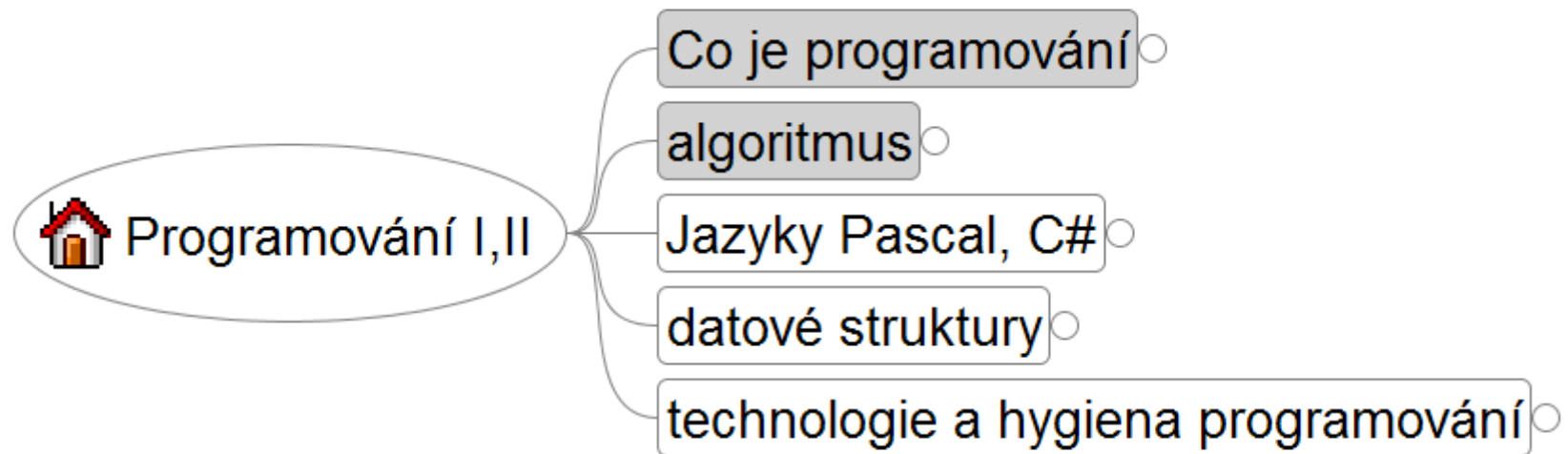


NPRG030 Programování I

RNDr. Tomáš Holan, Ph.D.

4.patro, dveře 404

<http://ksvi.mff.cuni.cz/~holan/>
Tomas.Holan@mff.cuni.cz



Programování je...

- způsob, jak ovládat počítač
- umění řešit úlohy
(a počítač nám v tom může pomoci)
- umění psát programy

? co je to program ?

- předpis, podle kterého počítač může provádět výpočet nějakého algoritmu

? co je to algoritmus ?

Příklad:

Algoritmus na sečtení dvou čísel zapsaných v desítkové soustavě.

1. Desítkové zápisy čísel umístíme pod sebe tak, aby jejich pravé okraje byly zarovnány
2. Delší z čísel doplníme zleva jednou nulou
3. Kratší z čísel doplníme zleva tolika nulami, aby byla obě čísla stejně dlouhá
4. Postupujeme zprava a ke každé dvojici číslic určíme číslici výsledku.

Přitom tato číslice nezáleží jen na této dvojici, ale i na stavu výpočtu

- Stavy jsou dva:
 - „s přenosem“
 - „bez přenosu“
- na začátku je stav „bez přenosu“
- výsledné číslice a stav lze určit například z tabulek:

Pro stav „bez přenosu“:

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Pro stav „s přenosem“:

	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4	5	6
7	8	9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6	7	8
9	0	1	2	3	4	5	6	7	8	9

výsledek „bez přenosu“
výsledek „s přenosem“

!!! !!! !!! !!! !!! !!! !!! !!! !!! !!! !!!

ALGORITMUS NENÍ

„vysvětlit něco, co známe,
někomu, kdo to taky zná“

!!! !!! !!! !!! !!! !!! !!! !!! !!! !!! !!!

Příklad:

Eukleidův algoritmus

Úloha: Najít největšího společného dělitele dvou přirozených čísel
($\text{NSD}(A,B)$).

Postup1: Projít všechna čísla $\leq \min(A,B)$...

Postup2: (Eukleides)

- Dvojice A,B
- Když $A \neq B$, vezmi novou dvojici: **to menší a rozdíl**
- Když $A=B$, je $A=B=\text{NSD}$ původní dvojice

Příklad:

NSD a_1, a_2, \dots, a_n

Postup:

- Pro $k=2, \dots, n$ označ $D_k = \text{NSD } a_1, a_2, \dots, a_k$
- $D_2 = \text{NSD}(a_1, a_2)$
- $D_{i+1} = \text{NSD}(D_i, a_{i+1})$

Příklad:

Úloha: Nalezněte nejkratší cestu šachovým koněm z jednoho pole (třeba D3) na všechna ostatní pole.

Postup:

0. Na startovní pole zapiš číslo 0

1. Všechna **dosud neoznačená** pole dostupná jedním tahem z pole označeného 0 označ 1

2. Všechna **dosud neoznačená** pole dostupná jedním tahem z pole označeného 1 označ 2

3.... 2 3

... atd.

Nalezení cesty pak proved' „odzadu“:

Je-li cílové políčko označené N, hledáme políčko

dostupné z něj jedním tahem a označené N-1 ... až k políčku 0.

			0				

		1		1			
	1				1		
			0				
	1				1		
		1		1			

	2		2		2		
2		2		2		2	
		1	2	1			2
2	1	2		2	1	2	
	2		0		2		2
2	1	2		2	1	2	
		1	2	1			2

	3		3		3		3
3	2	3	2	3	2	3	
2	3	2	3	2	3	2	3
3		1	2	1		3	2
2	1	2	3	2	1	2	3
3	2	3	0	3	2	3	2
2	1	2	3	2	1	2	3
3		1	2	1		3	2

4	3	4	3	4	3	4	3
3	2	3	2	3	2	3	4
2	3	2	3	2	3	2	3
3	4	1	2	1	4	3	2
2	1	2	3	2	1	2	3
3	2	3	0	3	2	3	2
2	1	2	3	2	1	2	3
3	4	1	2	1	4	3	2

„Algoritmus vlny“

Algoritmus

Posloupnost konečného počtu elementárních kroků
vedoucí k vyřešení daného typu úloh
/Encyklopedický slovník/

Charakteristické vlastnosti:

- hromadnost (vstupní data, výstupní data)
- výsledek lze získat zcela mechanicky
- konečnost

[bw_mapa.png](#)

Příklad: (Theseus, Ariadna a Minotaurus)

Úloha:

- určitě najít Minotaura (je-li tam)
- po (ne)nalezení se bez bloudění vrátit

Označme chodby:

- **červená** prošel 2x (značky fixem)
- **žlutá** prošel 1x (natažená nit)
- **zelená** neprošel (ostatní)

Postup:

- Hledání začíná u Ariadny
- V každé místnosti postupuj odpředu podle tabulky:

- | | |
|-----------------------|---|
| 1. je tu Minotaurus | => STOP, našel |
| 2. vede tu > 1 žlutá | => zpět, motej nit, žlutá → červená |
| 3. vede tu > 0 zelená | => vpřed, rozmotávej, zelená -> žlutá |
| 4. je tu Ariadna | => STOP, Minotaurus neexistuje |
| 5. je tu 1 žlutá | => zpět, motej nit, žlutá → červená |

Vlastnosti tohoto algoritmu:

- a) vždy existuje nějaké pravidlo
- b) skončí
- c) cesta zpět se nebude křížit
- d) nit neustále určuje cestu zpět
- e) u Ariadny se zastaví,
jedině když neexistuje cesta k Minotauovi

„Algoritmus prohledávání s návratem“ (backtracking)

Na každém rozcestí, na které přijdeš, zjisti všechny možné cesty a jednu z nich si vyber.

Takto postupuj, dokud to jde nebo dokud se nedostaneš do situace, ve které jsi už byl.

Nemůžeš-li dál, vrať se na poslední rozcestí, kde ses rozhodoval -
- a rozhodni se jinak.

Ověřování správnosti algoritmu

Neexistuje universální metoda zaručující úspěch !

KOREKTNOST

Nebyla opomenuta žádná z možností

ČÁSTEČNÁ SPRÁVNOST

Skončí-li, dá dobrý výsledek.

KONEČNOST

Pro všechna přípustná data skončí.

Dokazování konečnosti

Stačí najít způsob,
jak každý stav výpočtu ohodnotit přirozeným číslem
a ukázat, že provedením jednoho kroku algoritmu
se tato hodnota zmenší.

Invariant

...je tvrzení, které platí po celou dobu výpočtu.

**Kromě netypických výjimek
k ověření správnosti NIKDY NESTAČÍ
provést konečný počet výpočtů !**

=> zkušební výpočty nejsou **pokus o důkaz**, ale **pokus o vyvrácení**.

Úloha: Zjistěte, zda číslo N je prvočíslo

Postup (Čínský algoritmus):

Zjisti, zda N je dělitelem čísla $2^N - 2$.

↪ Je-li, N je prvočíslo.

↪ Není-li, N není prvočíslo.

Příklad:

$N=5$ $2^5 - 2 = 30$ $\Rightarrow 5$ je prvočíslo

$N=9$ $2^9 - 2 = 510$ $\Rightarrow 9$ není prvočíslo

Úloha: Zjistěte, zda číslo N je prvočíslo

Postup (Čínský algoritmus):

Zjisti, zda N je dělitelem čísla $2^N - 2$.

↪ Je-li, N je prvočíslo.

↪ Není-li, N není prvočíslo.

Příklad:

$N=5$ $2^5 - 2 = 30$ $\Rightarrow 5$ je prvočíslo

$N=9$ $2^9 - 2 = 510$ $\Rightarrow 9$ není prvočíslo

Pro $N=341$ selže!

*(Ten algoritmus je správný,
omezíme-li množinu přípustných dat na čísla ≤ 340 .)*

Programování

= Popisování složitějších algoritmických akcí pomocí akcí jednodušších.

Příklad: Výpočet druhé mocniny přirozeného čísla

Záleží na tom, jaké jednodušší akce můžeme použít.

Verse 1: Umocni dané číslo na druhou.

Verse 2: Vynásob dané číslo sebou samým.

Verse 3: Sečti dané číslo tolikrát, kolik je samo.

Verse 4: Označ dané číslo N. Sečti N sčítanců rovných N.

Verse 5: ... nejdříve nový pojem:

PROMĚNNÁ možnost ukládat a vybírat mezivýsledky
Proměnné jsou pojmenovávány pomocí **IDENTIFIKÁTORŮ**,
obvykle složených z písmen a číslic, musí začínat písmenem.

Verse 5:

N, **Počet** a **Suma** jsou proměnné pro celá čísla.

1. přečti do **N** hodnotu ze vstupu.
2. Do **Suma** dosad' 0.
3. Do **Počet** dosad' **N**
4. Dokud je hodnota proměnné **Počet** větší než 0, opakuj akce
 - 4.1 K hodnotě **Suma** přičti hodnotu proměnné **N**
 - 4.2 Hodnotu proměnné **Počet** zmenši o 1
5. Vypiš hodnotu proměnné **Suma**.

DEKLARACE PROMĚNNÝCH

= seznam proměnných a určení jejich **typů**

N, **Pocet**, **Suma**: integer

int **N**; int **Pocet**; int **Suma**;

Krušina, **sedlák** (baryton), **Ludmila**, jeho žena (soprán)

Jazyky pro zápis algoritmů (programovací jazyky).

PŘÍŘAZOVACÍ PŘÍKAZ ukládá hodnotu do proměnné

$x := v$

$x := N-7/2$

$y := y+1$

PŘÍKAZ VSTUPU čte hodnotu ze vstupu a uloží ji do proměnné

$\text{read}(x)$

$\text{read}(N)$

$\text{read}(A, B, C)$

PŘÍKAZ VÝSTUPU zapíše hodnotu výrazu do výstupu

$\text{write}(v)$

$\text{write}(N+1)$

Verse 6:

```
N, Pocet, Suma: integer
1.read( N )           { vstup }
2.Suma := 0           { počáteční hodnoty }
3.Pocet := N
4.Dokud je Pocet > 0, opakuj akce
  4.1.Suma := Suma + N
  4.2.Pocet := Pocet - 1
5.write( Suma )       { tisk výsledku }
```

KOMENTÁŘE

↳ zvyšují srozumitelnost

↳ nemají vliv na význam / běh

V jazyku Pascal: { cokoliv } (* cokoliv *)

Lze je napsat kamkoliv, kde smí být mezera

Řízení běhu programu

= v jakém pořadí se budou jednotlivé kroky/akce/příkazy provádět

Možnosti:

↪ **příkaz skoku**

mění pořadí provádění příkazů

↪ **strukturované příkazy**

vytvářejí složitější příkazy z jednodušších

PŘÍKAZ SKOKU určuje příští prováděnou instrukci
Ve většině jazyků má tvar

GOTO číslo příkazu

```
N, Pocet, Suma: integer
1: read( N )           { vstup }
2: Suma := 0           { pocatecni hodnoty }
3: Pocet := N
4: je-li Pocet = 0, pak GOTO 8
5: Suma := Suma + N
6: Pocet := Pocet - 1
7: GOTO 4
8: write( Suma )       { tisk vysledku }
```

Poznámka: řádka <-> příkaz

Poznámka: Dva druhy skoků:

↳ podmíněný

↳ nepodmíněný

↳ S těmito příkazy už bychom vystačili
↳ Když vynecháme deklarace, máme BASIC

STRUKTUROVANÉ PROGRAMOVÁNÍ

Edsger W. Dijkstra: Goto statement considered harmful, 1968

<= potřeba zvládat velké programy,
potřeba dělby práce

čím silnější prostředky máme k dispozici,
s tím větší kázní a obezřetností je musíme používat

nerespektování přirozených struktur
nebo dokonce jejich (násilná) likvidace
se v budoucnu projeví vážnými a nepředvídatelnými
negativními důsledky.

