# *3. Reprezentarea şi prelucrarea listelor şi arborilor în SWI-Prolog.*

## 3.1 Liste

O listă este o structură de date ce reprezintă o colecție de elemente de informație (ex. ana, tenis, toma, facultate) aranjate într-o anumită ordine. O asemenea listă poate fi scrisă în Prolog astfel:

*[ana, tenis, toma, facultate]*.

Folosind liste, putem acumula o mulțime de informații într-un singur obiect Prolog. De asemenea câştigăm spațiu şi o mai mare flexibilitate în gestiunea unor date complexe. În general, listele sunt definite recursiv. În funcție de numărul de elemente, o listă poate fi definită în SWI-Prolog, în două feluri:

(1) **lista vidă** (lista cu 0 elemente) este o listă reprezentată în Prolog prin atomul special **[]**.

(2) o **listă nevidă** este o structură cu două componente:

- primul element din listă (*capul listei* - *head*). Capul unei liste poate fi orice termen, constantă, variabilă sau funcție Prolog.
- *corpul sau coada (tail)* listei (format din următoarele elemente din listă). *Coada unei liste trebuie să fie o listă.* Sfârşitul unei liste este, de obicei, reprezentat ca lista vidă.

Pentru exemplul prezentat mai sus:

*[ana, tenis, toma, facultate]*.

capul listei este ***ana***, iar corpul este: ***[tenis, toma, facultate]***.

În general, capul listei poate fi orice obiect Prolog (de exemplu, un arbore sau o variabilă), iar corpul este întotdeauna o listă. Capul şi corpul sunt combinate într-o structură prin intermediul unui functor special. Alegerea functorului depinde de implementarea Prolog, însă pentru SWI-Prolog acesta este caracterul ”**.**”.

**.(Cap, Corp)**

Cum **Corp** este la rândul lui o listă, acesta poate fi lista vidă sau poate avea propriul său *Cap* şi *Corp*.

O listă cu un singur element **ana** se reprezintă în SWI-Prolog astfel:

**.(ana, [])**.

Pentru a reprezenta o listă de orice lungime se procedează astfel:

**.(ana, .(tenis, .(toma, .(facultate, []) ) ) )**.

Structura de listă de mai sus este reprezentată grafic în figura 3.1.

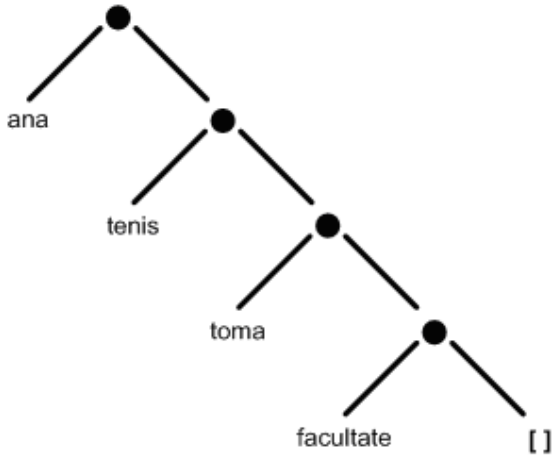


**Figura 3.1** *Reprezentarea arborescentă a listei [ana, tenis, toma, facultate]*

Deoarece structura de date de tip listă este foarte des utilizată în Prolog, limbajul oferă o sintaxă alternativă pentru descrierea listelor, formată din elementele listei separate de virgulă şi încadrate de paranteze drepte. De exemplu, cele două liste prezentate anterior pot fi exprimate astfel:

**[ana].**

**[ana, tenis, toma, facultate].**

Această sintaxă a listelor este generală şi valabilă în orice implementare Prolog. În aceste condiții, în SWI-Prolog sunt posibile următoarele enunțuri:

```
9 ?- Lista1=[a,b,c],
     Lista2 = .(a, .(b, .(c, []))).
Lista1 = [a, b, c],
Lista2 = [a, b, c].

12 ?- Pasiune1 = .(tenis, .(muzica, [])),
      Pasiune2 = [munti, calatorii],
      L = [ana, Pasiune1, toma, Pasiune2].
Pasiune1 = [tenis, muzica],
Pasiune2 = [munti, calatorii],
L = [ana, [tenis, muzica], toma, [munti, calatorii]].
```

În Prolog elementele unei liste pot fi orice: atomi, numere, liste şi în general orice structuri de date.

O operație frecventă asupra listelor este obținerea primului element din listă şi a restului listei, deci a celor două componente ale structurii de listă. Aceasta operație este realizată în Prolog de operatorul de scindare a listelor "|", bară verticală, care separă capul de corpul listei şi este scris astfel:

*L = [a, b, c]. Corp = [b, c]* şi *L = .(a, Corp)*

atunci:

**L = [a | Corp].**

Variabila **a** joacă rolul primului element din listă, iar variabila **Corp** reprezintă lista care conține toate elementele din listă cu excepția primului. Acest operator poate fi aplicat pe orice listă care conține cel puțin un element. Dacă lista conține exact un element, variabila **Corp** va reprezenta lista vidă. De fapt, bara verticală este mult mai generală. Prin intermediul ei se pot obține primele elemente ale listei şi restul listei. Un exemplu este prezentat mai jos.

