден ориентиран граф G проверява дали върховете на G могат така да се оцветят в два цвята, че всяко ребро да свързва различноцветни върхове

Р-ие:
```
p(G):- vertices(G,V), subset(V,Colored)
       not(member([A,B], G), sameColor(A,B,Colored)).

% sameColour(A,B,Coloured)

sameColour(A,B,Coloured):- member(Coloured,A),
                           member(Coloured,B).
sameColour(A,B,Coloured):- not(member(Coloured,A)),
                           not(member(Coloured,B)).
```

10
```
?- flatten([a,[b,[c,d],e]],X)
   X=[a,b,c,d,e]
```

Р-ие:
```
flatten(X,[X]):- not(is_list(X)).
flatten([],[]).
flatten([H|T],Zs):- flatten(H,HA), flatten(T,TA), append(HA,TA,Zs).

is_list([]).
is_list([_|T]):- is_list(T).
```

11. Да се дефинира предикат премахващ последователните повтарящи се елементи.

Р-ие:
```
compress([],[]).
compress([X],[X]).
compress([X,Y|T],[X|Zs]):- X\=Y, compress([Y|T],Zs).
compress([X,X|T],Zs):- compress([X|T],Zs).
```

12. Да се напише предикат, който групира последователните повтарящи елементи на листа в подлисти.

```
?- pack([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X)
   X=[[a,a,a,a],[b],[c,c],[a,a],[d],[e,e,e,e]]
```

Р-ие:
```
pack([],[]).
pack([X|Xs],[Z|Zs]):- transfer(X,Xs,Ys,Z), pack(Ys,Zs).

transfer(X,[],[],[X]).
transfer(X,[Y|Ys],[Y|Ys],[X]):- X\=Y.
transfer(X,[X|Xs],Ys,[X|Zs]):- transfer(X,Xs,Ys,Zs).
```

13: Run-length encoding of a list

```
?- encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X)
   X = ([4,a],[1,b],[2,c],[2,a],[1,d],[4,e]).
```

P-ue:

```
encode(L1,L2):- pack(L1,L), transform(L,L2).
transform([],[]).
transform([[X|Xs]|T],[[N,X]|Z]):- length([X|Xs],N), transform(T,Z).
```

14 Modified run-length encoding

P-ue: •

```
transform([],[]).
transform([[X|Xs]|T],[[N,X]|Z]):-length([X|Xs],N), N>1, transform(T,Z).
transform([[X|[]]|T],[X|Z]):- transform(T,Z).

encode_mod(L1,L2):- pack(L1,L), transform(L,L2).
```

•

```
encode_mod(L1,L2):- encode(L1,L), strip(L,L2).
strip([],[]).
strip([[1,X]|T],[X|Z]):- strip(T,Z).
strip([[N,X]|T],[[N,X]|Z]):- N>1, strip(T,Z).
```

15 Decode a run-length encoded list.

P-ue:

```
decode([],[]).
decode([X|Ys],[X|Zs]):- not(is_list(X)), decode(Ys,Zs).
decode([[1,X]|Ys],[X|Zs]):- decode(Ys,Zs).
decode([[N,X]|Ys],[X|Zs]):- N>1, N1 is N-1, decode([[N1,X]|Ys],Zs).
```

16 Direct run-length encoding

P-ue:

```
encode_direct([],[]).
encode_direct([X|Xs],[Z|Zs]):- count(X,Xs,Ys,1,Z),
                               encode_direct(Ys,Zs).

count(X,[],[],1,X).
count(X,[],[],N,[N,X]):- N>1.
count(X,[Y|Ys],[Y|Ys],1,X):- X\=Y.
count(X,[Y|Ys],[Y|Ys],N,[N,X]):- N>1, X\=Y.
count(X,[X|Xs],Ys,K,T):- K1 is K+1, count(X,Xs,Ys,K1,T).
```

Ys is the list that remains from the list Xs when all the leading copies are removed.
T is the term [N,X], where N is K plus the number of Xs that can be removed from Xs.
In the case of N=1, T is X instead of [1,X].

17. Duplicate the elements of a list.

P-ue:
```
duplicate([],[]).
duplicate([H|T],[H,H|X]):- duplicate(T,X).
```

18: Duplicate the elements of a list a given number of times
% duplicate(L,N,L1).

P-ue:
```
duplicate(L,N,L1):- duplicate(L,N,L1,N).

duplicate([],_,[],_).
duplicate([_|Xs],N,Ys,0):- duplicate(Xs,N,Ys,N).
duplicate([X|Xs],N,[X|Ys],K):- K>0, K1 is K-1,
                               duplicate([X|Xs],N,Ys,K1).
```

19: Drop every n'th element of a list.

P-ue:
```
drop(L,N,L1):- drop(L,N,L1,N).
drop([],_,[],_).
drop([_|Xs],N,Ys,1):- drop(Xs,N,Ys,N).
drop([X|Xs],N,[X|Ys],K):- K>1, K1 is K-1, drop(Xs,N,Ys,K1).
```

20 Split a list into two parts - the length of the first part is given.
// Do not use any predefined predicates
% split(L,N,L1,L2).