STRUKTUROVANÉ PROGRAMOVÁNÍ

vytváření složitějších (strukturovaných) příkazů
skládáním z jednodušších

System několika málo konstrukcí,
kterými lze vyjádřit všechny algoritmické konstrukce.
Přitom další rozšiřování už nic nepřidá.

TVRZENÍ

Strukturovaným příkazům rozumíme lépe
než (stejně složitým) příkazům nestrukturovaným.

PŘÍKLAD

↳ Ber, dokud dávám!

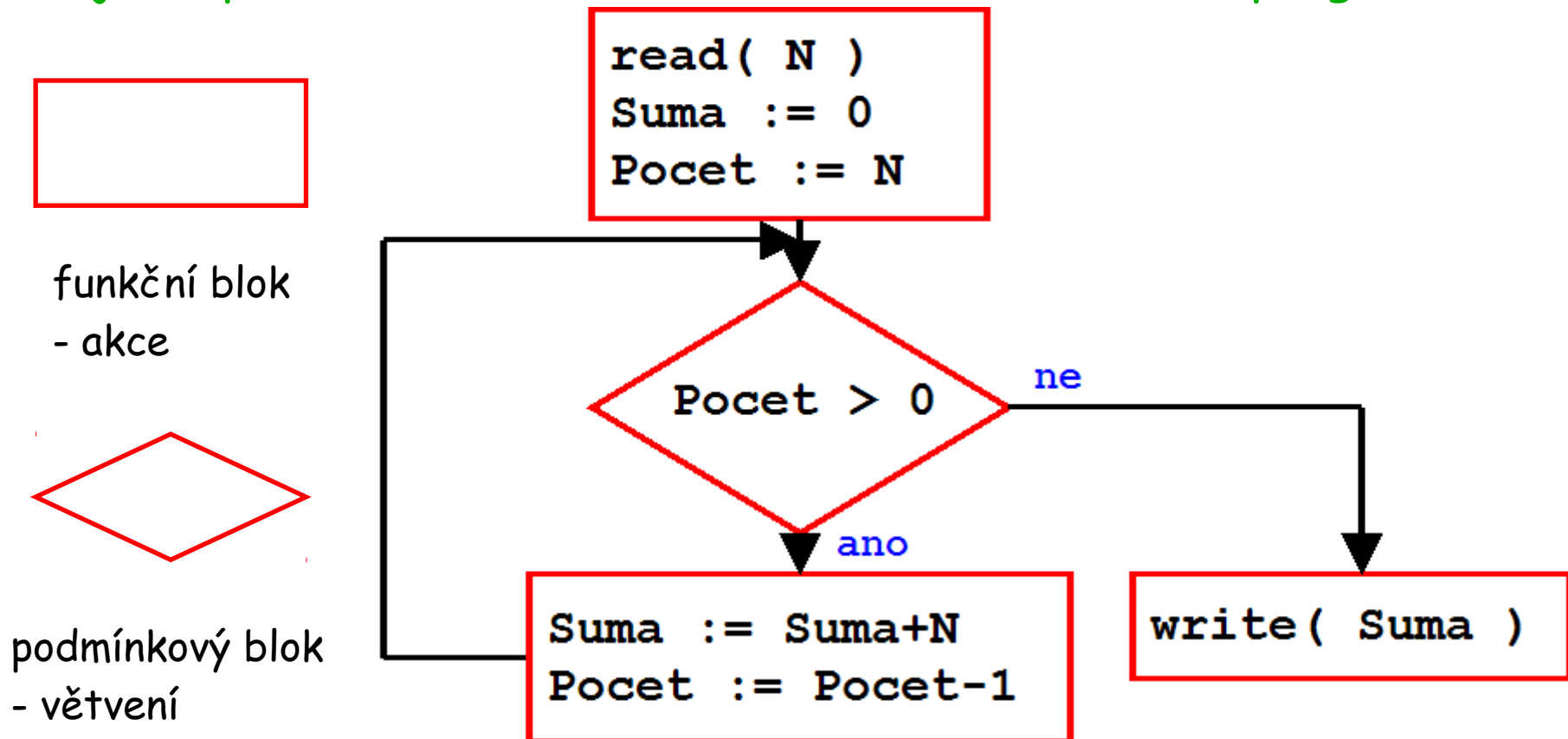
↳ Jestli prší, počkej, až pojede vlak.

Odbočka:

VÝVOJOVÉ DIAGRAMY

Historicky: Jeden z pokusů, jak překonat složitost programů.

Použijeme pro znázornění konstrukcí strukturovaného programování.

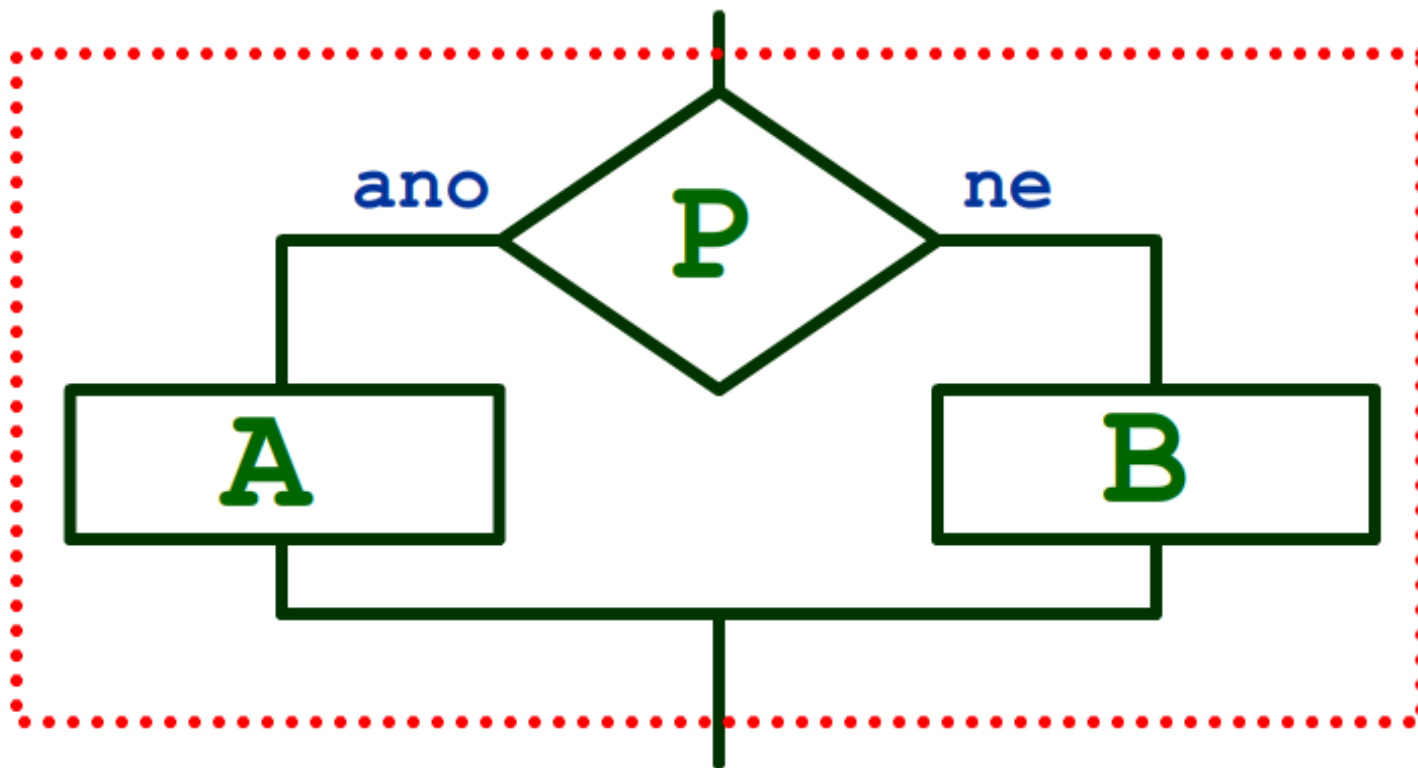


STRUKTUROVANÉ PROGRAMOVÁNÍ

Podmíněný příkaz

úplný:

if **P** **then** **A** **else** **B**

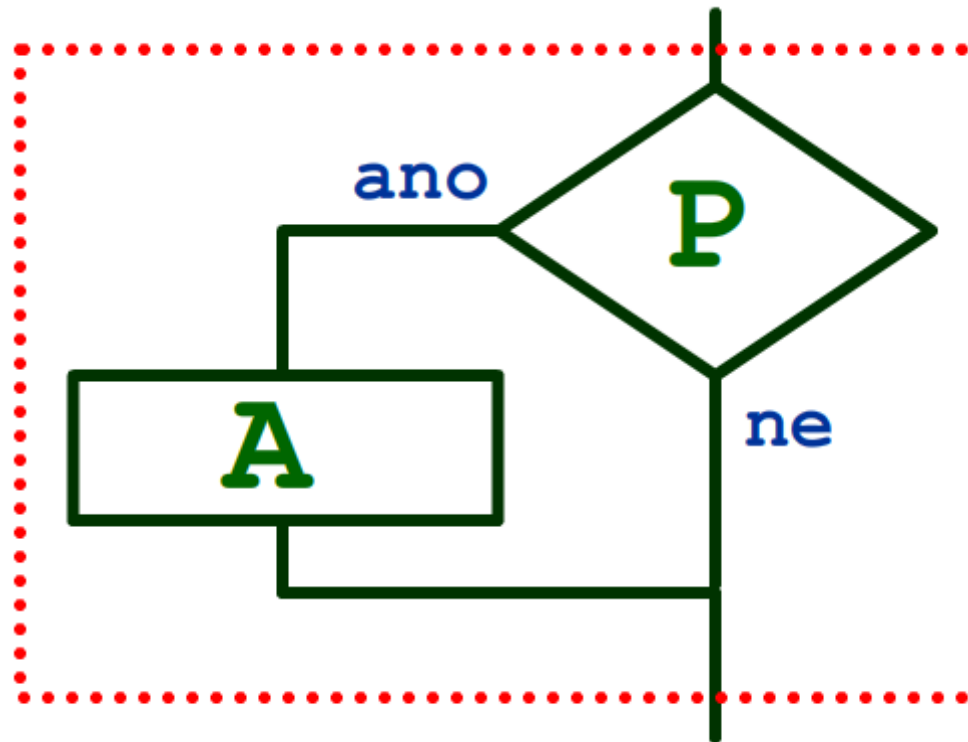


STRUKTUROVANÉ PROGRAMOVÁNÍ

Podmíněný příkaz

neúplný:

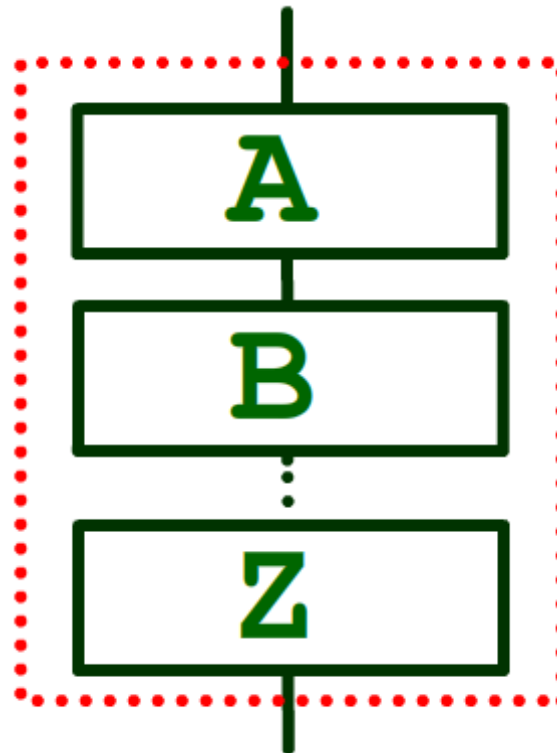
if **P** then **A**



STRUKTUROVANÉ PROGRAMOVÁNÍ

Složený příkaz

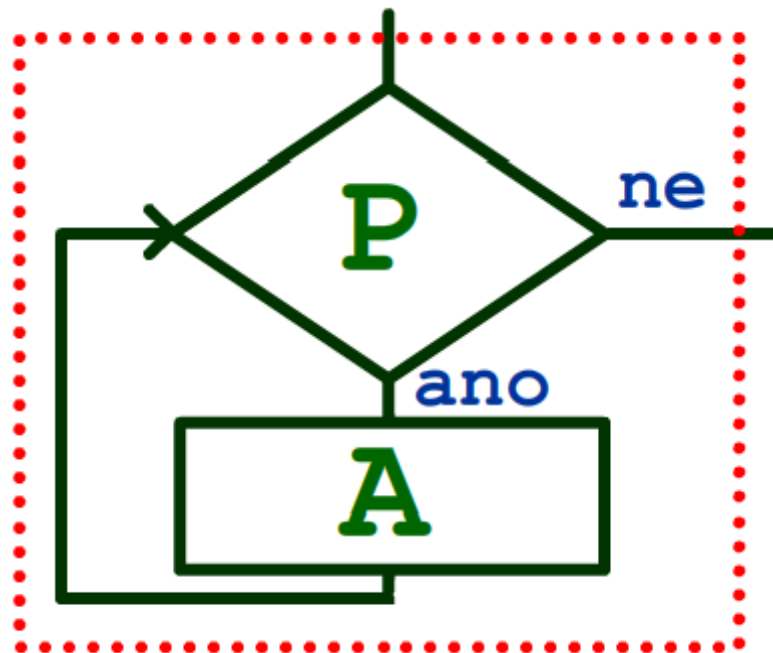
begin **A**;**B**; ... **Z** **end**



STRUKTUROVANÉ PROGRAMOVÁNÍ

Příkazy cyklu

while **P** **do** **A**

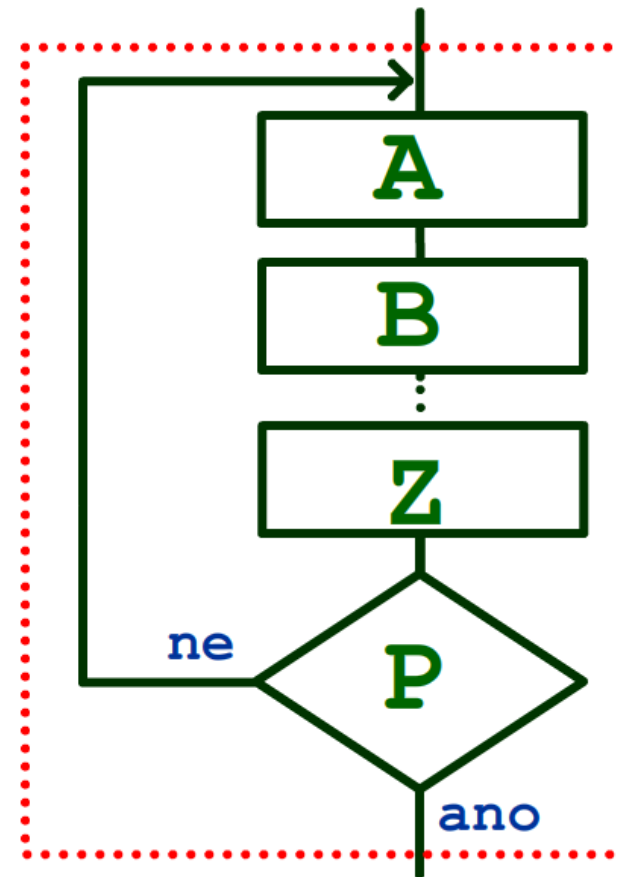


kontrola **PŘED** provedením těla cyklu

STRUKTUROVANÉ PROGRAMOVÁNÍ

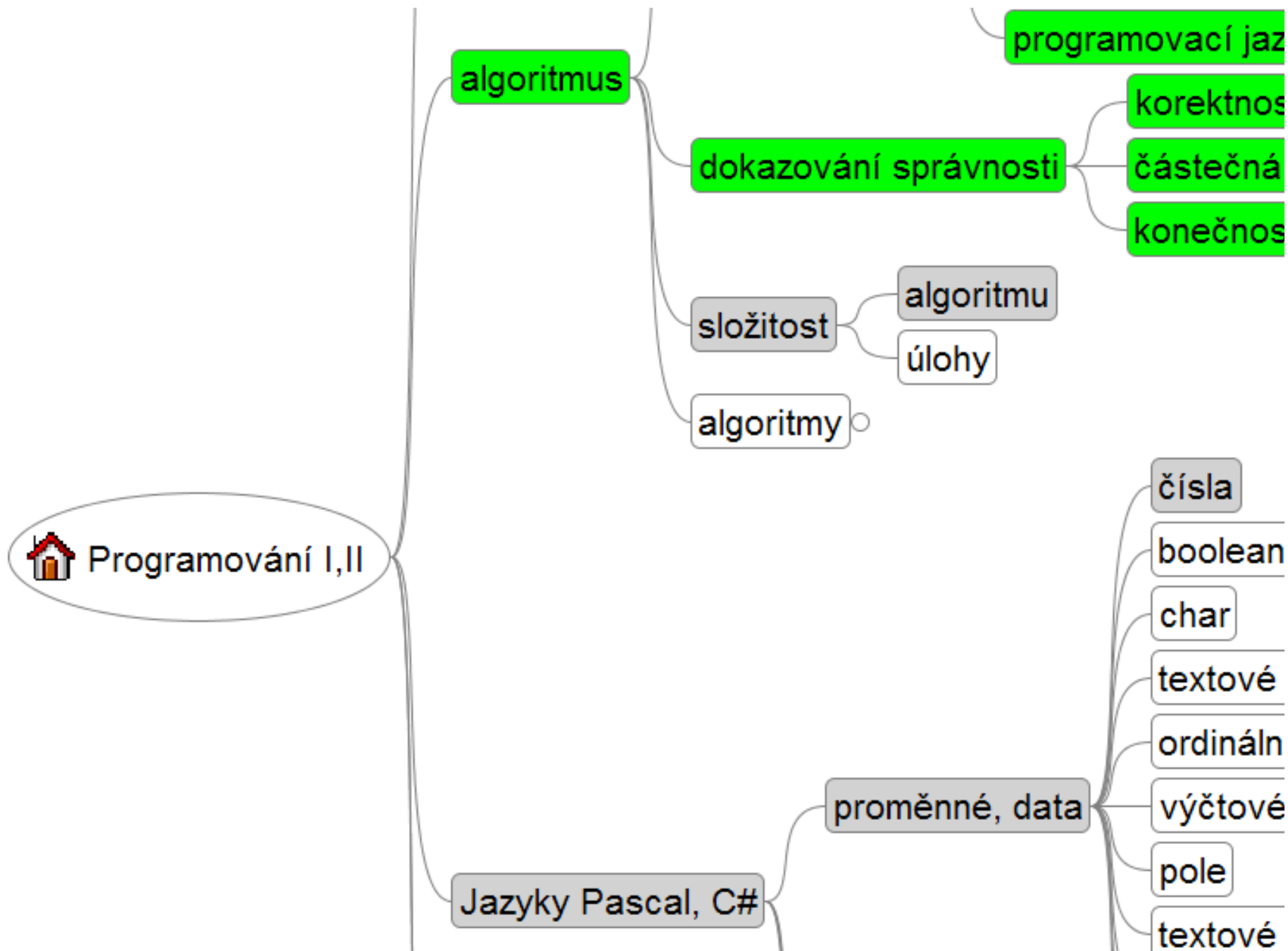
Příkazy cyklu

repeat **A;B;...Z** until **P**



kontrola PO provedení těla cyklu





Čísla v algoritmech a programech

- 10^{26} Poloměr vesmíru
- 2651 studujících studentů MFF UK
- 3.142857... Ludolfovo číslo
- 10^{16} stáří vesmíru v sekundách !!! budeme potřebovat !!!
- 3 délka bakalářského studia na MFF UK v letech
- 36524 délka života stoletého člověka (ve dnech) (!)
- ...

Čísla v algoritmech a programech

=> pracujeme se dvěma typy čísel:

CELÁ

malý omezený rozsah
přesné hodnoty

REÁLNÁ

velký rozsah
přibližné hodnoty
neumíme representovat všechny hodnoty

typ Integer

HODNOTY

Celá čísla zobrazitelná v počítači

ROZSAH se může lišit v jednotlivých implementacích,
proto konstanta **MAXINT**, rozsah je

-MAXINT . . +MAXINT

, typicky 2^{15} nebo 2^{31}

typ Integer – operace

OPERÁTORY TYPU NÁSOBENÍ

- * násobení
- div dělení
- mod zbytek po dělení
- / dělení s výsledkem typu real

OPERÁTORY TYPU SČÍTÁNÍ

- + sčítání
- odčítání

RELAČNÍ OPERÁTORY

=, <>, <, >, <=, >=

Celočíselné dělení a zbytek

$i \text{ div } j$ a $i \text{ mod } j$

je jasné, jaké hodnoty budou výsledkem, pro $i \geq 0$, $j > 0$

Mimo tento rozsah - raději vyzkoušet!

- ještě raději se vyhnout!

Priorita operátorů

1.operace typu násobení

2.operace typu sčítání

3.relační operátory

V případě stejné priority – odleva.

typ Integer – funkce

ABS (x) absolutní hodnota

SQR (x) druhá mocnina

ODD (x) x je liché

POZOR! POZOR! POZOR! POZOR!

**Správná hodnota aritmetického
výrazu je zaručena pouze tehdy,
když jsou všechny argumenty
i mezivýsledky hodnotami typu
(integer)!**

Kdyby MAXINT byl 1000:

600+500-400

(běhové kontroly)

Turbo/Borland/Free Pascal

integer 2B

shortint 1B

longint 4B

byte 1B

word 2B

typ **Real**

HODNOTY

Racionální čísla,
jen konečná množina,
výsledek je jen **APROXIMACE** správného výsledku.

VÝSLEDNÉ CHYBY záleží

na representaci čísel
na řešení úloze
na zvoleném algoritmu.

Odhady velikosti chyb se zabývá **NUMERICKÁ MATEMATIKA**.

typ Real - zápis konstant

HODNOTY

desetinná tečka a nepovinně číslice za ní
nepovinně exponent ve tvaru E<integer>

SEMILOGARITMICKÝ TVAR

0.3768

-326.21

1234.

0.03E-4

-10.583E60

POZOR! POZOR! POZOR! POZOR!

V paměti je jiná reprezentace
než desítkový zápis

=>

při vstupu a výstupu dochází
k zaokrouhlování!

POZOR! POZOR! POZOR! POZOR!

Co udělá program?

Poučení

Testovat reálná čísla
(vypočtená nebo načtená)
na rovnost nemusí mít dobrý smysl !

Místo toho:

```
if abs(koruny1-koruny2) < Epsilon  
    then
```

typ Real - operace

- * + - výsledek je typu Real,
je-li alespoň jeden z operandů
typu Real
- / výsledek je vždy typu Real

Takže $4 / 2$ je typu Real.

typ Real – funkce

ABS(x)

SQR(x)

SIN(x)

ARCTAN(x)

EXP(x)

TRUNC(x)

SQRT(x)

COS(x)

LN(x)

ROUND(x) rozsah

Složitost algoritmů

potřeba ČASU, PAMĚTI...

Jak ji měřit?

- . NE absolutně
- . NE pro konečně mnoho úloh

Raději:

„Co se stane, když se velikost úlohy
zdvojnásobí?“

Složitost algoritmů

Formálně:

N velikost úlohy

$f(N)$ spotřeba (času nebo paměti)

- . v nejhorším případě
- . v průměrném případě

Složitost algoritmů

Definice:

Funkce g je $O(f)$

když

$$\exists n_0, C$$
$$\forall n > n_0: g(n) \leq C \cdot f(n)$$

Složitost algoritmů - Příklad

```
var x,i,N,Sum: integer;  
begin  
    read( N );  
    Sum := 0;  
    i := 1;  
    while i <= N do  
        begin  
            read( x );  
            Sum := Sum + x;    i := i+1;  
        end;  
        writeln( Sum / N )  
    end
```

časová složitost $O(N)$

Složitost algoritmů - Příklady

složitost	1 sekunda	1 hodina	1000 hodin
n	1	3.600	3.600.000
$n \log n$	1	568	278.000
n^2	1	60	1.897
n^3	1	15	153
2^n	1	12	22

POZOR! POZOR! POZOR! POZOR!