***[a, b, c] = [a | [b, c] ] = [a, b | [c] ] = [a, b, c | [ ] ] = [a | [b | [c] ] ] = [a | [b | [c | [ ] ] ] ]*.**

### 3.2 Predicate pentru prelucrarea listelor în SWI-Prolog

Cele mai comune operații asupra listelor sunt: verificare dacă un obiect aparține sau nu unei liste, concatenarea a două liste şi obținerea celei de-a treia, adăugarea de noi obiecte în listă, eliminarea (ştergerea) obiectelor din listă, etc.

1. Predicatul **member –** testează apartenența unui obiect la o listă şi se defineşte astfel:

   **member (X, L)**

   unde X este un obiect şi L este o listă. Scopul **member (X, L)** este *true* dacă X apare în L. Predicatul **member** lucrează astfel:

   *X este membru al lui L dacă*

   *(1) X este capul listei L. Acesta este un simplu fapt:*

   ***member (X, [X | Corp]).***

   *(2) X este membru în corpul lui L.* Acesta este o regulă de forma:

   **member(X, [Cap | Corp]):-**

   **member(X, Corp).**

De exemplu,

**member (b, [a,b,c])** este **true.**

**member (b, [a, [b,c]])** este **false.**

**member ([b,c], [a, [b,c]])** este **true.**

**member (X, [a,b,c])** rezultă **X=a, X=b, X=c.**

**2.** Predicatul **concatenare (***notat în SWI-Prolog cu append***)** – concatenează două liste L1 şi L2, rezultatul obținându-se în lista L3, inițial neinstanțiată.

**append (L1, L2, L3)**,

**append ([a,b], [c,d], [a,b,c,d])** este **true.**

**append([a,b], [c,d], [a,b,a,c,d])** este **false.**

În funcție de argumentul L1, putem avea două cazuri:

**(1)** Dacă primul argument este o listă vidă atunci al doilea şi al treilea argument trebuie să fie aceeaşi listă (o vom nota cu L). Acest lucru se exprimă în Prolog astfel:

**append([], L, L).**

**(2)** Dacă primul argument al predicatului *append* este o listă nevidă atunci are un cap şi un corp, iar reprezentarea în Prolog este:

**[X | L1]**

În figura 3.2 prezentăm concatenarea lui **[X | L1]** cu lista L2.

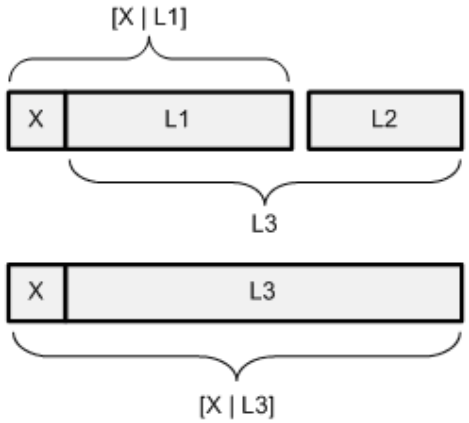


**Figura 3.2** *Concatenare liste*

Rezultatul este lista **[X | L3]** unde L3 este concatenarea lui L1 cu L2. În SWI-Prolog, acest lucru se scrie:

**append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).**

Pentru concatenarea a două liste date se procedează astfel:

```
5 ?- append([a,b,c], [1,2,3], L).
L = [a, b, c, 1, 2, 3].

6 ?- append([a,[b,c],d], [a,[],b], L).
L = [a, [b, c], d, a, [], b].
7 ?- append(L1, L2, [a,b,c]).
L1 = [],
L2 = [a, b, c] ;
L1 = [a],
L2 = [b, c] ;
L1 = [a, b],
L2 = [c] ;
L1 = [a, b, c],
L2 = [] ;
false.
```

Se observă că pentru cazul în care predicatul de concatenare are un singur argument neinstanțiat există o singură soluție (exemplele 5 şi 6), iar pentru cazul în care primele două argumente sunt neinstanțiate (variabile), exemplul 7, se obțin mai multe soluții, corespunzătoare tuturor variantelor de liste care, prin concatenare, generează cea de-a treia listă.

De asemenea, putem utiliza predicatul concatenare pentru a găsi o anumită succesiune în listă. De exemplu, putem căuta atât lunile care preced cât şi lunile care succed o lună dată (de exemplu: *mai*), astfel:

```
10 ?- append(Inainte, [mai | Dupa], [ian, feb, mar, apr, mai,
                        iun, iul, aug, sep, oct, noi, dec]).
Inainte = [ian, feb, mar, apr],
Dupa = [iun, iul, aug, sep, oct, noi, dec] ;
false.
```

Mai departe putem găsi predecesorul şi succesorul lunii mai astfel:

```
8 ?- append(_, [Luna1, mai, Luna2|_], [ian, feb, mar, apr,
                  mai, iun, iul, aug, sep, oct, noi, dec]).
Luna1 = apr,
Luna2 = iun ;
false.
```

La punctul 1 am arătat modalitatea de utilizare a predicatului *member*. Utilizând concatenarea (predicatul *append în SWI-Prolog*) putem defini relația member prin următoarea clauză:

```
member1(X, L) :-
            append(L1, [X | L2], L).
```

Clauza se interpretează astfel: *X* este membru al listei *L* dacă *L* poate fi descompus în două liste astfel încât a doua listă are pe *X* ca antet (head, cap). Este evident faptul că ***member1*** defineşte aceeaşi relație ca ***member***. Am folosit nume distincte (member, member1) pentru a face distincție clară între cele două tipuri de implementări.

