Operace na datových strukturách

Aleš Horák

E-mail: **hales@fi.muni.cz**

http://nlp.fi.muni.cz/uui/

Obsah:

- ➜ Práce se seznamy
- ➜ Binární stromy
- ➜ Reprezentace grafů

OPERACE NA DATOVÝCH STRUKTURÁCH

Seznam:

- ➜ rekurzivní datová struktura
- ➜ uspořádaná posloupnost prvků (libovolných termů včetně seznamů)
- ➜ operátor **./2**; prázdný seznam **[]**
- ➜ **.(Hlava,Tělo)**, alternativně **[Hlava|Tělo]**, **Hlava** je (typu) *prvek seznamu*, **Tělo** je (typu) *seznam*

.(a,[])	**[a]**	**[a|[]]**
.(a,.(b,.(c,[])))	**[a,b,c]**	**[a,b|[c]]**, **[a|[b,c]]**, **[a,b,c|[]]**,
		[a|[b,c|[]]], **[a|[b|[c|[]]]]**
...	**[a1,[[b3,c3],d2,e2],f1]**	...

PRÁCE SE SEZNAMY – member

member(+Prvek,+Seznam) – **true**, pokud v seznamu existuje zadaný prvek

1.
```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
?- member(a,[X,b,c]).
     X=a
     Yes
```

2.
```
member(X,[Y|_]) :- X == Y.
member(X,[_|T]) :- member(X,T).
?- member(a,[X,b,c]).          ?- member(a,[a,b,a]),write(ok),nl,fail.
     No                             ok
                                    ok
                                    No
```

3.
```
member(X,[Y|_]) :- X == Y.
member(X,[Y|T]) :- X \== Y, member(X,T).
?- member(a,[a,b,a]),write(ok),nl,fail.
     ok
     No
```


PRÁCE SE SEZNAMY – del A insert

predikát **del(+A,+L,-Vysl)** smaže všechny výskyty prvku **A** ze seznamu **L**

del1(+A,+L,-Vysl) smaže vždy jeden (podle pořadí) výskyt prvku **A** v seznamu **L**

```
del(_,[],[]).                                   ?- del(1,[1,2,1,1,2,3,1,1], L).
del(A,[A|T],V) :- del(A,T,V).                         L = [2, 2, 3]
del(A,[H|T1],[H|T2]) :- A\=H, del(A,T1,T2).           Yes
                                                ?- del1(1,[1,2,1],L).
del1(A,[A|T],T).                                      L = [2, 1] ;
del1(A,[H|T1],[H|T2]) :- del1(A,T1,T2).               L = [1, 2] ;
                                                      No
```

insert(+A,+L,-Vysl) vkládá postupně (při žádosti o další řešení) na všechny pozice seznamu **L** prvek **A**

jednoduchý **insert1(+A,+L,-Vysl)** vloží **A** na začátek seznamu **L** (ve výsledku **Vysl**)

```
insert(A,L,[A|L]).                              ?- insert(4,[2,3,1], L).
insert(A,[H|T1],[H|T2]):- insert(A,T1,T2).            L = [4, 2, 3, 1] ;
                                                      L = [2, 4, 3, 1] ;
                                                      L = [2, 3, 4, 1] ;
insert1(X,List,[X|List]).                             L = [2, 3, 1, 4] ;
                                                      No
```

PRÁCE SE SEZNAMY – PERMUTACE

1. pomocí **insert**

```
perm1 ([],[]).
perm1([H|T],L):- perm1(T,V), insert(H,V,L).
```

```
?- perm1([1,2,3],L).
   L = [1, 2, 3] ;
   L = [2, 1, 3] ;
   L = [2, 3, 1] ;
   L = [1, 3, 2] ;
   L = [3, 1, 2] ;
   L = [3, 2, 1] ;
   No
```

2. pomocí **del1**

```
perm2 ([],[]).
perm2(L,[X|P]) :- del1(X,L,L1),perm2(L1,P).
```

3. pomocí **append**

```
perm3 ([],[]).
perm3(L,[H|T]):- append(A,[H|B],L),append(A,B,L1), perm3(L1,T).
```