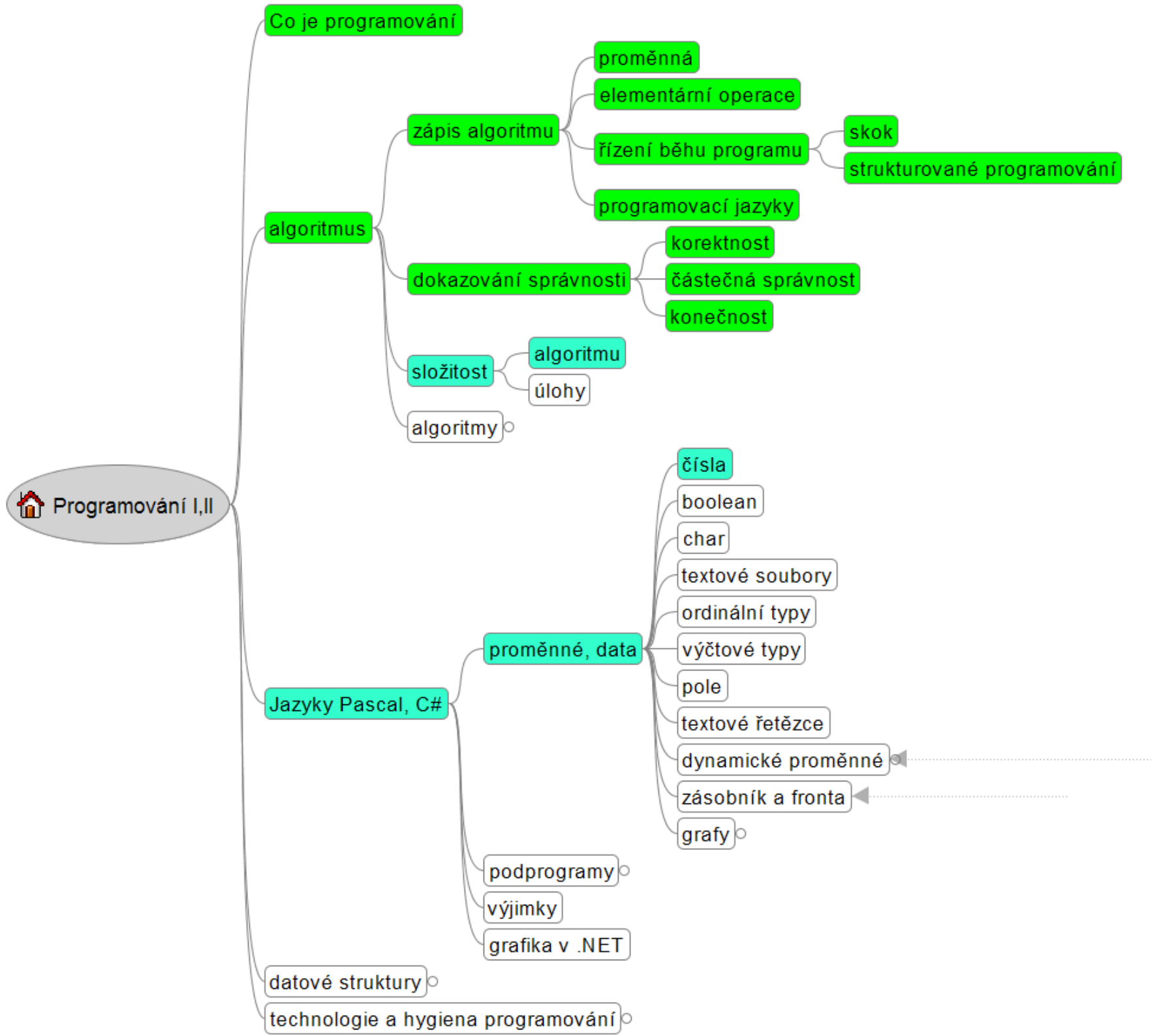
**Složitost neříká (skoro) nic
o konkrétním případě!**

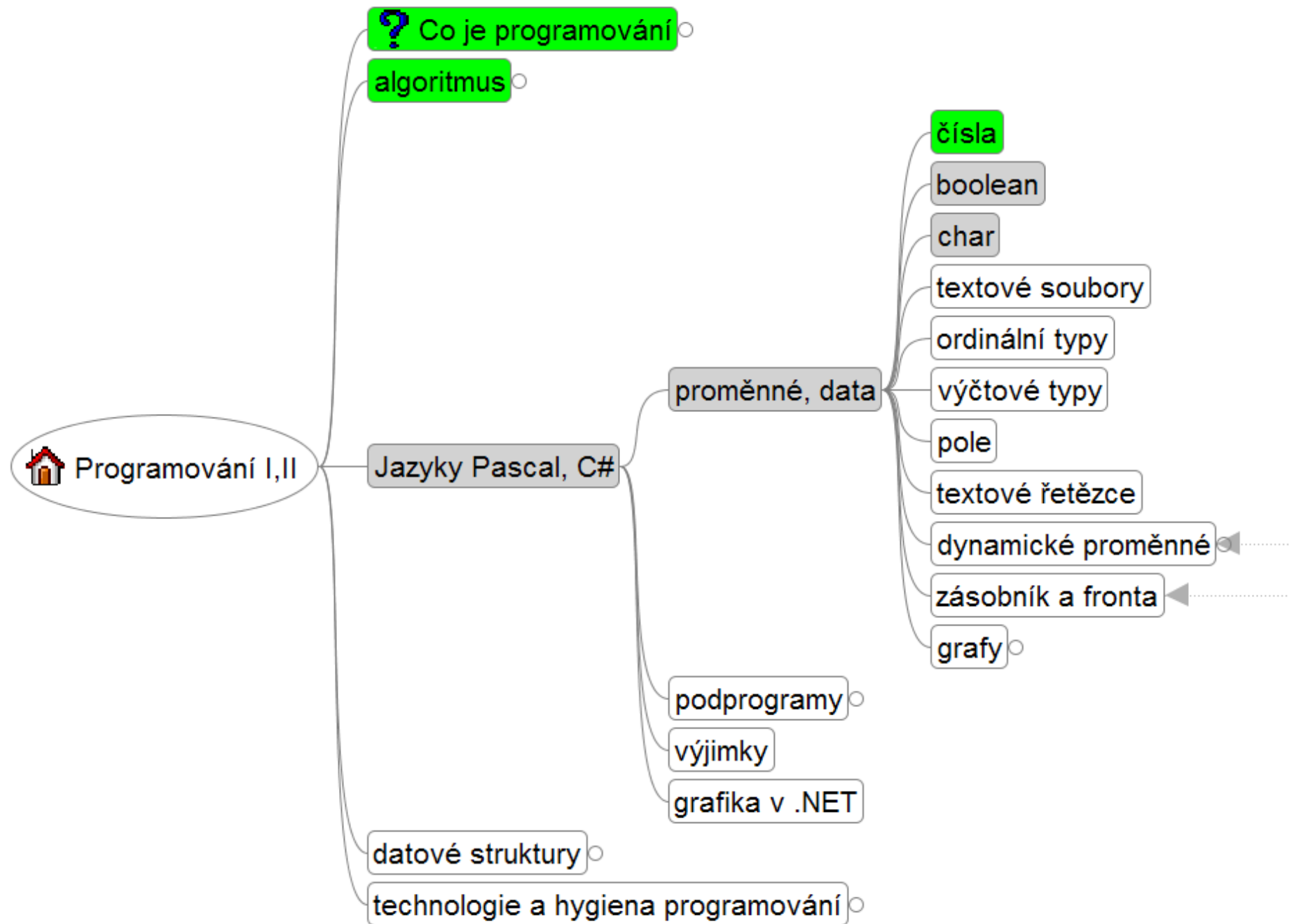
Příklad:

algoritmus	složitost
A1	$1000\ n$
A2	$100\ n\ \log\ n$
A3	$10\ n^2$
A4	n^3
A5	2^n

? Který algoritmus je nejrychlejší?

N	nejrychlejší algoritmus
2..9	A5
10..35	A3
36..22026	A2
22027..	A1





Podmínka

= něco, co JE, nebo NENÍ splněno

typ Boolean

hodnoty: **TRUE** pravda

FALSE lež

domluva (optimistická):

FALSE < TRUE

když X , Y jsou (porovnatelné) výrazy, potom

$X = Y$

$X \neq Y$

$X < Y$

$X > Y$

$X \leq Y$

$X \geq Y$

jsou výrazy typu boolean.

$2 < 3$

$\text{ABS}(x-3) < \text{ABS}(Y-3)$

typ Boolean - operace

NOT negace

AND konjunkce

OR disjunkce

not P P neplatí

P and Q platí P i Q současně

P or Q platí P nebo Q (nebo oboje)

? jak zapsat VYLUČUJÍCÍ NEBO ?

? a implikaci ?

typ Boolean - priority operací

1) negace

2) konjunkce

3) disjunkce

POZOR! POZOR! POZOR! POZOR!

Logické operátory mají vyšší prioritu
než relační operátory.

Takže $x < y \text{ and } y < z$

se vyhodnocuje jako

$x < (y \text{ and } y) < z$

= CHYBA!

Zpět k definici IF, WHILE, REPEAT:

„podmínka“

znamená

„výraz typu boolean“

příklad MONOTONIE...

Zkrácené vyhodnocování

výrazů typu `boolean`

P or Q pokud `P=TRUE`,
výsledek bude `TRUE`
=> nezávisí na Q

R and S pokud `R=FALSE`,
výsledek bude `FALSE`
=> nezávisí na S

Pro určení hodnoty výrazu (v takovém případě)
už nemusíme vyhodnocovat ostatní jeho členy.

ÚPLNÉ VYHODNOCOVÁNÍ

vyhodnotí všechny podvýrazy

ZKRÁCENÉ VYHODNOCOVÁNÍ

skončí vyhodnocování výrazu nebo i podvýrazu
ve chvíli, kdy dokáže určit výsledek

příklad hledání čísla s logaritmem...

příklad test prvočíselnosti...

```
... while PRVOCISLO and (D<=MEZ) do
```

práce se znaky: typ CHAR = písmeno

konstanty typu CHAR: '@', ' ' ' ' ' '

příklad

```
var c: char;  
begin  
    c := '?';  
    repeat  
        read( c )  
    until c='*';  
    read( c );  
    write( c )  
end.
```

kód znaku

každému znaku, který patří do znakové sady, je přiřazeno určité číslo, tzv. **Ordinální číslo znaku**

standardní funkce:

ORD (c) char -> integer
CHR (x) integer -> char

příklad

ord('A') = 65
chr(33) = '!'
chr(-5) = **CHYBA!**

Kódy znaků závisí na počítači a překladači, norma požaduje:

- '0', '1', ..., '9'

v tomto pořadí a bezprostředně za sebou

- 'A', 'B', ..., 'Z'

v tomto pořadí

- jsou-li k dispozici i malá písmena, musí pro ně platit stejná podmínka jako pro velká písmena

Otázka:

jak zapsat podmínku

„znak Z je číslice“

?

Odpověď 1:

`(Z='0') or (Z='1') or...or (Z='9')`

Odpověď 2:

$(z \geq '0')$ and $(z \leq '9')$

Příklad:

Vstup celých čísel znak po znaku

$C_0C_1C_2C_3 \dots C_n$

(zápis čísla bez znaménka)

$$10^n * C_0 + 10^{n-1} * C_1 + \dots + 10^0 * C_n$$

$$= (\dots (c_0 * 10 + c_1) \\ * 10 + c_2) \\ * 10 + c_3) \\ * \dots) \\ * 10 + c_n$$

HORNEROVO SCHEMA

Hodnota číslice

$$n = \text{ord}(n) - \text{ord}('0')$$

příklad čtení čísla, se znaménkem a mezerami

Textový soubor

posloupnost znaků rozdělená do řádek

READ (x) načtení x

READLN (x) načtení x
a potom přechod na další řádku

oba příkazy čtou ze STANDARDNÍHO
VSTUPNÍHO TEXTOVÉHO SOUBORU

WRITE (x) výstup x

WRITELN (x) výstup x

a potom přechod na další řádku

oba příkazy píší na **STANDARDNÍ VÝSTUPNÍ
TEXTOVÝ SOUBOR**

Lze použít i pro jiné textové soubory.

Představa o textovém souboru

- . na konci každého řádku je zvláštní znak
ODDĚLOVAČ ŘÁDEK
- . na konci každého textového souboru je zvláštní znak
UKONČOVACÍ ZNAK SOUBORU
- . tyto zvláštní znaky NEPATŘÍ od množiny hodnot
typu CHAR
- . jejich smysl je dávat souboru STRUKTURU

Základní akce = vstup/výstup jednoho znaku.

`Read(c)` naplní **c** a posune se v souboru o jeden znak dále. **Zpátky NELZE.**

Po přečtení posledního znaku nelze číst dál, pokus o čtení způsobí chybu.

Standardní funkce

- `. eof` **end of file** {jsem na konci souboru}
- `. eoln` **end of line** {jsem na konci řádky}

```
read( x1, ..., xn )
```

```
begin
```

```
    read( x1 );
```

```
    ...
```

```
    read( xn )
```

```
end;
```

```
readln( x1, ..., xn )
```

```
begin
```

```
    read( x1 );
```

```
    ...
```

```
    read( xn );
```

```
    readln
```

```
end;
```

Skutečná implementace (nejen v BP):

- . řádky textových souborů jsou odděleny dvojicí znaků CR a LF (`chr(13)` a `chr(10)`)
(!nebo jenom CR nebo jenom LF!)
- . textový soubor ukončen znakem EOF (`chr(26)`)
(pokud je ukončen, nemusí)
- . čtení čísla končí až na bílém znaku za číslem
=> vstup

123AB

způsobí chybu.

Výstup do standardního výstupního souboru

vystupovat mohou hodnoty typu

`char`, `integer`, `real`, `boolean`

...a znakové řetězce

Příkaz `writeln` zapíše jen oddělovač řádek.

```
write( x1,...,xn )
```

```
begin
```

```
    write( x1 );
```

```
    ...
```

```
    write( xn )
```

```
end;
```

```
writeln( x1,...,xn )
```

```
begin
```

```
    write( x1 );
```

```
    ...
```

```
    write( xn );
```

```
    writeln
```

```
end;
```

Formátování výstupu

výraz : délka

Význam:

Hodnota výraz má být zapsána na výstup tolika znaky, kolik je hodnota délka (to může být také výraz!).

Ne vždy to lze splnit.

Pokud to splnit lze, doplňuje se mezerami zleva.

Boolean se tiskne jako **TRUE** nebo **FALSE**.