În figura 3.3 prezentăm modalitatea de lucru pentru procedura *member1*.

Presupunem următoarea interogare:

```
12 ?- member1(b, [a,b,c]).
true
```

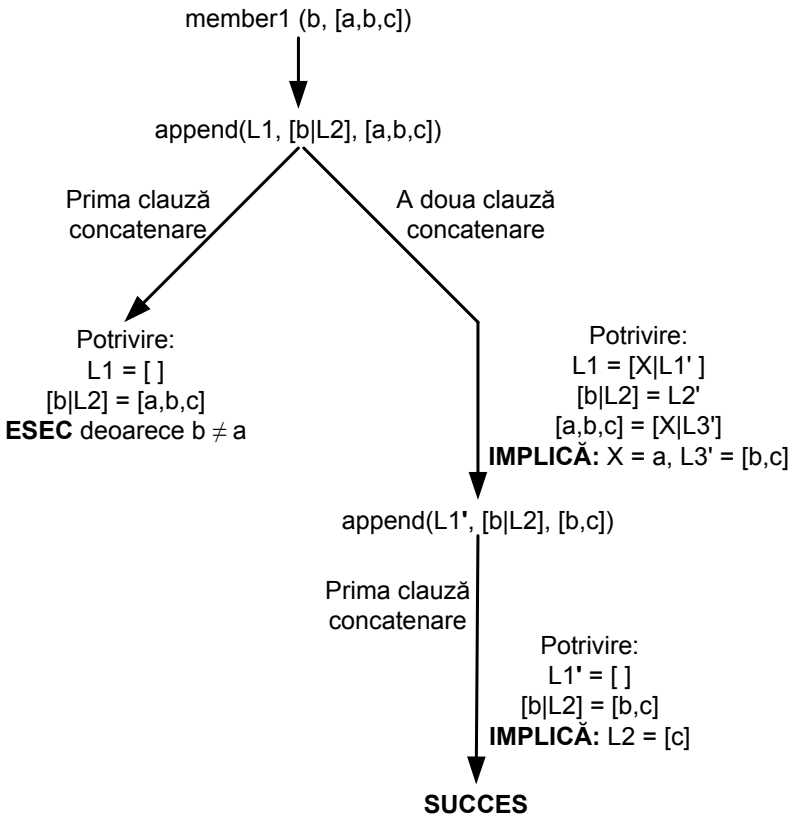


**Figura 3.3** *Procedura de căutare secvențială a unui element într-o listă*

Din figura 3.3 se observă că member1 lucrează identic ca member. Această procedură inspectează lista, element cu element, până când elementul din interogare este găsit sau până la epuizarea listei.

**3.** Adăugarea şi eliminarea elementelor în/din listă (predicatele **select**, **delete**).

Predicatul *select* permite atât inserarea cât şi ştergerea elementelor din listă.

**select(?Elem, ?Lista, ?Rest),**

selectează *Elem* din lista *List* şi returnează elementele rămase în lista *Rest*.

Exemplu de utilizare:

```
3 ?- select(a, [a,b,c],Rest).
Rest = [b, c] ;
false.
```

Conform acestei implementări, *select* nu va elimina decât o apariție a elementului căutat. Astfel, selectarea (eliminarea) lui *a* din lista *[a, b, c, a]* va genera două soluții posibile:

```
4 ?- select(a, [a,b,c,a],Rest).
Rest = [b, c, a] ;
Rest = [a, b, c] ;
false.
```

Predicatul *select* poate fi utilizat şi pentru inserarea de elemente în listă. Spre exemplu, dacă vrem să inserăm constanta *a* (în orice poziție) în lista *[1, 2, 3]* procedăm astfel:

```
8 ?- select(a,L,[1,2,3]).
L = [a, 1, 2, 3] ;
L = [1, a, 2, 3] ;
L = [1, 2, a, 3] ;
L = [1, 2, 3, a] ;
false.
```

În general, operația de inserare a unui element *X* pe orice poziție într-o listă *Lista* se poate realiza prin clauza:

```
insert(X, Lista, ListaExtinsa):-
          select(X, ListaExtinsa, Lista).
```

Predicatul *delete:*

**delete(+Lista1, ?Elem, ?Lista2),**

şterge toate elementele din lista *Lista1* care unifică simultan cu *Elem* şi depune rezultatul în *Lista2*.

Exemplu de utilizare:

```
17 ?- delete([a,b,c,a,d], a, Lista2).
Lista2 = [b, c, d].
```

Din exemplul de mai sus se constată diferența dintre predicatul ***select*** şi predicatul ***delete***. Predicatul *select* elimină doar *prima apariție* a unui element *Elem* din listă, pe când *delete* elimină *toate elementele Elem* din listă.

