Arbori

1 Obiective

În lucrarea de față vom prezenta operațiile pe arbori binari de căutare. Vom aborda două tipuri de arbori: arbori binari de căutare și arbori ternari. În Prolog, arborii sunt reprezentați prin structuri recursive. Arborele gol îl reprezentăm prin simbolul *nil*.

2 Considerații teoretice

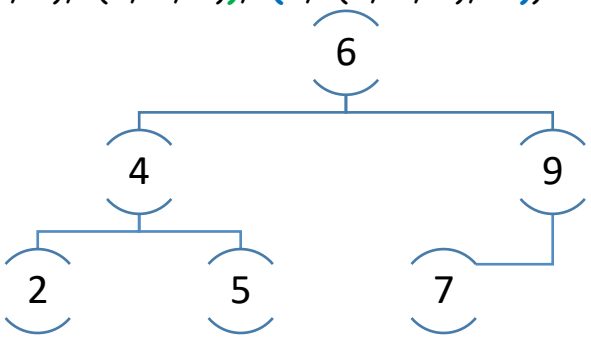
2.1 Reprezentare

Arborele binar de căutare este reprezentat printr-o structură recursivă cu 3 argumente:

- cheia nodului curent,
- subarborele stâng (structură de același tip) și
- subarborele drept (structură de același tip).

Exemplu:

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



Pentru a nu scrie de fiecare dată arborele în interpretorul Prolog, puteți „salva” structura în fișier într-un fapt (axiomă).

Exemple:

```
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)))).
…
?- tree1(T), operatie_pe_arbore(T, …).
```

2.2 Traversarea arborelui

Există 3 posibilități de a traversa arborele în funcție de ordinea în care sunt procesate nodurile: în-ordine, pre-ordine și post-ordine. De exemplu, traversarea în inordine colectează nodurile din subarborele stâng, apoi nodul curent și în final nodurile din subarborele drept.

```
% subarbore stâng, cheie și subarbore drept (ordinea din append)
inorder(t(K,L,R), List):-
        inorder(L,LL),
        inorder(R,LR),
        append(LL, [K|LR], List).
inorder(nil, []).

% cheie, subarbore stâng și subarbore drept (ordinea din append)
preorder(t(K,L,R), List):-
        preorder(L,LL),
        preorder(R, LR),
        append([K|LL], LR, List).
preorder(nil, []).

% subarbore stâng, subarbore drept și apoi cheia (ordinea din append-uri)
postorder(t(K,L,R), List):-
        postorder(L,LL),
        postorder(R, LR),
        append(LL, LR,R1),
        append(R1, [K], List).
postorder(nil, []).
```

Observație1. Cu toate că apelul recursiv al subarborelui drept este făcut înainte de procesarea nodului rădăcină, ordinea corectă este păstrată la construirea listei de ieșire în apelul către predicatul *append*. Astfel, nodurile din subarborele stâng apar primele, urmate de cheia rădăcinii, iar la final nodurile din subarborele drept.
Observație2. Observați că doar poziția nodului curent se schimbă, prin urmare singura diferență între aceste predicate este unde concatenăm nodul curent (*K*).

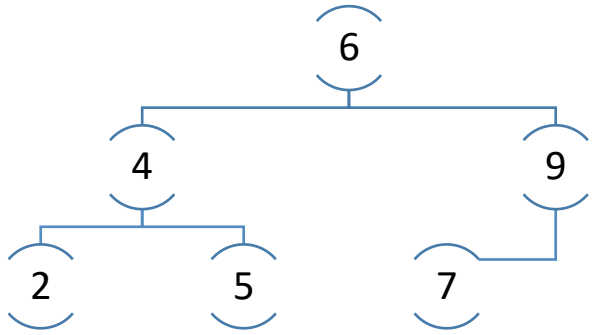
Urmărește execuția la:

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

2.3 Afișare „pretty”

Ne vom folosi de afișarea „pretty” pentru a vizualiza corectitudinea operațiilor pe arbori. Cea mai ușoară variantă de a afișa un arbore este de a-l traversa în inordine și fiecare nod să fie afișat la un număr de spații egal cu adâncimea la care se află nodul în arbore. Rădăcina arborelui se află la adâncimea 0 și fiecare nod este afișat pe un rând diferit.