Real:

- v semilogaritmickém tvaru
- jedna číslice před desetinou tečkou
- nejméně jedna číslice za desetinou tečkou
- **E** nebo **e**
- znaménko exponentu
- exponent

výraz : délka : míst

Textové soubory v TP

1. deklarace

```
var f: text;
```

2. přiřazení

```
assign( f, 'c:\vstup.txt' );
```

3. otevření

```
reset( f ) NEBO rewrite( f )
```

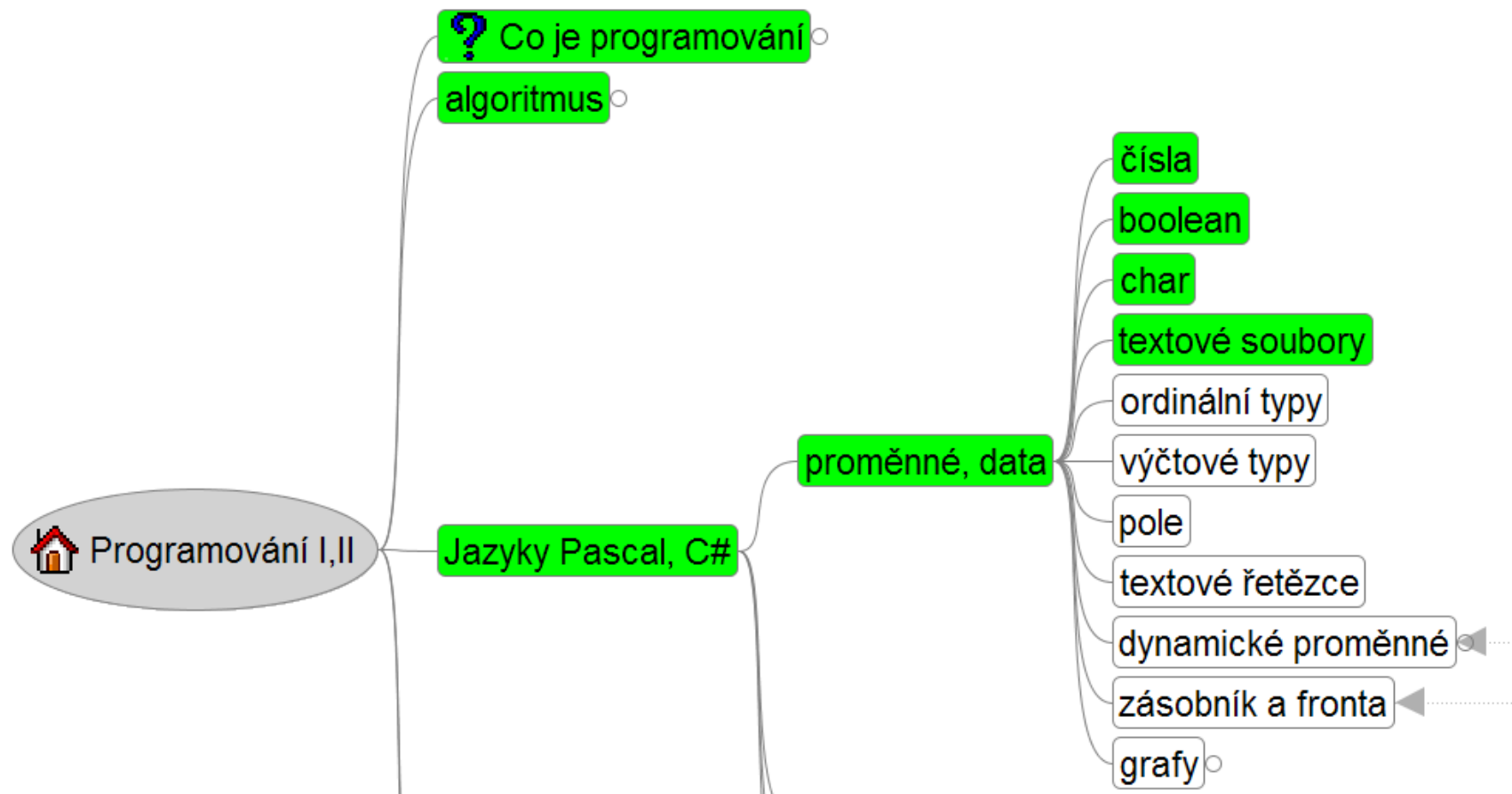
4. vstup

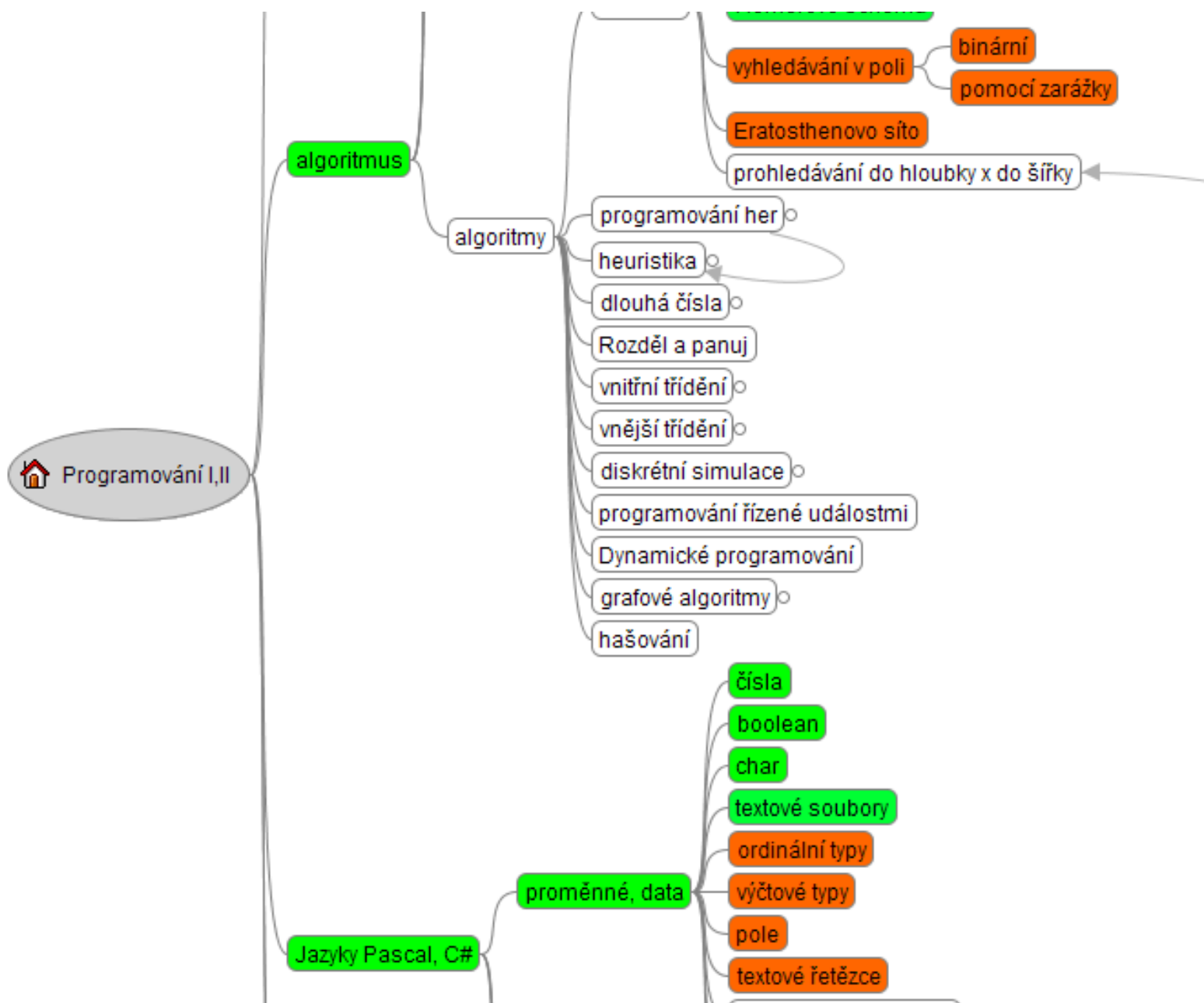
výstup

```
read( f, ...)      write( f, ...)  
readln( f, ...)    writeln( f, ...)
```

zavření

```
close( f )
```





Ordinální typy

standardní: integer, char, boolean

Vlastnosti ordinálních typů:

1. hodnot je konečný počet
a hodnoty jsou uspořádány
2. ke každé hodnotě (kromě té největší) existuje
bezprostřední následník
a ke každé hodnotě (kromě té nejmenší) existuje
bezprostřední předchůdce

Funkce

ORD **ordinální hodnota**

boolean

`ord(FALSE)=0` `ord(TRUE)=1`

integer

`ord(N)=N`

char

`ord(c)= podle tabulky znaků`

PRED předchůdce

```
pred( -123 )=-124  
pred( 0 )=-1  
pred( TRUE )=FALSE  
pred( '3' )='2'
```

SUCC následník

```
succ( -123 )=-122  
succ( 9 )=10  
succ( FALSE )=TRUE //POZOR BP, FPC...  
succ( '3' )='4'
```

Příklad

```
var zn: char;  
begin  
    zn := 'A';  
    repeat  
        writeln( 'Znak ', zn, ' má číslo ', ord(zn) );  
        zn := succ( zn )  
    until zn > 'Z'  
end.
```



**Jak jednoduše zjistit,
jsou-li v naší znakové sadě písmena
bezprostředně za sebou**



Výčtové typy

Definice typu:

`identifikátor = popis typu`

Popis výčtového typu:

`(identifikátor {,identifikátor})`

Příklad

```
den_tydne = (pondeli, utory,  streda,  
             ctvrtek,  patek,  sobota,  nedele) ;  
  
var den1, den2: den_tydne;  
...  
den1  := ctvrtek;  
den2  := pondeli;  
den1  := den2;  
den1  := succ( den1 );
```

? pondeli vs. 'pondeli' ?

Typ interval

popis typu interval

`<konstanta>..<konstanta>`

např.

```
male_cislo = 0..255;  
cislice    = '0'..'9';  
pracovni_dny = pondeli..patek;  
cisla_od_m_do_n = m..n;
```

Typ interval

obě konstanty:

- . stejný typ
- . ordinální typ
- . první \leq druhá

Interval je podmnožinou hostitelského typu

=> všechny jeho hodnoty jsou zároveň hodnotami hostitelského typu

definice typů jsou uvozeny klíčovým slovem **TYPE**

Příkaz cyklu FOR

```
for <identifikátor> := <výraz>  
  to/downto <výraz> do <příkaz>
```

= pro cykly, kde **předem** známe počet opakování

Např.

```
for i:=1 to N do  
  soucet := soucet + i;  
for zn := 'a' to 'z' do  
  writeln( 'znak ',zn,  
           'má číslo ',ord(zn) );
```

```
for i:=1 to N do ...
```

```
  i := 1;
```

```
  while i<=N do
```

```
  begin
```

```
    ...
```

```
    i := succ(i)
```

```
  end;
```

```
for i:=N downto 1 do ...
```

```
  i := N;
```

```
  while i>=1 do
```

```
  begin
```

```
    ...
```

```
    i := pred(i)
```

```
  end;
```

`continue`

ukončení těla cyklu

`break`

vyskočení z cyklu

!! Nedosazujte do řídicí proměnné cyklu !!

Strukturované datové typy

umíme strukturovat příkazy,
můžeme strukturovat i data

POLE

slovo

1	2	3	4	5	6	7	8	9	10	11	12
P	R	O	G	R	A	M	O	V	Á	N	Í

popis typu:

`array[typ{ , typ}] of typ`

`slovo: array[1..12] of char;`

Pole

indexovaná proměnná

`proměnná [výraz { , výraz }]`

`slovo[3] = 'O'`

`slovo[9] = 'V'`

příklad: obracet slova

příklad: frekvence znaků

Eratosthenovo síto

(Έρατοσθένης,

276/272 – 194 př. n. l. v Alexandrii)

Vícerozměrné pole

= podle definice

```
slovník = array[1..VelikostSlovníku] of  
          array[jazyky] of  
            array[1..MaxDelka] of char
```

lze zapsat i jako

```
slovník = array[1..VelikostSlovníku,  
                jazyky,  
                1..MaxDelka] of char
```

podobně

```
sl[CisloHesla][jazyk][pismeno]
```

lze zapsat i jako

```
sl[CisloHesla, jazyk, pismeno]
```

? Co znamená zápis

```
sl[CisloHesla[jazyk]][pismeno] ?
```

? Kolik prvků typu integer má pole

A: `array[-2..2, 3..8, boolean]` of integer ?