**4.** Predicatul **sublist** (*incluziunea listelor - nu există în SWI-Prolog*).

Acest predicat returnează *adevărat* (*true*) dacă o listă este sublista alteia. De exemplu,

**sublist([c, d, e], [a, b, c, d, e, f])** este adevărat, iar

**sublist([c, e], [a, b, c, d, e, f])** este fals.

Programul Prolog pentru *sublist* se bazează pe acelaşi principiu ca *member1*, numai că, de această dată, relația este mult mai generală (după cum se observă şi în figura 3.4).

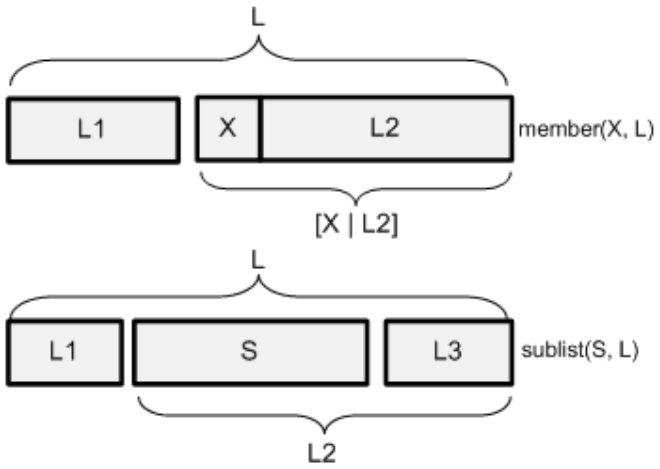


**Figura 3.4** *Relaţiile dintre predicatele member şi sublist*

Relația are două argumente, o listă L şi o listă S, şi verifică dacă lista S este sublistă a lui L. Relația poate fi formulată astfel:

*S este sublistă a lui L dacă*
*(1) L poate fi descompusă în două liste, L1 şi L2,*
*şi*
*(2) L2 poate fi descompusă în două liste, S şi L3.*

Relația de mai sus poate fi exprimată în Prolog astfel:

```
sublist(S,L):-
                append(L1,L2,L),
                append(S,L3,L2).
```

Este evident faptul că procedura *sublist* oferă o mare flexibilitate permițând, pe lângă verificarea dacă o listă este sublista altei liste, găsirea tuturor sublistelor unei liste date.

5. Predicatul **permutation** (*generarea tuturor permutărilor elementelor unei liste date*) – are ca argumente două liste astfel încât una este permutarea celeilalte.

**permutation(?Lista1, ?Lista2,).**

Utilizând procedura *permutation*, se generează toate permutările unei liste (prin *tehnica backtracking*), ca în exemplu de mai jos:

```
6 ?- permutation([a,b,c],P).
P = [a, b, c] ;
P = [b, a, c] ;
P = [b, c, a] ;
P = [a, c, b] ;
P = [c, a, b] ;
P = [c, b, a] ;
false.
```

Relația poate fi formulată astfel:

*(1) Dacă prima listă este vidă atunci şi a doua listă trebuie să fie vidă.*

*şi*

*(2) Dacă prima listă nu este vidă atunci aceasta are forma* [X | L] *şi o permutare a acesteia poate fi construită ca în figura 3.5: mai întâi permută L şi obține L1 şi apoi inserează X pe orice poziție în L1.*

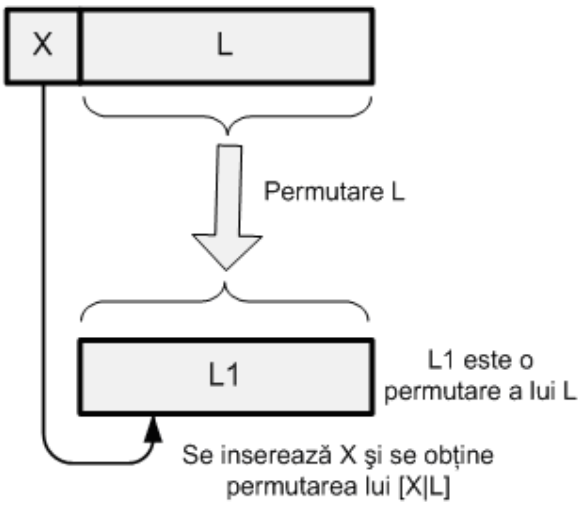


**Figura 3.5** *Modalitate construcție permutări listă [X/L]*

Cele două cazuri (1) şi (2) pot fi exprimate în Prolog prin două clauze astfel:

```
permutare( [], []).
permutare( [X | L], P) :-
        permutare(L, L1),
        insert( X, L1, P).
```

6. Predicatul **sumlist –** calculează suma elementelor unei liste.

   **sumlist(Lista, Suma).**

   Exemplu de utilizare:

```
8 ?- sumlist([10,20.23,3],Suma).
Suma = 33.23.
```

7. Predicatele **max_list** şi **min_list** - returnează maximul/minimul dintr-o listă sau adevărat (*true*) dacă Max/Min este cel mai mare/mic număr din listă.