PRÁCE SE SEZNAMY – append

append(?Seznam1,?Seznam2,?Seznam) – **Seznam** je spojení seznamů **Seznam1** a **Seznam2**

```
append([],L,L).
append([H|T1],L2,[H|T]) :- append(T1,L2,T).
```

predikát **append** je **vícesměrný**:

```
?- append([a,b],[c,d],L).
   L = [a, b, c, d]
   Yes
?- append(X,[c,d],[a,b,c,d]).
   X = [a, b]
   Yes
?- append(X,Y,[a,b,c]).
   X = []          Y = [a, b, c];
   X = [a]         Y = [b, c];
   X = [a, b]      Y = [c];
   X = [a, b, c]   Y = [];
   No
```

PRÁCE SE SEZNAMY – VYUŽITÍ append

predikát **append** je všestranně použitelný:

```
member(X,Ys)         :- append(As,[X|Xs],Ys).
last (X,Xs)          :- append(As,[X],Xs).
prefix (Xs,Ys)       :- append(Xs,As,Ys).
suffix (Xs,Ys)       :- append(As,Xs,Ys).
sublist (Xs,AsXsBs)  :- append(AsXs,Bs,AsXsBs), append(As,Xs,AsXs).
adjacent(X,Y,Zs)     :- append(As,[X,Y|Ys],Zs).
```


PRÁCE SE SEZNAMY – EFEKTIVITA append

Efektivní řešení predikátu **append** – **rozdílové seznamy** (difference lists)

Rozdílový seznam se zapisuje jako **Seznam1-Seznam2**.

Např.: **[a,b,c]** ... **[a,b,c] - []** nebo **[a,b,c,d] - [d]** nebo **[a,b,c,d,e] - [d,e]**, **obecně [a,b,c|X] - X**
[] ... **A-A**
[a] ... **[a|A]-A**

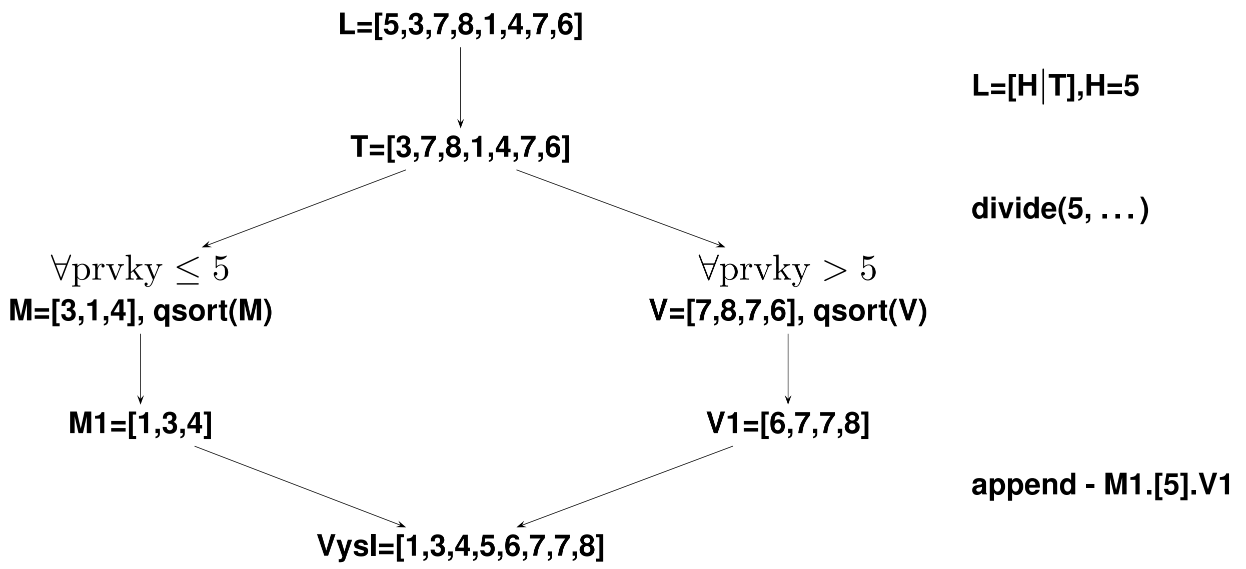
Seznam2 jako volná proměnná slouží jako "ukazatel" na konec seznamu **Seznam1**

predikát **append** s rozdílovými seznamy (**append_dl**):

```
append_dl(A-B,B-C,A-C).

?- append_dl([a,b|X]-X,[c,d|Y]-Y,Z).
X = [c, d|Y]
Y = Y
Z = [a, b, c, d|Y] - Y
Yes
```