Vyhledávání v poli

Úloha:

V neuspořádaném poli najít prvek
s jednou konkrétní hodnotou

```
1:   for i:=zac to kon do  
      if P[i]=HLEDANY then...
```

```
2:   i:=zac;  
      while (i<=kon) and (P[i]<>HLEDANY] do  
          i := i+1;
```

**! Pokud vyhodnocuje booleovské výrazy nezkráceně,
dojde k chybě ! (RangeCheck!)**

3: trik: „vyhledávání se záložkou“

pole o 1 prvek delší a tam umístíme hledanou hodnotu

```
P[kon+1] := HLEDANY;  
i := zac;  
while P[i]<>HLEDANY do  
    i := i+1;  
if i = kon+1 then... { nebyl tam }
```

Strukturované konstanty v BP

const

<jmeno>: <typ> = <hodnota>;

Příklad:

const

```
A: array[1..10] of integer
  = (2,3,5,7,11, 13,17,19,23,29);
AA: array[boolean, 1..5] of char
  = ( ('f','a','l','s','e'),
      (' ','t','r','u','e') );
```

**Ve skutečnosti to jsou jen proměnné
s počáteční hodnotou!**

Representace znakových řetězců (nejen v pascalu)

pole

- a) ukládat délku
- b) ukončovací znak

V TP:

typ string

var

```
a: string;  
b, c: string[10];  
d: string[50];
```

`string[MaxDelka]`

`string[N]` je representováno jako
`array[0..N] of char,`

0-tý prvek obsahuje délku řetězce
(jako char)

`length(s)`

`copy(s, odkud, kolik)`

`pos(co, kde),` vrací 0, pokud neobsahuje

Příklad: nalezení všech výskytů podřetězce

Příklad: náhrada podřetězce

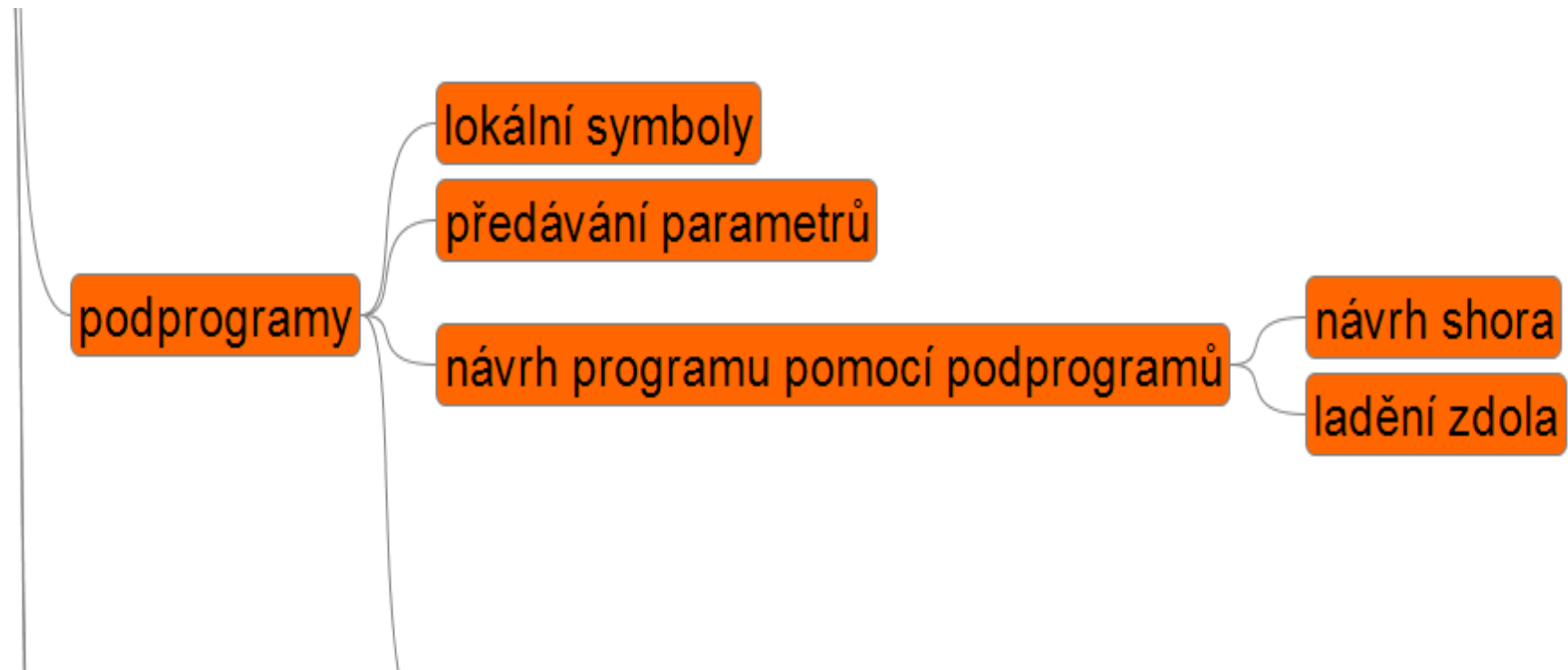
Těžká úloha:

**Napište program, který vytiskne
svůj vlastní zdrojový kód
(aniž by ho odněkud četl).**

Ten program nemusí dělat nic jiného.

Pokud se vá to podaří, pošlete mailem, Subj.: „SELF“.





Podprogramy

Příklad:

Vytiskněte tabulku malé násobilky ve tvaru

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X  X  1  2  3  4  5  6  7  8  9  10 X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X 1 X  1  2  3  4  5  6  7  8  9  10 X
X 2 X  2  4  6  8  10 12 14 16 18 20 X
X 3 X  3  6  9.....
X 4 X  4  8.....
X 5 X  5 10.....
.....
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

blok

má tvar

```
const   definice konstant;  
type    definice typů;  
var     deklarace proměnných;  
        deklarace procedur a funkcí;  
tělo
```

Kromě těla může kterákoliv část chybět

Program, procedura i funkce mají tvar

hlavička; blok

BP/FP nevyžaduje dodržení pořadí a libovolná část
z const, type, var, se může opakovat

např.

```
Program Muj;  
const max = 15;  
type  male = 1..max;  
var A, B: male;  
begin  
    read( A,B );  
    writeln( A+B )  
end.
```

Hlavička procedury

```
procedure  
    <identifikátor>  
    (<seznam formálních parametrů>)
```

například:

```
procedure A;  
procedure B( N: integer );  
procedure C( M,N: integer );  
procedure D( M: integer; c: char; k: integer );
```

Volání procedury

<identifikátor> (<seznam skutečných parametrů>)

například

A;

B(i);

C(k, 5);

D(2, 'a', 7);

Hlavička funkce

function

<identifikátor>

(<seznam formálních parametrů>)

: <typ výsledku>

například

```
function A: integer;
```

```
function B( N: integer ): boolean;
```

```
function C( M,N: integer ): char;
```

```
function D( M: integer; c: char ): char;
```

Příklad

```
function tg( x: real ): real;  
begin  
    tg := sin(x)/cos(x) { určení výsledné hodnoty }  
end
```

Přirozený požadavek:

Bez ohledu na průběh výpočtu funkce musí být
v jejím těle jejímu identifikátoru
(nejméně jednou) přiřazena výsledná hodnota.

Pozn.: C# kontroluje

Volání funkce

<identifikátor> (<seznam skutečných parametrů>)
... i uvnitř výrazu!

například

```
a := sqrt(z) ;  
b := CisloDne( den, mesic, rok ) ;  
c := 40+round( 40*sin(fi*2*PI/360) ) ;
```

Viditelnost identifikátorů

1. definice objektu musí předcházet jeho použití
2. viditelnost objektu je určena hierarchickou strukturou programu
3. jednoznačný význam identifikátoru v rámci
4. zastínění globáln(ějš)í definice lokáln(ějš)í definicí

```

program A
var i: integer; j: integer;
  procedure B( i: integer );
    function C( i: integer ): integer;
    begin
      C := i+j
    end;
  begin
    writeln( i );
    writeln( C(i) );
    i := i+1
  end;

  procedure D( x: integer );
  begin
    i := 1+x;
    B( x ); B( i )
  end;

begin
  i := 5;
  j := 2;
  D( 7 )
end.

```

Lokální symboly

- proměnné deklarované uvnitř podprogramů
- formální parametry
- a další

mají pouze lokální platnost

=>

- nelze s nimi pracovat v hlavním programu ani jinde mimo podprogram
- po vyvolání podprogramu (i opakovaném) hodnoty jeho lokálních proměnných **NEJSOU DEFINOVÁNY**

Shrnutí

- programátor může deklarovat nové podprogramy
- podprogram může mít své lokální proměnné (konstanty, typy, podprogramy)
- v hlavičce podprogramu mohou být deklarovány jeho formální parametry
- v příkazu volání uvedeme skutečné parametry
- jednou deklarovaný podprogram můžeme zavolat, kolikrát chceme

Proč podprogramy

- členění problému/programu na části, které můžeme řešit odděleně
- další úroveň oddělení CO TO DĚLÁ od JAK TO DĚLÁ
- skrývání proměnných atd., které mají význam jen pro řešení určité části
- **re-use** - možnost jednou vytvořený podprogram použít i v jiných programech

Předávání parametrů

a) hodnotou

b) odkazem

c) konstantní parametry (BP/FP)

d) výstupní parametry (C#)

e) výsledkem

f) jménem

g) ...

Skutečným parametrem může být

a) výraz

b) proměnná odpovídajícího* typu

* odpovídající...

HODNOTOU

nová proměnná, do které se na začátku dosadí hodnota skutečného parametru

ODKAZEM

jen nové jméno pro proměnnou předanou jako parametr

```
procedure MINMAX( p: POLE; var MIN,MAX: real );
```

var platí pro všechny následující parametry až do dvojtečky

příklad: P(A, B, C: integer)...

Volba způsobu předávání parametrů

1. Pokud má přenášet hodnotu ven

=> odkazem

2. Pokud má skutečným parametrem být výraz

=> hodnotou

3. Vstupní data jednoduchého typu zpravidla hodnotou

4. Velké proměnné zpravidla odkazem

Konstantní parametry

= předávané odkazem, ale překladač nedovolí dosadit

Příklad:

Napište program, který najde
10 nejčastějších slov v souboru

Programování shora (dolů)

= řešení úlohy rozkladem na pod-úlohy

= využívá volání (zatím neexistujících) podprogramů

Programování zdola (nahoru)

= vytváření podprogramů (řešení pod-úloh),
o kterých myslíme, že je budeme potřebovat
a následně z nich skládáme řešení větších
pod-úloh, až k hlavní úloze

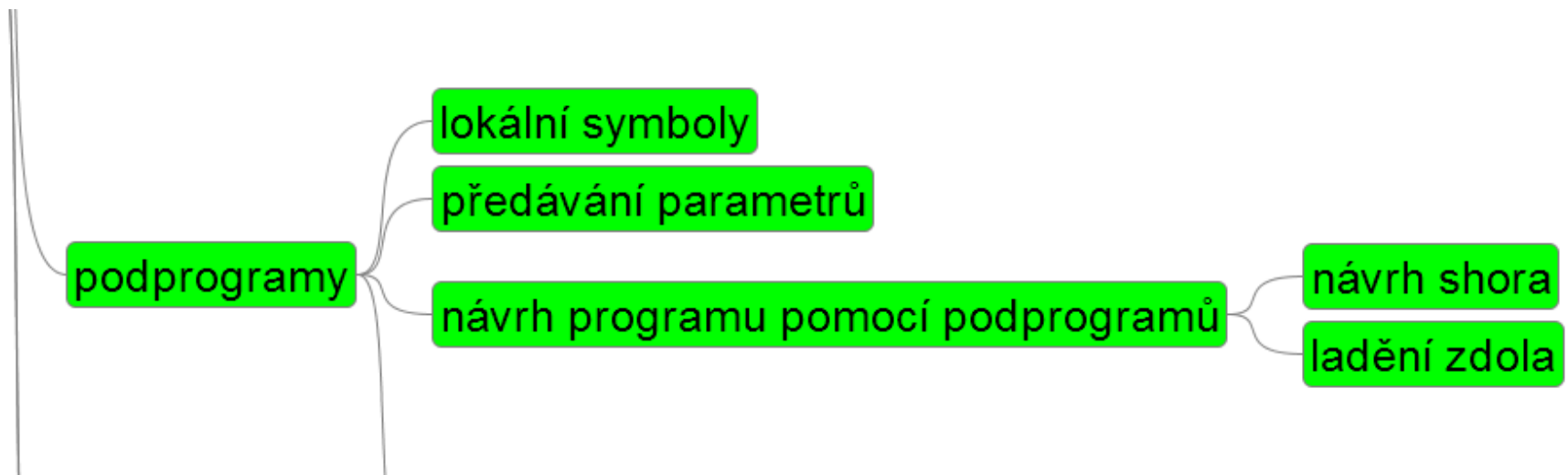
Ladění zdola

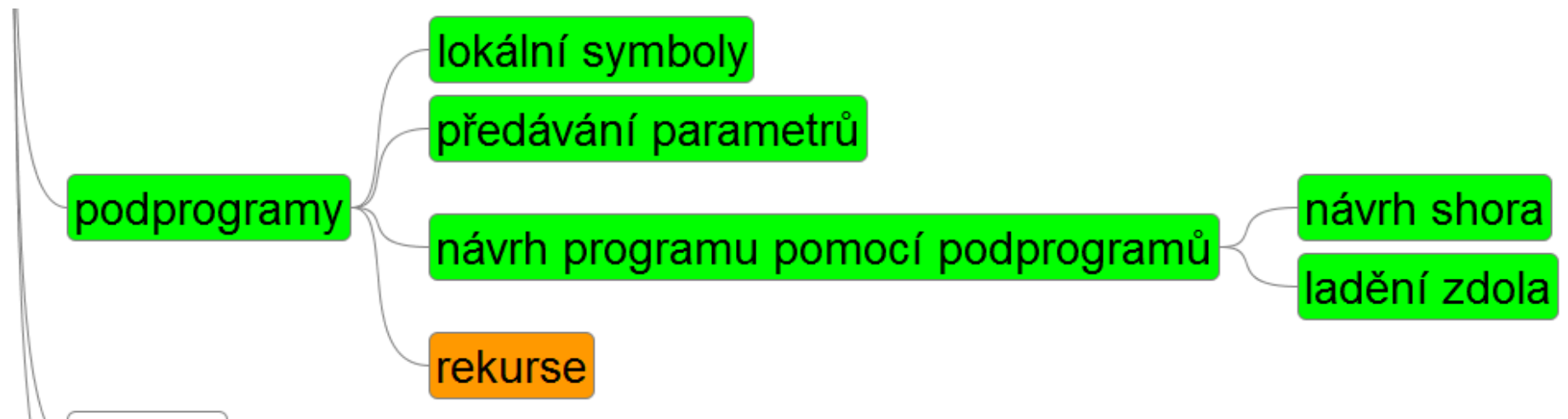
testovací podprogramy

Ladění shora

náhradní obsah podprogramů

Testovací podprogramy, vyhodnocování správnosti,
počítání chyb, automatické testování





Rekursivní volání

= volání sebe sama (přímo NEBO zprostředkovaně)

Rekursivní definice:

$$\begin{array}{ll} N! = 1 & \text{pro } N \leq 1 \\ N \times (N-1)! & \text{pro } N > 1 \end{array}$$

Rekursivní, *adj.*: viz Rekursivní

V Pascalu:

```
function fakt(N: integer): integer;  
begin  
    if N <= 1 then fakt := 1  
    else fakt := N*fakt(N-1)  
end;
```



rekursivní volání

opětovné (rekursivní volání)

=> nový exemplář funkce fakt
s novými parametry
s novými proměnnými...