   **max_list (Lista, Max).**

   **min_list (Lista, Min).**

   Exemple de utilizare:

```
21 ?- max_list([12.46,2,3], Max).
Max = 12.46.

18 ?- max_list([12,2,3], 34).
false.

20 ?- min_list([12.23,2,3], 2).
true.
```

8. Predicatul **numlist** – construieşte o listă ce conține toate numerele dintre argumentele Low şi High, cu condiția ca Low şi High să fie numere întregi şi Low ≤High.

   **numlist(Low, High, Lista).**

   Exemplu de utilizare:

```
25 ?- numlist(5,9,L).
L = [5, 6, 7, 8, 9].
```

9. Predicatul **reverse** – inversează ordinea elementelor din *Lista1* şi unifică rezultatul cu elementele din *Lista2*.

   **reverse(Lista1, Lista2).**

Exemplu de utilizare:

```
28 ?- reverse([i,o,n], L).
L = [n, o, i].
```

**10.** Predicatul **select** – selectează *Elem* din *Lista* lăsând *Rest*.

**select(Elem, Lista, Rest).**

Exemplu de utilizare:

```
29 ?- select(a, [a,b,c], L).
L = [b, c] ;
false.
```

În afara selectării elementelor din listă, predicatul *select* poate fi utilizat şi pentru inserarea de elemente.

```
31 ?- select(d, L, [a,b,c]).
L = [d, a, b, c] ;
L = [a, d, b, c] ;
L = [a, b, d, c] ;
L = [a, b, c, d] ;
false.
```

**11.** Predicatul **last** – întoarce succes (*true*) atunci când *Elem* se unifică cu ultimul element al listei *Lista*. Dacă Elem este o variabilă neinstanțiată atunci aceasta primeşte valoarea ultimului element din listă.

**last (Lista, Elem).**

Exemple de utilizare:

```
33 ?- last([w,q,r,1],1).
true.

34 ?- last([w,q,r,1],M).
M = 1.
```

**12.** Predicatul **nextto** – are succes când *Y* urmează imediat pe *X* în lista *Lista*.

**nextto(X, Y, Lista).**

Exemplu de utilizare:

```
37 ?- nextto(X,mai,[ian,feb,mar,apr,mai]).
X = apr ;
false.
```

**13.** Predicatul **is_list** – are succes când argumentul *Termen* este o listă reprezentată prin [] sau prin functorul ".".

**is_list(Termen).**

Exemplu de utilizare:

```
64 ?- is_list([a,c,n]).
true.
```

**14.** Predicatul **length** – are succes când *Int* reprezintă numărul de elemente din lista *Lista*.

**length(Lista, Int).**

Exemplu de utilizare:

```
40 ?- length([a,b,c,1,4],Int).
Int = 5.
```

**15.** Predicatul **sort** – returnează în lista *Sorted* elementele listei *Lista* sortate în ordine crescătoare. *Duplicatele sunt eliminate.*

**sort(Lista, Sorted).**

Exemplu de utilizare:

```
49 ?- sort([23,12,1,100,12],S).
S = [1, 12, 23, 100].
```

**16.** Predicatul **msort** – returnează în lista *Sorted* elementele listei *Lista* sortate în ordine crescătoare, *fără eliminarea duplicatelor*.

**msort(Lista, Sorted).**

Exemplu de utilizare:

```
50 ?- msort([23,12,1,100,12],S).
S = [1, 12, 12, 23, 100].
```

**17.** Predicatul **merge** – returnează în lista *Lista3* elementele listelor *Lista1 şi Lista2* sortate în ordine crescătoare, *fără eliminarea duplicatelor*. Înainte de aplicarea acestui predicat listele Lista1 şi Lista2 trebuie să fie sortate.

**merge(Lista1, Lista2, Lista3).**

Exemplu de utilizare:

```
54 ?- merge([23,32,100], [23,25,27,123],S).
S = [23, 23, 25, 27, 32, 100, 123].
```

### 3.3 Arbori

Printr-un arbore *A* se înțelege o mulțime finită de obiecte de date numite noduri cu următoarele particularități:

- Există un nod particular numit *rădăcină*,
- Celelalte noduri ale arborelui formează o mulțime finită m ≥1 sau m ≥ 0 de subarbori ai arborelui definit.

Arborele descrie o structură ierarhică, iar forma cea mai generală de descriere a unui arbore este următoarea:

**A = (N, R)**

unde:

N - reprezintă mulțimea nodurilor;

R - reprezintă mulțimea relațiilor dintre noduri sau a arcelor.

O formă generală de arbore este prezentată în figura 3.6.