TŘÍDĚNÍ SEZNAMŮ — quicksort

predikát **qsort(+L,-Vysl)** – třídí seznam **L** technikou **rozděl a panuj**



TŘÍDĚNÍ SEZNAMŮ — quicksort

predikát **qsort(+L,-Vysl)** – třídí seznam **L** technikou **rozděl a panuj**

```
qsort ([],[])  :- !.       % " řez " - zahoď další možnosti řešení
qsort([H],[H]) :- !.
qsort([H|T],L) :- divide(H,T,M,V),
                  qsort(M,M1), qsort(V,V1),
                  append(M1,[H|V1],L).

divide(_ ,[],[],[])      :- !.
divide(H,[K|T],[K|M],V) :- K=<H, !, divide(H,T,M,V).
divide(H,[K|T],M,[K|V]) :- K>H, divide(H,T,M,V).
```

TŘÍDĚNÍ SEZNAMŮ — quicksort II

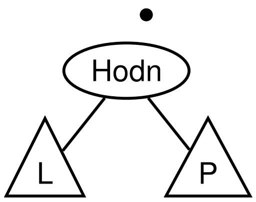
predikát **qsort_dl(+L,-Vysl)** – efektivnější varianta predikátu **qsort** s rozdílovými seznamy

```
qsort(L,S):- qsort_dl(L,S-[]).

qsort_dl ([], A-A).
qsort_dl([H|T],A-B):- divide(H,T,L1,L2),
                      qsort_dl(L2,A1-B),
                      qsort_dl(L1,A-[H|A1]).

divide(_ ,[],[],[])  :- !.
divide(H,[K|T],[K|M],V):- K=<H, !, divide(H,T,M,V).
divide(H,[K|T],M,[K|V]):- K>H, divide(H,T,M,V).
```


USPOŘÁDANÉ BINÁRNÍ STROMY

Reprezentace binárního stromu:

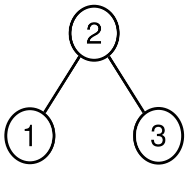
- **nil** – prázdný strom
- **t(L,Hodn,P)** – strom



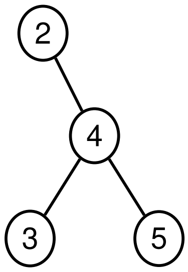
Příklady stromů:

t(nil,8,nil)

t(t(nil,1,nil),2,t(nil,3,nil))



t(nil,2,t(t(nil,3,nil),4,t(nil,5,nil)))



Přidávání do binárního stromu

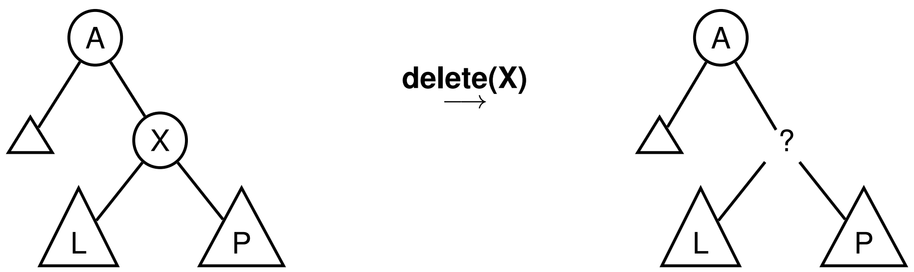
addleaf(+T,+X,-Vysl) přidá do binárního stromu **T** hodnotu **X** na správnou pozici vzhledem k setřídění stromu

```
addleaf(nil,X,t(nil,X,nil)).
addleaf(t(Left,X,Right),X,t(Left,X,Right)).
addleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- Root>X,addleaf(Left,X,Left1).
addleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- Root<X,addleaf(Right,X,Right1).

?- addleaf(nil,6,T),addleaf(T,8,T1), addleaf(T1,2,T2), addleaf(T2,4,T3), addleaf(T3,1,T4).
 ...
 T4 = t(t(t(nil, 1, nil), 2, t(nil, 4, nil)), 6, t(nil, 8, nil))
?- addleaf(t(t(t(nil,1,nil),2,t(t(nil,3,nil),4,t(nil,5,nil))),
      6,t(t(nil,7,nil),8,t(nil,9,nil))),
      10,
      T).
 T = t( t( t(nil, 1, nil), 2, t( t(nil, 3, nil), 4, t(nil, 5, nil))),
      6, t( t(nil, 7, nil), 8, t( nil, 9, t(nil, 10, nil))))
```