?

```
function fakt2( N: integer ): integer;  
begin  
    fakt2 := N*fakt2( N-1 )  
end;
```

? co udělá tato procedura:

```
procedure P;  
var  c: char;  
begin  
    read( c );  
    if c<>' ' then  
    begin  
        P;  
        write( c )  
    end  
end;  
end;
```

Klady a zápory rekurse

- + zjednodušení algoritmu (některých)
- zpravidla náročnější na čas i paměť
-

Odstrašující příklad:

Fibonacciho posloupnost: $a_i = a_{i-1} + a_{i-2}$

```
function fib( i: integer ): integer;  
begin  
    if i <= 1 then fib := 1  
        else fib = fib(i-1)+fib(i-2)  
end
```

Náprava ukládáním výsledků...

Příklady

kombinace

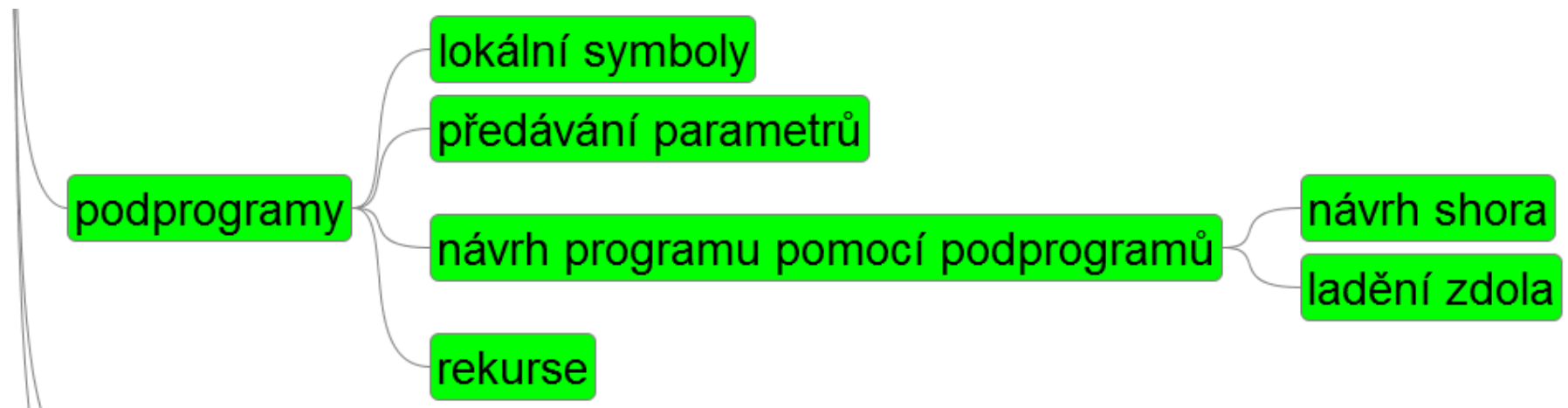
Hanoj

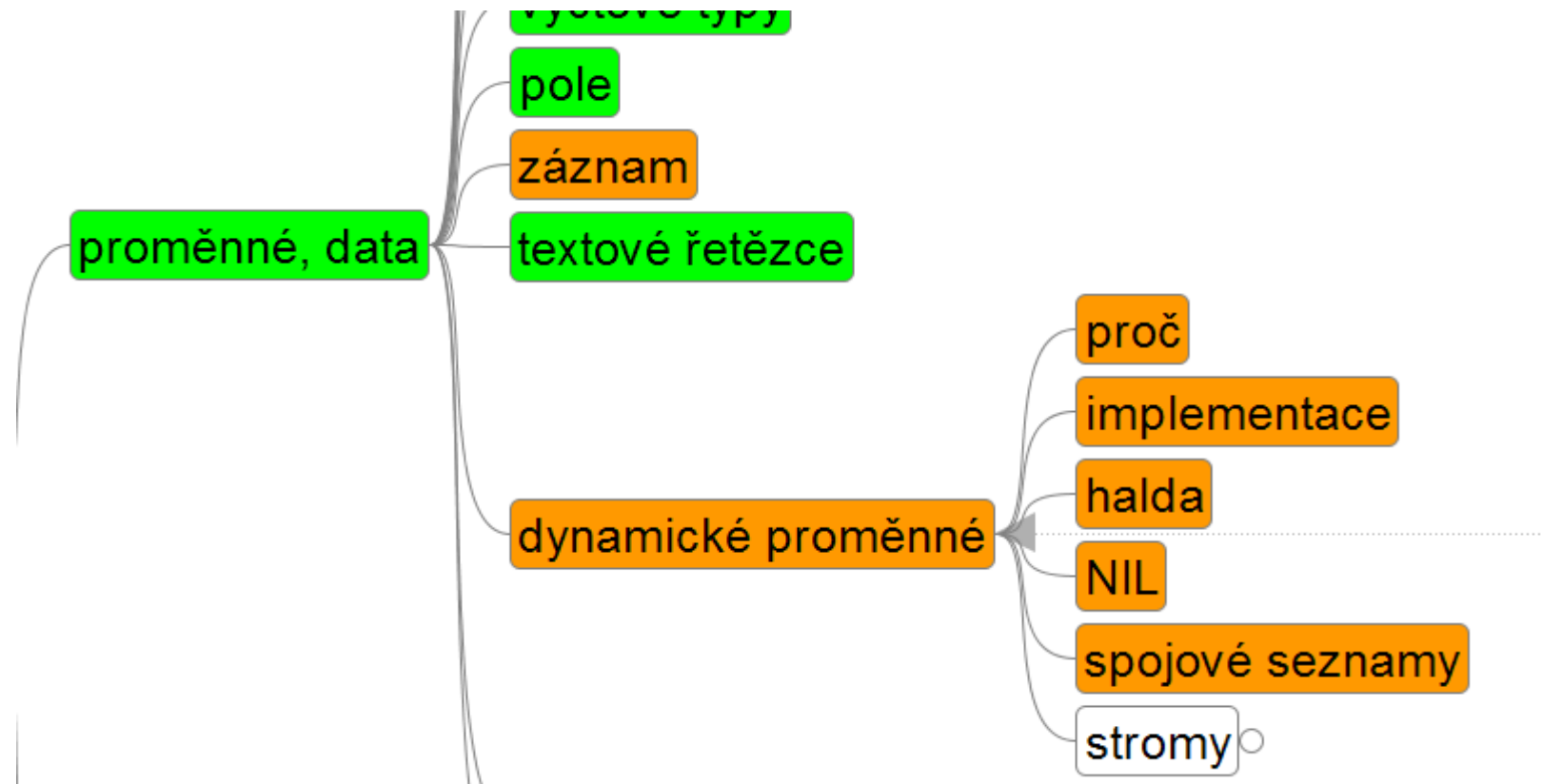
loupežníci

kůň

dámy

BP: call stack





typ záznam (record)

další způsob strukturování dat

```
record <seznam položek> end
```

```
type
```

```
    Komplex = record  
                Re, Im: real  
            end
```

```
var
```

```
    K: Komplex;  
    . . .  
    K.Re := 1.00;  
    K.Im := -5.25;
```

Dynamické proměnné

Strukturované příkazy

jednoduchý příkaz

složený příkaz

if, case

for-cyklus

while/repeat cyklus

procedura, funkce

Strukturovaná data

skalární proměnná

záznam

~~—záznam s variantami—~~

pole

soubor

?

Procedura, funkce:

Nevolá sama sebe

= nezajímavá.

Volá sama sebe („rekurzivní“) => data obsahující sama sebe!

Příklad

type

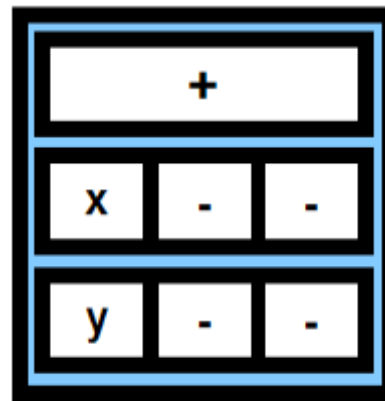
TVyraz = record

operator: char; { +, -, *, / nebo promenna }

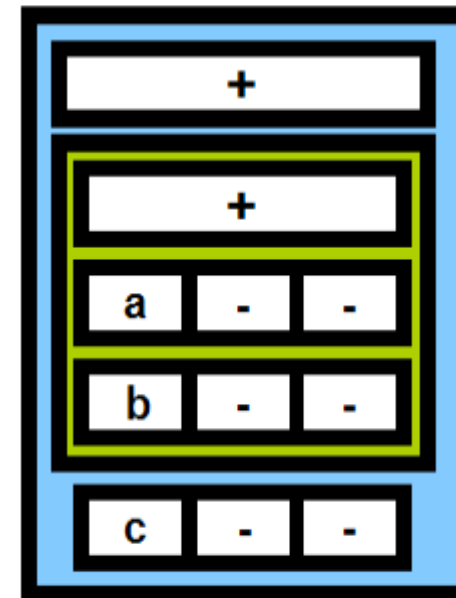
pv1, pv2: TVyraz

end;

$x+y$

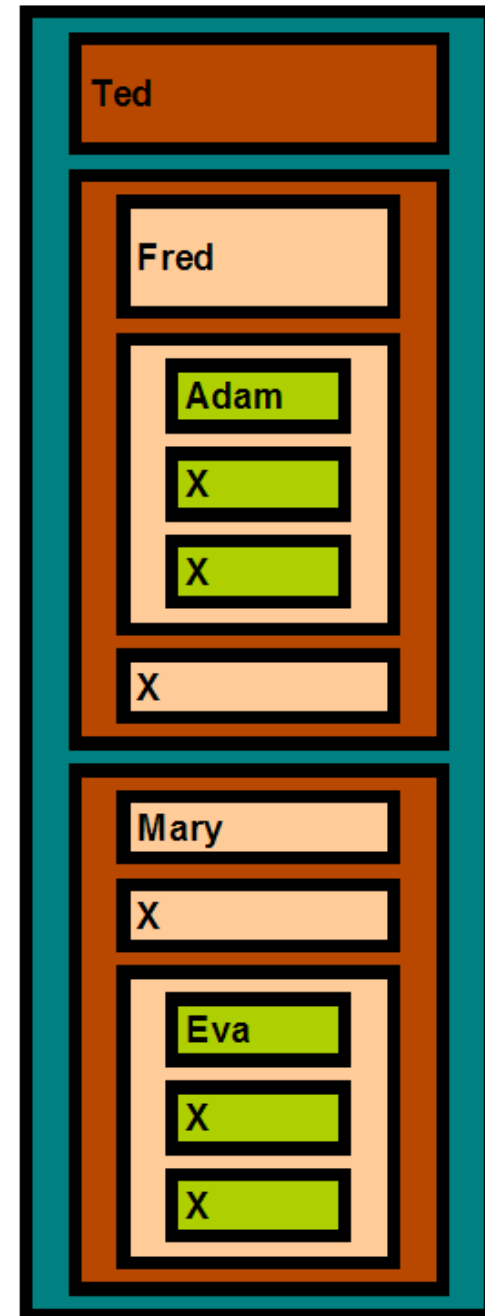


$a+b+c$



Příklad

```
type
  TRodokmen = record
    jmeno: string;
    { X znamena neznamy }
    otec, matka: TRodokmen
  end;
```



Zápis rekursivních dat

Ted

Fred

Adam

X

X

X

Mary

X

Adam

X

X

pomocí závorek:

((Ted, (Fred, (Adam, X, X) , X) , (Mary, X, (Eva, X, X))))

Typická vlastnost prvku:

VARIANTNOST

rekursivní procedura bez možnosti větvení
taky nikdy neskončí

PROMĚNNÁ VELIKOST

Důsledek:

Takovým proměnným NELZE při překladu
přiřadit pevný počet paměťových míst.