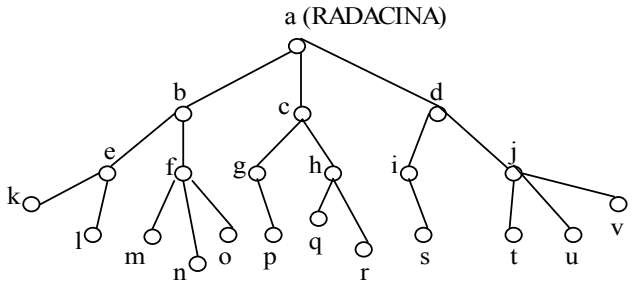


**Figura 3.6** *Exemplu de structură de arbore*

Un *arbore binar* are proprietatea că pentru un nod părinte, fiecare nod aflat în partea stânga a sa are o valoare numerică mai mică decât a părintelui şi fiecare nod aflat în partea dreaptă a nodului părinte are o valoare mai mare decât a sa.

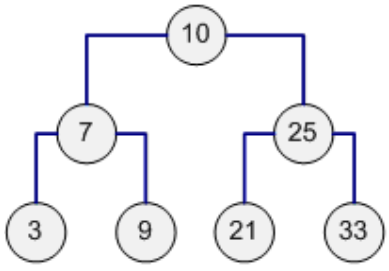


**Figura 3.7** *Exemplu de arbore binar de căutare*

Într-un *arbore binar de căutare* (figura 3.7), pentru orice nod, informația (*inf*) fiului stâng este mai mică decât informația fiului drept, iar informația fiului drept este mai mare sau egală cu informația nodului tată. Este cea mai utilizată categorie de arbori binari pentru că atât căutarea, cât şi inserarea şi ştergerea durează puțin. Timpul de căutare este proporțional cu $\log_2^N$, unde N reprezintă numărul de noduri ale arborelui.

**Parcurgerea (traversarea) arborilor**

Pentru a determina dacă un arbore dat conține un nod cu o informație căutată trebuie efectuată o operație de căutare în structura arborelui. Deci trebuie stabilit un procedeu prin care să poată fi vizitat orice nod al arborelui până la nodul frunză.

Există două procedee de parcurgere (căutare) consacrate:

1. **Căutarea transversală** care constă în vizitarea mai întâi a tulpinii, apoi a tuturor fiilor tulpinii, apoi a fiilor fiilor, ş.a.m.d. până la frunze.
2. **Căutarea în adâncim**e prin care se încearcă să se ajungă, pornind de la rădăcină, cât mai repede posibil până la prima frunză. La acest tip de căutare, în funcție de ordinea de parcurgere (vizitare) a nodurilor există trei moduri distincte de parcurgere:
   - **traversarea (parcurgerea) în pre-ordine -** mai este numită şi parcurgerea RSD(Rădăcină, Stânga, Dreapta).
   - **traversarea (parcurgerea) în post-ordine –** mai este numită şi parcurgerea SDR(Stânga, Dreapta, Rădăcină).
   - **traversarea (parcurgerea) în in-ordine sau traversarea simetrică –** mai este numită şi parcurgerea SRD(Stânga, Rădăcină, Dreapta).

## 3.4 Probleme rezolvate

1. În cele ce urmează prezentăm un program Prolog ce determină numărul elementelor unei liste şi afişează la consolă elementele acesteia.
   - determinare elemente listă (recursiv):
     - avem 0 elemente numai dacă lista este vidă (aceasta este condiția de oprire a recursivității).
     - În cea de-a doua clauză, eliminăm primul element din listă, incrementăm contorul şi numărăm câte elemente are lista rămasă.

```
nrElem([], 0).
nrElem([_|Rest], Nr):-
                    nrElem(Rest, Nr1),
                    Nr is Nr1 + 1.
```

- Pentru afişarea elementelor unei liste vom face apel la recursivitate.
  - Luăm elementele din listă, unul câte unul, şi le afişăm.
  - Oprirea algoritmului are loc atunci când lista va fi goală. Deci, *condiția elementară* (de oprire a recursivității) va fi când lista este vida.

```
afisare([]).
afisare([Prim|Rest]):-
                    write(Prim), write('  '),
                    afisare(Rest).
```

Exemplu de utilizare (*lista* a fost declarată anterior în fişierul sursă Prolog: *lista([roşu, galben, albastru]).*).

```
?- lista(L), nrElem(L,Nr), afisare(L).
```

Răspunsul sistemului este:

```
rosu  galben  albastru
L = [rosu, galben, albastru],
Nr = 3.
```

2. În cele ce urmează prezentăm un exemplu de program Prolog ce afişează la consolă ultimul element din listă.

```
lista([unu, doi, trei, patru, cinci]).

ultim([X|[]], X).
ultim([_|Rest], X):-
                ultim(Rest, X).
```

Condiția de oprire din recursivitate este ca antetul X să fie ultimul element din listă (corpul listei este o listă vidă).