Odebírání z binárního stromu

Predikát **addleaf není** vícesměrný ☹ ⇒ nelze definovat:

```
del(T,X,T1) :- addleaf(T1,X,T).
```

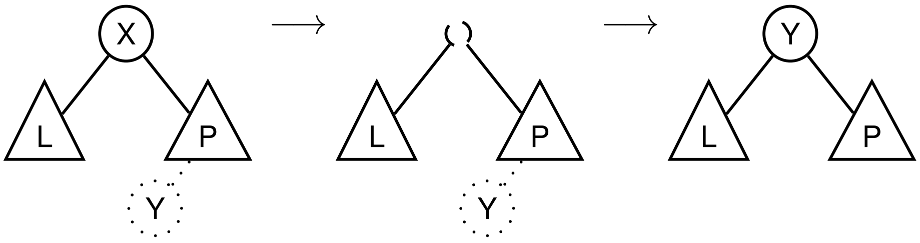


Odebírání z binárního stromu

správný postup:

- ➜ pokud je odebíraná hodnota v **listu** → nahradí se hodnotu **nil**
- ➜ jestliže je ale v **kořenu** (pod)stromu → je nutné tento (pod)strom přestavět

Přestavba binárního stromu při odstraňování kořene **X**:



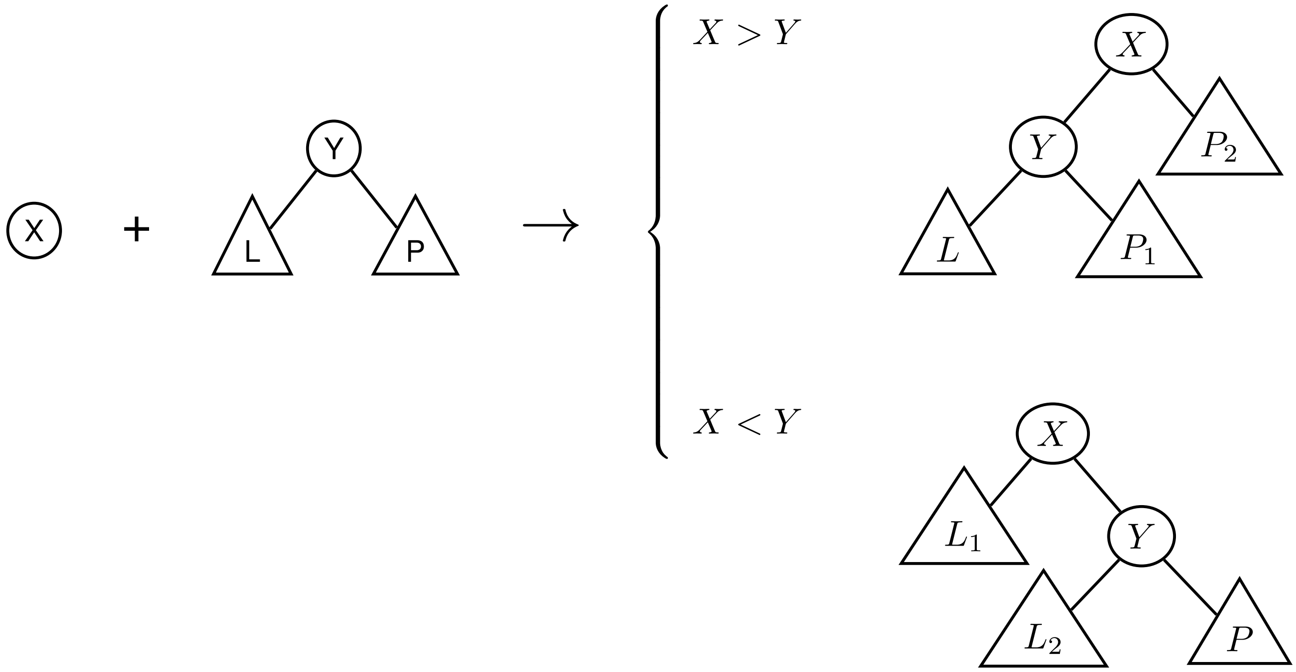
Odebírání z binárního stromu

delleaf(+T,+X,-Vysl) odstraní ze stromu **T** uzel s hodnotou **X**

```
delleaf(t(nil,X,Right),X,Right).
delleaf(t(Left,X,nil),X,Left).
delleaf(t(Left,X,Right),X,t(Left,Y,Right1)) :- delmin(Right,Y,Right1).
delleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- X<Root,delleaf(Left,X,Left1).
delleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- X>Root,delleaf(Right,X,Right1).

delmin(t(nil,Y,R),Y,R).
delmin(t(Left,Root,Right),Y,t(Left1,Root,Right)) :- delmin(Left,Y,Left1).
```

VÍCESMĚRNÝ ALGORITMUS PRO VKLÁDÁNÍ/ODEBÍRÁNÍ

Jiný způsob vkládání:



VÍCESMĚRNÝ ALGORITMUS PRO VKLÁDÁNÍ/ODEBÍRÁNÍ

add(?T,+X,?Vysl) přidá do binárního stromu **T** uzel s hodnotou **X** s přeuspořádáním stromu (jako kořen nebo jinam při navracení)

```
% přidej jako kořen
add(T,X,T1) :- addroot(T,X,T1).
% nebo kamkoliv do stromu (se zachováním uspořádání)