P-ue:
```
split([L,0,[],L]).
split([X|Xs],N,[X|Ys],Zs):- N>0, N1 is N-1, split(Xs,N1,Ys,Zs).
```

21 Extract a slice from a list
Given L indices, I and K, the slice is the list containing the elements between the I'th and K'th element of the original list
Start counting the elements with 1.

```
?- slice([a,b,c,d,e,f,g,h,i,k],3,7,L)
L = [c,d,e,f,g]
```

P-ue:
```
slice([X|_],1,1,[X]).
slice([X|Xs],1,K,[X|T]):- K>1, K1 is K-1, slice(Xs,1,K1,T).
slice([_|Xs],I,K,Ys):- I>1, I1 is I-1, K1 is K-1,
                       slice(Xs,I1,K1,Ys).
```

22 Rotate a list N places to the left

rotate ([a,b,c,d,e,f,g,h], 3, [d,e,f,g,h,a,b,c]).
rotate ([a,b,c,d,e,f,g,h], -2, [g,h,a,b,c,d,e,f]).

P-ue:

```
rotate(L1, N, L2) :- N >= 0, length(L1,N1), N2 is N mod N1,
                     rotate_left(L1, N2, L2).

rotate_left(L, 0, L).
rotate_left(L, N, L2) :- split(L, N, A, B), append(B, A, L2).
```

23 Remove the K'th element of the list.

% remove_at(X, L, K, Res).

P-ue:

```
remove_at(X, [X|T], 1, T).
remove_at(X, [H|T], K, [H|Res]) :- K > 1, K1 is K-1,
                                   remove_at(X, T, K1, Res).
```

24: Insert an element at a given position into a list.

% insert_at(X, L, K, R).

P-ue:

```
insert_at(X, L, K, R) :- remove_at(X, R, K, L).
```

25: Create a list containing all integers within a given range.

% range(A, B, L)

P-ue:

```
range(A, A, [A]).
range(A, B, [A|Res]) :- A < B, A1 is A+1, range(A1, B, Res).
```

26: Extract a given number of randomly selected elements from a list.

? - rnd_select([a,b,c,d,e,f,g,h], 3, L).
L = [e,d,a].

P-ue:

```
rnd_select([], 0, []).
rnd_select(Xs, N, [X|Zs]) :- N > 0, length(Xs, L), I is (random(L)+1),
                             remove_at(X, Xs, I, Ys), N1 is N-1,
                             rnd_select(Ys, N1, Zs).
```

built-in

27. Draw N different numbers from the set 1..M

% lotto(N, M, L)

P-ue:

```
lotto(N, M, L) :- range(1, M, L1), rnd_select(L1, N, L).
```

28: Generate a random permutation of the elements of a list.

% rnd_perm(L1, L2).

P-ue 25:

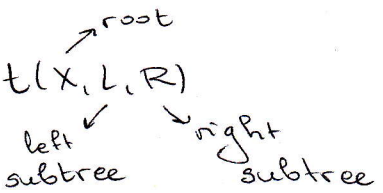
```
rnd_perm(L1, L2) :- length(L1, N), rnd_select(L1, N, L2).
```

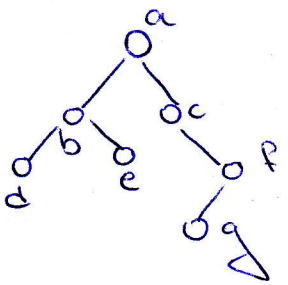
29. A list of prime numbers.

```
%prime_list(A,B,L)

p_list(A,B,[]):- A>B.
p_list(A,B,[A|L]):-.is_prime(A), next(A,AA), p_list(AA,B,L).
p_list(A,B,L) :-next(A,AA), p_list(AA,B,L).

next(2,3).
next(A,AA):- AA is A+2.
```

54. - - - -

Binary tree





TA=t(a,t(b,t(d,nil,nil),t(e,nil,nil)),t(c,nil,t(f,t(g,nil,nil),nil)))

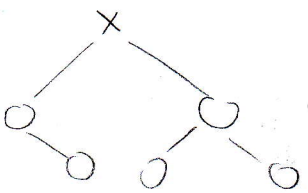
30 Check whether a given term represents a tree

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

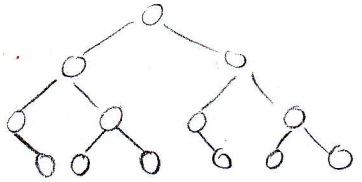
31 Construct completely balanced binary tree

```
cbal_tree(0,[]).
cbal_tree(N,t(_,L,R)):- N>0, NO is N-1, NA is NO//2, N2 is NO-NA,
                        distrib(NA,N2,NL,NR),
                        cbal_tree(NL,L), cbal_tree(NR,R).

distrib(N,N,N,N).
distrib(N1,N2,N1,N2):- N1\=N2.
distrib(N1,N2,N2,N1):- N1\=N2.
```

cbal_tree(6,t(x,L,R)).