3. Construire şi parcurgere arbore binar de căutare.

Un exemplu de program Prolog ce construieşte un arbore binar de căutare şi afişează elementele arborelui în in-ordine şi pre-ordine.

```
% Author:
% Date: 5/7/2009
arbore1(arb(10, arb(7, arb(3,null,null), arb(9,null,null)),
        arb(25, arb(21,null,null), arb(33,null,null)))).
%parcurgere RSD
preord_RSD(null).
preord_RSD(arb(R,S,D)):- write(R), write(' '),
                         preord_RSD(S),
                         preord_RSD(D).
%parcurgere SRD
inord_SRD(null).
inord_SRD(arb(R,S,D)):- inord_SRD(S),
                        write(R), write(' '),
                        inord_SRD(D).
rsd(A):-preord_RSD(A).
srd(A):-inord_SRD(A).
%inserare element in arbore
ins(Val, null, arb(Val, null, null)).
ins(Val, arb(Rad, L_T, R_T), Rez) :- Val < Rad,
           ins(Val, L_T, Rez1), Rez = arb(Rad, Rez1, R_T).
ins(Val, arb(Rad, L_T, R_T), Rez) :- Val > Rad,
           ins(Val, R_T, Rez1), Rez = arb(Rad, L_T, Rez1).
%cautare element in arbore
cauta(null, _):- write('Numarul nu exista in arbore!').
cauta(arb(X, _, _), X):- write('Numarul intr. exista!')
cauta(arb(Rad, S, _D), X) :- X < Rad, cauta(S, X).
cauta(arb(_Rad, _S, D), X) :- cauta(D, X).
```

*Rezultate obținute la testare:*

```
4 ?- arbore1(A), srd(A), nl, rsd(A), nl, cauta(A,21).
3 7 9 10 21 25 33
10 7 3 9 25 21 33
Numarul intr. exista!
```

4. Sortare liste folosind arbori binari.

În acest exemplu prezentăm o modalitate de utilizarea a arborilor binari pentru sortarea listelor. Algoritmul este următorul: transformăm lista în arbore, apoi transformăm arborele în listă. În acest fel obținem o listă sortată. Evident, aceasta nu este o utilizare normală a arborilor, deoarece dacă vrem să beneficiem de avantajele oferite de arbori ar trebui să memorăm datele direct în ei. Prin acest exemplu vrem să demonstrăm eficiența utilizării arborilor în programarea declarativă. Arborii reprezintă o alternativă la liste pentru situațiile în care datele trebuie căutate rapid sau

memorate într-o anumită ordine. Algoritmul de sortare este prezentat în listingul de mai jos.

```
% sortarea unei liste folosind arbore binar
% lista este convertita in arbore binar si
% apoi este reconvertita
sortare_Lista_cu_ArboreBinar(Lista,ListaNoua):-
                        lista_in_arbore(Lista,Arbore),
                        arbore_in_lista(Arbore, ListaNoua).

% inserare element din lista in arbore
inserare(ElementNou,null, arb(ElementNou, null, null) ) :-
                        !.
inserare(ElementNou, arb(Element, Stanga, Dreapta),arb(
         Element, StangaNou, Dreapta)):-
                        ElementNou < Element,
                        !,
                        inserare(
                        ElementNou, Stanga, StangaNou).
inserare(ElementNou, arb(Element, Stanga, Dreapta), arb(
         Element, Stanga, DreaptaNou)):-
                inserare(ElementNou, Dreapta, DreaptaNou).

% inserarea tuturor elementelor din lista in arbore
inserare_lista([H|T], Arbore, ArboreNou):-
                !,
                inserare(H, Arbore, JumatateArbore),
                inserare_lista(
                T, JumatateArbore, ArboreNou).
inserare_lista([], Arbore, Arbore).

%inserarea tuturor elem. din lista intr-un arbore gol
lista_in_arbore(Lista,Arbore):-
                inserare_lista(Lista, null, Arbore).

%inserare elemente sortate din arbore in lista
arbore_in_lista(Arbore, Lista):-
                arbore_in_lista_aux(Arbore, [], Lista).
arbore_in_lista_aux(null, Lista, Lista).
arbore_in_lista_aux(arb(Elem, Stanga, Dreapta),
                    ListaVeche, ListaNoua):-
    arbore_in_lista_aux(Dreapta, ListaVeche, Lista1),
    arbore_in_lista_aux(Stanga, [Elem|Lista1],ListaNoua).
```

```
%verificare
test:-
    sortare_Lista_cu_ArboreBinar([7,0,6,15,4,10,4,6,3,1]
    ,Sortat), write(Sortat).
```

Rezultate obținute la testare:

```
79 ?- test.
[0, 1, 3, 4, 4, 6, 6, 7, 10, 15]
true
```

5. Problemă de rutare (problema drumurilor).

Un exemplu clasic de program în Prolog pentru planificarea traseelor între oraşe utilizând structuri de date de tip listă, tehnici de tip backtracking şi recursivitate este prezentat în cele ce urmează.

```
% baza de date cu drumuri între oraşe, de forma drum(oras1,
oras2, distanta):
drum('Pitesti', 'Bucuresti', 126).
drum('Pitesti', 'Craiova', 150).
drum('Pitesti', 'Brasov', 149).
drum('Pitesti', 'Rm. Valcea', 68).
drum('Bucuresti', 'Ploiesti', 60).
drum('Rm. Valcea', 'Sibiu', 93).
drum('Sibiu', 'Brasov', 142).
drum('Brasov', 'Sinaia', 37).
drum('Sinaia', 'Ploiesti', 64).

member(X, [Y | T]) :-
            X == Y, ! ; member(X, T).

%Predicatul traseu(X, Y, Traseu, Distanta) este adevărat dacă
%se poate ajunge de la oraşul X la oraşul Y, calculând şi
%traseul Traseu între cele două oraşe.

traseu(X, Y) :-
            traseu(X, Y, [X],0).
traseu(Y, Y, Traseu, Distanta) :-
            reverse(Traseu, Traseu1),
            writeln(Traseu1), writeln(Distanta).
traseu(X, Y, Traseu, Distanta) :-
            (drum(X, Z, Dist) ; drum(Z, X, Dist)),
             not(member(Z, Traseu)),
             traseu(Z, Y, [Z | Traseu],(Distanta + Dist)).
```

```
traseu( _ , _ , _ ,_) :-
              writeln('Nu exista traseu!').
```