add(t(L,Y,R),X,t(L1,Y,R)) :- gt(Y,X),add(L,X,L1).
add(t(L,Y,R),X,t(L,Y,R1)) :- gt(X,Y),add(R,X,R1).
addroot(nil,X,t( nil ,X, nil )).
addroot(t(L,Y,R),X,t(L1,X,t(L2,Y,R))) :- gt(Y,X),addroot(L,X,t(L1,X,L2)).
addroot(t(L,Y,R),X,t( t(L,Y,R1),X,R2)) :- gt(X,Y),addroot(R,X,t(R1,X,R2)).
addroot(t(L,X,R),X,t(L,X,R)).
```

Definice predikátu **gt(X,Y)** – na konečném uživateli.

Funguje i “obráceně” ⇒ lze definovat:

```
del(T,X,T1) :- add(T1,X,T).
```

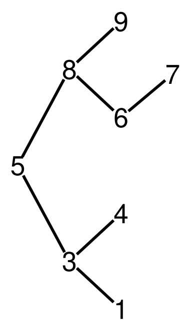
Výpis binárního stromu

pomocí odsazení zobrazujeme úroveň uzlu ve stromu a celkové uspořádání uzlů (strom je tedy zobrazen "naležato")

```
t(
    t(
        t( nil ,1, nil ),
        3,
        t( nil ,4, nil )),
    5,
    t(
        t( nil ,6,
            t( nil ,7, nil )),
        8,
        t( nil ,9, nil )))
```

⟶

```
            9
        8
                7
            6
    5
            4
        3
            1
```



show(+T) vypíše obsah uzlů stromu **T** se správným odsazením

```
show(T) :- show2(T,0).
show2(nil,_).
show2(t(L,X,R),Indent) :- Ind2 is Indent+2,show2(R,Ind2),tab(Indent),
    write(X),nl,show2(L,Ind2).
```

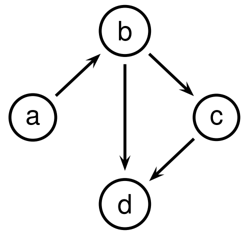

Reprezentace grafů

Příklady způsobů reprezentace grafů (v Prologu):

1. term **graph(V,E)**, kde **V** je seznam vrcholů grafu a **E** je seznam hran grafu.