cbal_tree(7,t(x,L,R))



32. Generate all completely balanced tree

P-ue
```
tree(0, []).
tree(N, [x,L,R]):- N>0  N0 is N-1, N1 is N0//2, N2 is N0-N1,
                    distribute(N1,N2, NL, NR), tree(NL, L), tree(NR,R).

distribute(N,N,N,N).
distribute(N1,N2, X, Y):- N1 = N2,
                    ((X is N1, Y is N2); (Y is N1, X is N2)).

tree([x,L,R]):- nat(N), tree(N, [x,L,R]).
```

33 Let us call a binary tree symmetric if you can draw a vertical line through the node and then the right subtree is the mirror image of the left subtree.
Write a predicate to check if a given binary tree is symmetric.

```
symmetric([]).
symmetric([_,L,R]):- mirror(L,R).

mirror([],[]).
mirror([_,L1,R1], [_,L2,R2]):- mirror(L1,L2), mirror(R1,R2).
```

34 Construct a BST from a list of integers

```
bst([],[]).
bst([H|L], T):- bst(L,T1), insert(H,T1,T).

insert(X, [], [X,[],[]]).
insert(X, [T,L,R], [T,L1,R]):- X =< T, insert(X, L, L1).
insert(X,[T,L,R], [T,L,R1]):- X>T, insert(X,R,R1).
```

35 Make a list from a BST.

```
ltr([],[]).
ltr([T,L,R], New):- ltr(L,L1), ltr(R,R1),
                    append(L1,[T|R1], New).
```

36. Generate all balanced symmetric trees.

```
symtree(T):- tree(T), symmetric(T).
```

37. Generate height-balanced tree

```
htree([_,L,R]):-nat(N), htree(N,[_,L,R]).
htree(0,nil).
htree(1,[root,nil,nil]).
htree(N,[root,L,R]):- N>1, N1 is N-1, N2 is N-2,
                      distribute(N1,N2,NL,NR),
                      htree(NL,L), htree(NR,R).

distribute(N,_,N,N).
distribute(N1,N2,X,Y):-((X is N1, Y is N2);(X is N2, Y is N1)).
```

38 Count the leaves of a binary tree.

```
count_leaves(nil,0).
count_leaves([_,nil,nil],1).
count_leaves([_,L,R],N):- count_leaves(L,NL),
                          count_leaves(R,NR),
                          N is NL+NR.
```

39 Collect the internal nodes of a tree in a list.

```
internals([[],[],[]],[]).
internals([X,[],[]],[]):- X\=[].
internals([X,L,R],[X|S]):- internals(L,XL), internals(R,XR),
                           append(SL,SR,S).
```

40 Collect the leaves of a tree in a list.

```
leaves([[],[],[]],[]).
leaves([X,[],[]],[X]):- X\=[].
leaves([_,L,R],S):- leaves(L,SL), leaves(R,SR), append(SL,SR,S).
```

41. Collect nodes at a given level of a binary tree.

```
atlevel([[],[],[]],_,[]).
atlevel([X,[],[]],1,[X]).
atlevel([_,L,R],L,S):- D>1, D1 is D-1, atlevel(L,D1,SL),
                       atlevel(R,D1,SR), append(SL,SR,S).
```

42. Construct a complete binary tree.

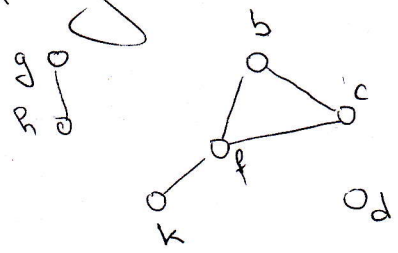
```
cbt(N,T):- cbt(N,T,1).
cbt(N,[[],[],[]],A):- A>N.
cbt(N,[_,L,R],A):- A=<N, AL is 2*A, AR is AL+1,
                   cbt(N,L,AL),
                   cbt(N,R,AR).
```

// Graphs

I Methods to represent graphs in prolog:

1) Edges as facts ..

```
e(h,g)
e(k,f)
e(f,b).
...
```



-isolated nodes cannot be represented.

2) G = [V, E]. where V = [b,c,d, f,g,h,k]
E = [[b,c], [c,f], [f,b], [k,f], [g,h]]

3) [b-c, f-c, g-h, d, f-b, k-f, h-g]

43. Path in a graph // G - list of edges

```
path(_,X,X,_,[X]).
path(G,X,Y,Visited,[X|R]):- X\=Y,
                            member([X,Z],G),
                            not(member(Z,Visited)),
                            path(G,Z,Y,[X|Visited],R)
```

44 Cycle in a graph

```
cycle(G,[X|T]):- member([X,Y],G), path(G,Y,X,[],T).
```