Exemple de utilizare:

```
?- traseu('Pitesti','Bucuresti').
```

Răspunsul sistemului este:

```
[Pitesti, Bucuresti]
0+126
```

```
?- traseu('Pitesti','Sinaia').
```

Răspunsul sistemului este:

```
[Pitesti, Bucuresti, Ploiesti, Sinaia]
0+126+60+64
```

```
?- traseu('Pitesti','Constanta').
```

Răspunsul sistemului este:

```
Nu exista traseu!
```

O imagine simplificată pentru această problemă este prezentată în figura 3.8

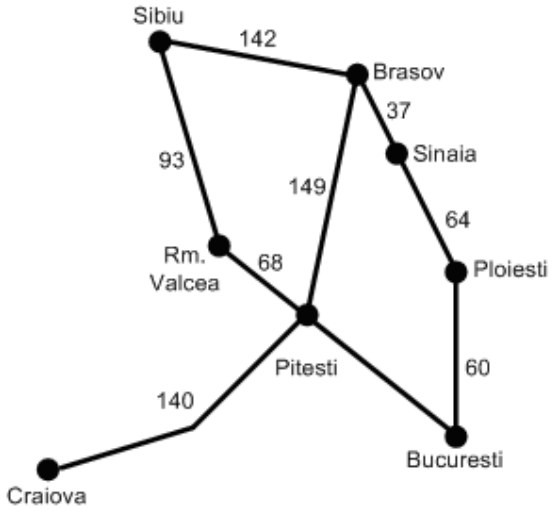


**Figura 3.8** *Schema traseelor considerate în problema de rutare*

În această aplicație am considerat că dacă există drum de la **X** la **Y**, atunci există drum şi de la **Y** la **X.** La realizarea aplicației trebuie avute în vedere unele precauții cu privire la evitarea trecerii de două ori prin același oraş, în cazul parcurgerii unei rute (algoritmul întrând astfel într-o buclă infinită).

### 3.5 Probleme propuse

1. Se vor studia problemele rezolvate (problemele prezentate pe parcursul acestui laborator), încercând găsirea altor posibilități de soluționare a acestora. Utilizați şi alte scopuri (interogări) pentru a testa definițiile predicatelor introduse. Se atrage atenția asupra faptului că toate cunoştințele din acest capitol vor fi necesare şi în derularea celorlalte capitole.
2. Se vor rezolva următoarele probleme propuse şi se va urmări execuția lor corectă.
   - Scrieți listele de mai jos folosind reprezentarea alternativă cu "**.**" (functor) şi [] pentru lista vidă:

     | | |
     |---|---|
     | [a, b] | [a \| [b, c]] |
     | [a \| b] | [a, b \| [ ]] |
     | [a, [b, c], d] | [ [ ] \| [ ] ] |
     | [a, b \| X] | [a \| [b, c \| [ ] ] ] |

   - Să se determine numărul maxim/minim dintr-o listă dată de numere.
   - Să se determine ultimul element dintr-o listă.
   - Scrieți un predicat, *listaPara*, care are două argumente. Primul argument este o listă de numere întregi, iar al doilea argument returnează o listă cu toate numerele pare din prima listă.
   - Să se elimine un element anume dintr-o listă (numai prima apariție a numărului).
   - Să se determine primul număr prim dintr-o listă.
   - Să se elimine primele trei elemente de la începutul unei liste.
   - Să se scrie un predicat Prolog care să calculeze media numerelor unei liste.
   - Să se scrie un program Prolog care calculează şi afişează cel mai mare divizor comun al tuturor numerelor dintr-o listă.
   - Să se scrie un program Prolog care să sorteze descrescător numerele unei liste.
   - Să se scrie un program Prolog care să calculeze şi afişeze suma unei liste de numere întregi.
   - Scrieți un predicat (*postord_SDR*) care să parcurgă în post-ordine arborele prezentat în problema rezolvată 3.
   - Scrieți un predicat (*suma*) care să calculeze suma elementelor din nodurile arborelui.
   - Scrieți un predicat Prolog care să insereze un element într-un arbore binar de căutare.

- Scrieți un predicat Prolog care să caute un număr dat într-un arbore binar.
- Pentru problema rezolvată 4 prezentați o soluție prin care elementele din arbore să fie afişate fără a mai fi salvate mai întâi în listă.
- Pentru problema rezolvată 4 prezentați o soluție prin care la afişare să eliminați duplicatele din listă.