   Každá hrana je tvaru **e(V1,V2)**, kde **V1** a **V2** jsou vrcholy grafu.

```
G = graph([a,b,c,d ],[ e(a,b),e(b,d),e(b,c),e(c,d )]).
```

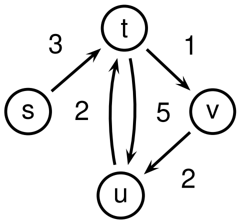


znázorňuje **orientovaný** graf

(2) **vgraph(V,E)** definuje uspořádanou dvojici seznamů vrcholů (**V**) a hran (**E**).

Hrany jsou tvaru **a(PocatecniV, KoncovyV, CenaHrany)**.

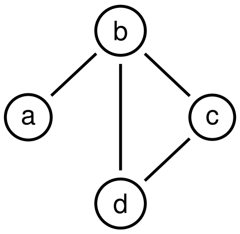
```
G = vgraph([s,t,u,v],[a(s,t,3),a(t,v,1),a(t,u,5),a(u,t,2),a(v,u,2)]).
```



znázorňuje **orientovaný ohodnocený** graf

(3) graf může být uložen v programové databázi jako posloupnost faktů (i pravidel).

```
edge(g3,a,b).
edge(g3,b,c).
edge(g3,b,d).
edge(g3,c,d).
edge(X,A,B) :- edge(X,B,A).
```



díky přidanému pravidlu představuje **neorientovaný** graf (bez pravidla je orientovaný).

CESTY V GRAFECH

Cesta v neorientovaném grafu:

path(+A,+Z,+Graf,-Cesta) v grafu **Graf** najde z vrcholu **A** do vrcholu **Z** cestu **Cesta** (**Graf** je ve tvaru 1).

```
path(A,Z,Graf,Cesta) :- path1(A,[Z],Graf,Cesta).

path1(A,[A|Cesta1],_,[A|Cesta1]).
path1(A,[Y|Cesta1],Graf,Cesta) :- adjacent(X,Y,Graf),not(member(X,Cesta1)),
    path1(A,[X,Y|Cesta1],Graf,Cesta).

adjacent(X,Y,graph(Nodes,Edges)) :- member(e(X,Y),Edges);member(e(Y,X),Edges).
```

Cesty v grafech II

Cesta v ohodnoceném neorientovaném grafu:

path(+A,+Z,+Graf,-Cesta,-Cena) hledá libovolnou cestu z jednoho vrcholu do druhého a její cenu v ohodnoceném neorientovaném grafu.

```
path(A,Z,Graf,Cesta,Cena) :- path1(A,[Z],0,Graf,Cesta,Cena).

path1(A,[A|Cesta1],Cena1,Graf,[A|Cesta1],Cena1).
path1(A,[Y|Cesta1],Cena1,Graf,Cesta,Cena) :- adjacent(X,Y,CenaXY,Graf),
    not(member(X,Cesta1)),Cena2 is Cena1+CenaXY,
    path1(A,[X,Y|Cesta1],Cena2,Graf,Cesta,Cena).

adjacent(X,Y,CenaXY,Graf) :- member(X-Y/CenaXY,Graf);member(Y-X/CenaXY,Graf).
```

Graph je seznam hran ve tvaru **X-Y/CenaXY** (viz **adjacent**).

Kostra grafu

Kostra grafu je strom, který prochází všechny vrcholy grafu a jehož hrany jsou zároveň hranami grafu.

```
stree(Graph,Tree) :- member(Edge,Graph),spread([Edge],Tree,Graph).

spread(Tree1,Tree,Graph) :- addedge(Tree1,Tree2,Graph),spread(Tree2,Tree,Graph).
spread(Tree,Tree,Graph) :- not(addedge(Tree,_,Graph)).

addedge(Tree,[A-B|Tree],Graph) :- adjacent(A,B,Graph),node(A,Tree),
    not(node(B,Tree)).

adjacent(A,B,Graph) :- member(A-B,Graph);member(B-A,Graph).

node(A,Graph) :- adjacent(A,_,Graph).
```

```
?- stree([a-b,b-c,b-d,c-d],T).
   T = [b-d, b-c, a-b]
   Yes
```