Exemplu:




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

Dacă studiem afișarea de mai sus putem observa că cheile din stânga sunt afișate primele, urmate de rădăcină, iar apoi de cheile din dreapta. Acest fapt ne sugerează că traversarea în inordine este potrivită pentru un astfel de afișaj. Prin urmare, predicatul care afisează este:

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

% predicatul care printează arborele
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).

% predicat care afișează cheia K la D tab-uri față de marginea din stânga
% și inserează o linie nouă
```

```
print_key(K, D):-D>0, !, D1 is D-1, tab(8), print_key(K, D1).
print_key(K, _):-write(K), nl.
```

Urmărește execuția la:
?- tree2(T), pretty_print(T).

2.4 Căutare unei chei

Predicatul *search_key(Key, T)* verifică dacă există un nod cu cheia *Key* în arborele *T*. Recurența matematică poate fi formulată astfel:

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

Poate fi scris în pseudocod ca:

```
if currentNode = null then return -1; //nu a fost găsit
if searchKey = currentNode.key then return 0; //găsit (clauza 1)
else  if searchKey < currentNode.key (clauza 2)
           //căutam in subarborele stâng
           then search(searchKey, currentNode.left);
      else (clauza 3)
           // căutam in subarborele drept
           search(searchKey, currentNode.right);
```

Acest pseudocod de mai sus este ușor de transformat în specificații Prolog. Deoarece vrem ca predicatul să dea fail în cazul în care cheia nu este găsită, putem să specificăm acest fapt explicit printr-un *fail* explicit când se ajunge la un *nil*, sau implicit, prin nerezolvarea cazului în care se ajunge la nil.

În Prolog se va scrie:

```
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).
```

Urmărește execuția la:
?- tree1(T), search_key(5,T).
?- tree1(T), search_key(8,T).

2.5 Inserarea unei chei

Primul pas constă în căutarea poziției în arbore unde trebuie inserat nodul cu cheia dată (nodul care se inserează este de tip frunză). Înainte a executa inserarea, trebuie să căutam poziția pentru cheia nouă. Dacă mai există un nod cu aceeași cheie atunci nu se realizează inserarea. Când se ajunge la un nil în procesul de căutare, putem crea nodul nou.

```
insert_key(Key, nil, t(Key, nil, nil)). % inserează cheia în arbore
insert_key(Key, t(Key, L, R), t(Key, L, R)):- !. % cheia există deja
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).
```

Urmărește execuția la:

```
?- 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).
```

2.6 Ștergerea unei chei

În cazul ștergerii unei chei trebuie să căutăm nodul care are acea cheie și apoi trebuie să ne asiguram că în urma ștergerii nodului respectiv, arborele va continua să fie arbore binar de căutare.

După găsirea unei chei, putem distinge între trei situații:

- Trebuie să ștergem un nod frunză (**clauza 1 și 2**, când L sau R = nil)
- Trebuie să ștergem un nod cu un copil **(clauza 1 și 2)**
- Trebuie să ștergem un nod cu doi copii **(clauza 3)**

Primele două cazuri sunt destul de simple. Pentru al treilea caz avem două alternative: sau înlocuim nodul care urmează să fie șters cu predecesorul (sau succesorul) – prin reconstruirea legăturilor, sau să mutăm subarborele stâng în partea stângă a subarborele drept (sau vice versa).

Prima alternativă este implementată în predicatul *delete_key/3* de mai jos:

```
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).

% caută nodul predecesor
get_pred(t(Pred, L, nil), Pred, L):- !.
get_pred(t(Key, L, R), Pred, t(Key, L, NR)):- get_pred(R, Pred, NR).
```

Urmărește execuția la:

```
?- 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).
```

2.7 Înălțime unui arbore

Predicatul *height(T,R)* calculează distanța maximă între rădăcina arborelui și o frunză. Recurența matematică poate fi formulată astfel:

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

Sau poate fi formulat în modul următor:

- Înălțimea unui nod nil este 0
- Înălțimea unui nod diferit de nil este maximul dintre înălțimea subarborelui stâng și înălțimea subarborelui drept, plus 1 (max{T.Left, T.Right} + 1)

În Prolog se va scrie:

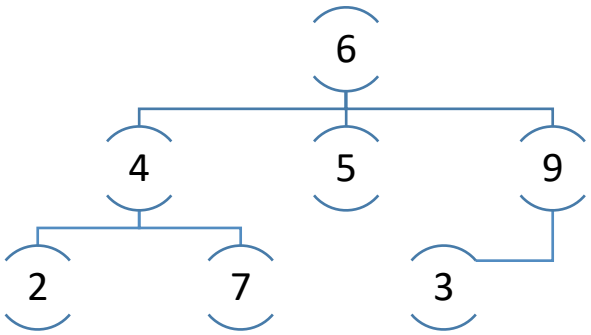
```
height(nil, 0).
height(t(_, L, R), H):-
        height(L, H1),
        height(R, H2),
        max(H1, H2, H3),
        H is H3+1.
```

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

Urmărește execuția la:
?- tree1(T), pretty_print(T), height(T, H).
?- tree1(T), height(T, H), pretty_print(T), insert_key(8, T, T1), height(T1, H1), pretty_print(T1).

2.8 Arbori ternari

Într-un arbore ternar un nod poate să aibă maxim 3 copii. Cu toate că nu este la fel de ușor ca în cazul arborilor binari, relațiile de ordine pot fi stabilite și pentru arborii ternari. Arborele îl putem reprezenta folosind o structură de tipul *t(Key, Left, Middle, Right)*.



Example:

```
ternary_tree(
        t(6,
                t(4,
                        t(2, nil, nil, nil),
                        nil,
                        t(7, nil, nil, nil)),
                t(5, nil, nil, nil),
                t(9,
                        t(3, nil, nil, nil),
                        nil,
                        nil)
        )
).
```

2.8.1 Pretty Print pentru arbori ternari

Deoarece un nod într-un arbore ternar poate avea până la 3 copii, trebuie să folosim o strategie diferită pentru afișarea unei astfel de structuri decât cea folosită pentru arbori binari. O soluție ar fi să printăm fiecare nod la o adâncime de indentare de la marginea stângă (ca înainte); mai mult de atât, subarborele unui nod trebuie să apară sub nodul respectiv (ar trebui afișat după afișarea nodului), dar deasupra nodului următor la același nivel:

```
|6
|       4
|               2
|               7
|       5
|       9
|               3
```

Un astfel de tipar poate fi obținut prin traversarea în preordine a arborelui (Root, Left, Middle, Right), și afișarea fiecărei chei pe o linie, la diferite adâncimi de indentare de la marginea stângă.

3 Exerciții

1. Scrieți predicatele care iterează peste elementele unui arbore ternar
   1.1. *inorder*=Left->Root->Middle->Right
   1.2. *preorder*=Root->Left->Middle->Right
   1.3. *postorder*=Left->Middle->Right->Root

```
?- ternary_tree(T), ternary_preorder(T, L).
L = [6, 4, 2, 7, 5, 9, 3], T= ...


?- ternary_tree(T), ternary_inorder(T, L).
L = [2, 4, 7, 6, 5, 3, 9], T= ...


?- ternary_tree(T), ternary_ postorder (T, L).
L = [2, 7, 4, 5, 3, 9, 6], T= ...
```

2. Scrieți un predicat *pretty_print_ternary/1* care face afișarea unui arbore ternar.

```
?- ternary_tree(T), pretty_print_ternary(T).
6
    4
        2
        7
    5
    9
        3
T= ... ;
```

3. Scrieți un predicat care calculează înălțimea unui arbore ternar.

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

4. Rescrieți predicatul *delete_key* folosind nodul succesor.

```
?- tree1(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)).
```

5. Scrieți un predicat care colectează într-o listă toate cheile din frunzele arborelui binar.

```
?- tree1(T), leaf_list(T, R).
R=[2,5,7], T= ... ;
false.
```

6. Scrieți un predicat care calculează diametrul unui arbore binar.

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

```
?- tree1(T), diam(T, D).
D = 5, T = ... ;
false.
```

7. Scrieți un predicat care colectează într-o listă toate nodurile de la aceeași adâncime din arborele binar.

```
?- tree1(T), same_depth(T, 2, R).
R = [4, 9], T = ... ;
false.
```

8. Verificați dacă un arbore binar este simetric. Un arbore binar este simetric dacă subarborele stâng este imaginea în oglindă a subarborelui drept. Ne interesează structura arborelui nu și valorile din noduri.

```
?- tree1(T), symmetric(T).
false.

?- tree1(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. Scrieți un predicat care colectează într-o listă toate cheile interne (non-frunze) a unui arbore binar de căutare.

```
?- tree1(T), internal_list(T, R).
R = [4, 6, 9], T = … ;
false.
```