Důsledek důsledku:

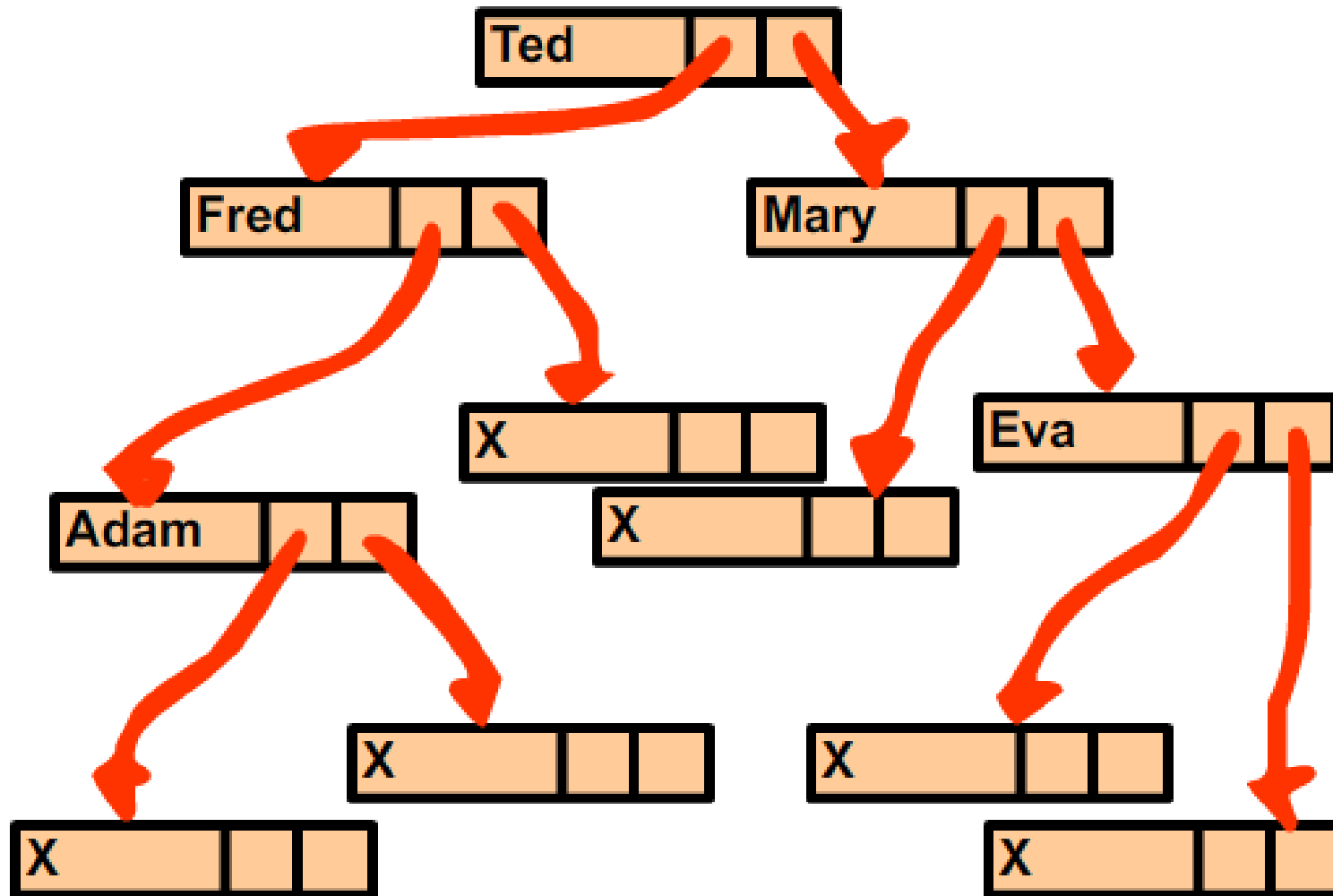
Kompilátor nemůže adresovat jednotlivé složky
takových rekursivních proměnných,
protože neví, KDE (a JESTLI VŮBEC) budou.

Jak se to tedy dělá?

- * paměť se přiděluje dynamicky jednotlivým složkám
- * kompilátor místo paměti pro další složku přidělí jen místo pro její ADRESU.

ADRESA = UKAZATEL = POINTER = SMERNÍK
= מצביע = نقطة = ...

Příklad



Jak je to v Pascalu

typ „ukazatel na typ TMujTyp“

type

PMujTyp = ^TmujTyp;

Ukazatel získá hodnotu, když vznikne (dostane paměť) prvek,
na který má ukazovat

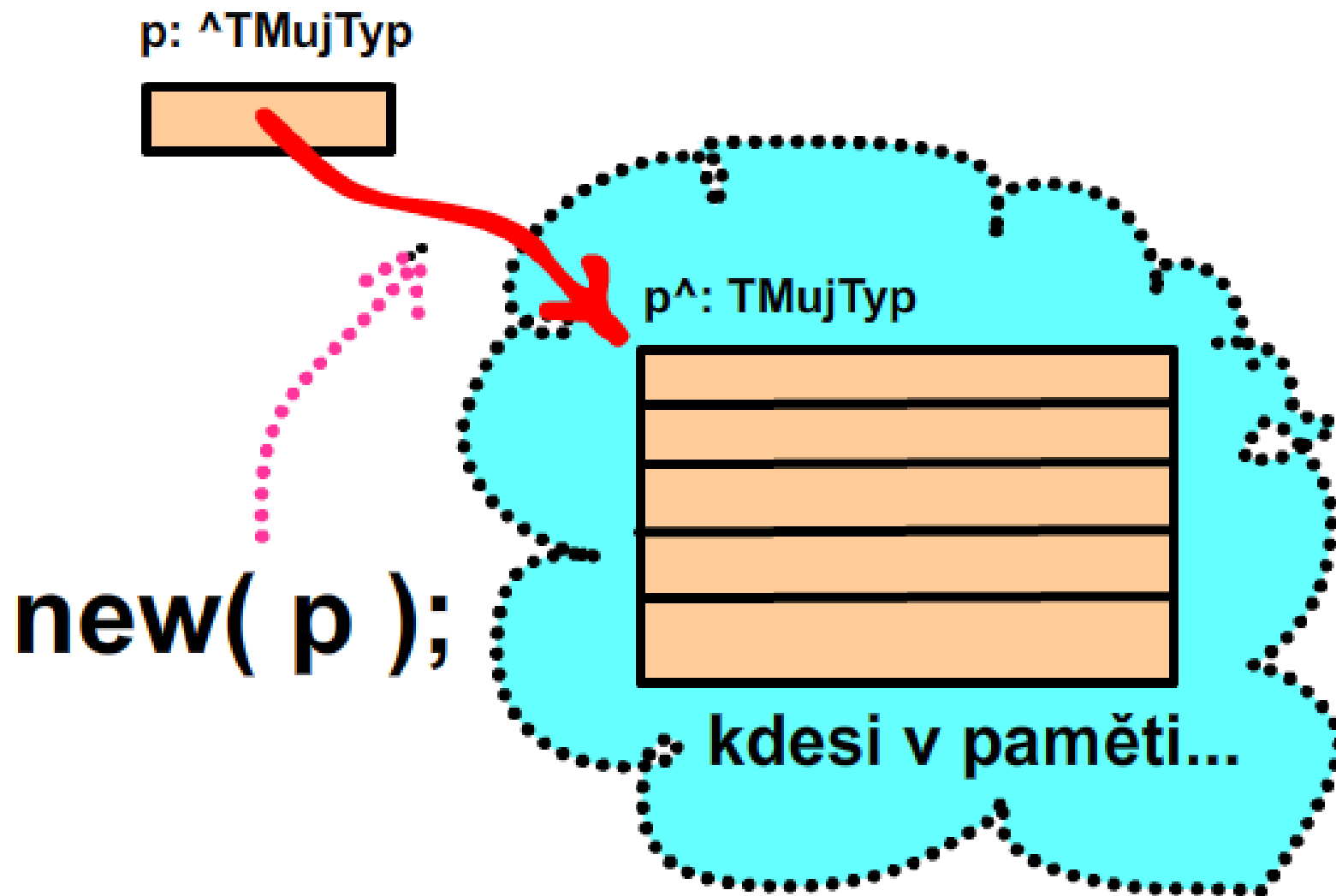
= voláním procedury new

var

p: PMujTyp; { stejne jako p: ^TMujTyp }

...

new(p);



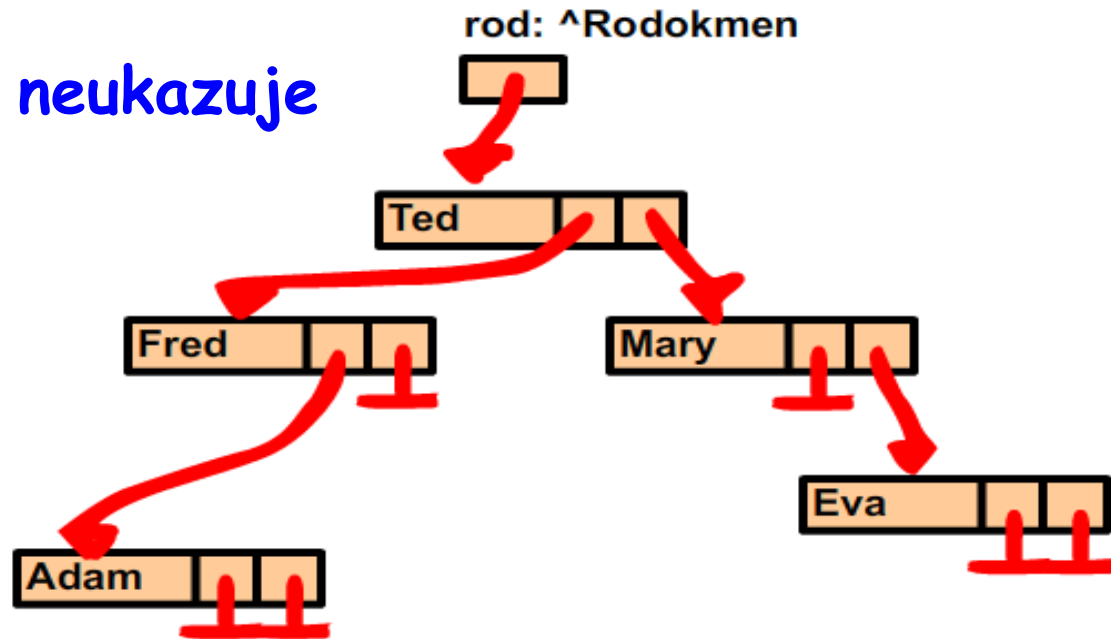
!!! NEW vytváří proměnné, které nemají vlastní jméno !!!

Ukazování ukazatelem

```
var rod: ^TRodokmen;  
...  
    new( rod ); { ted rod ukazuje na (novou)  
                  promennou typu TRodokmen:  
                      rod^ }  
rod^.Jmeno := 'Ted';  
rod^.otec  := ...
```

Konstanta NIL

ukazatel, který nikam neukazuje



```
new( rod );
rod^.Jmeno := 'Ted';
new( rod^.Otec );
rod^.Otec^.Jmeno := 'Fred';
new( rod^.Otec^.Otec );
rod^.Otec^.Matka := NIL;
...
```

Zrušení dynamické proměnné

```
dispose ( rod ) ;
```

Uvolní (vrátí) obsazenou paměť.

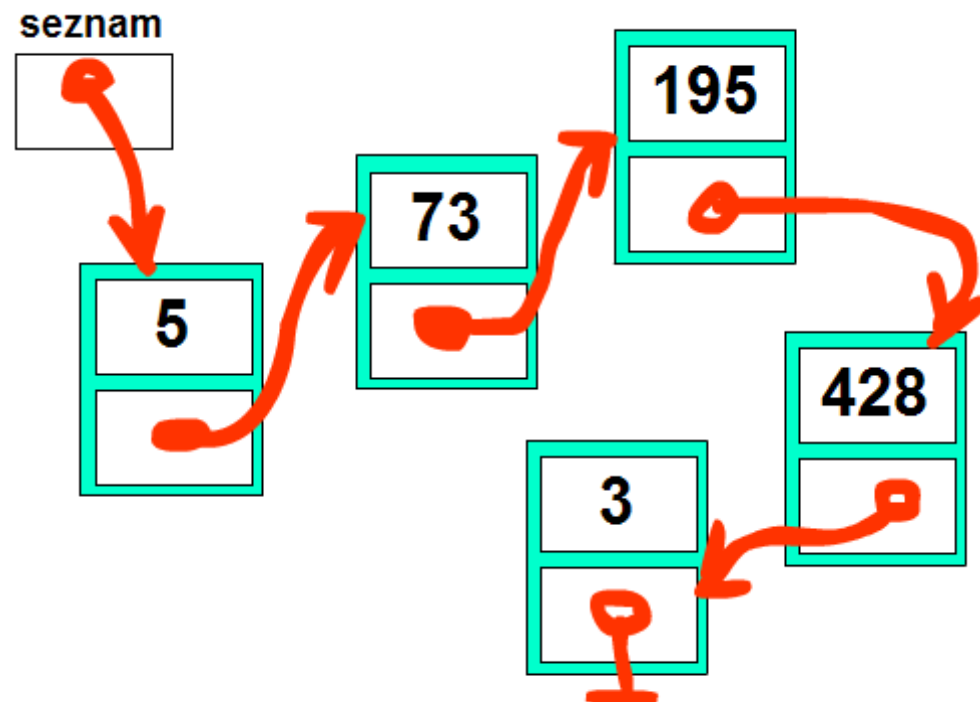
POZOR: Nedosazuje do té proměnné !

V C# automatická správa paměti (garbage/collector).

? Kam ukazuje NIL ?

Spojový seznam (jednosměrný, lineární)

```
type
  PPrvek = ^TPrvek;
  TPrvek = record
    Hodnota: NejakyTyp;
    Dalsi: PPrvek
  end;
var
  seznam: PPrvek;
```



Základní kroky

projít

přidat na začátek

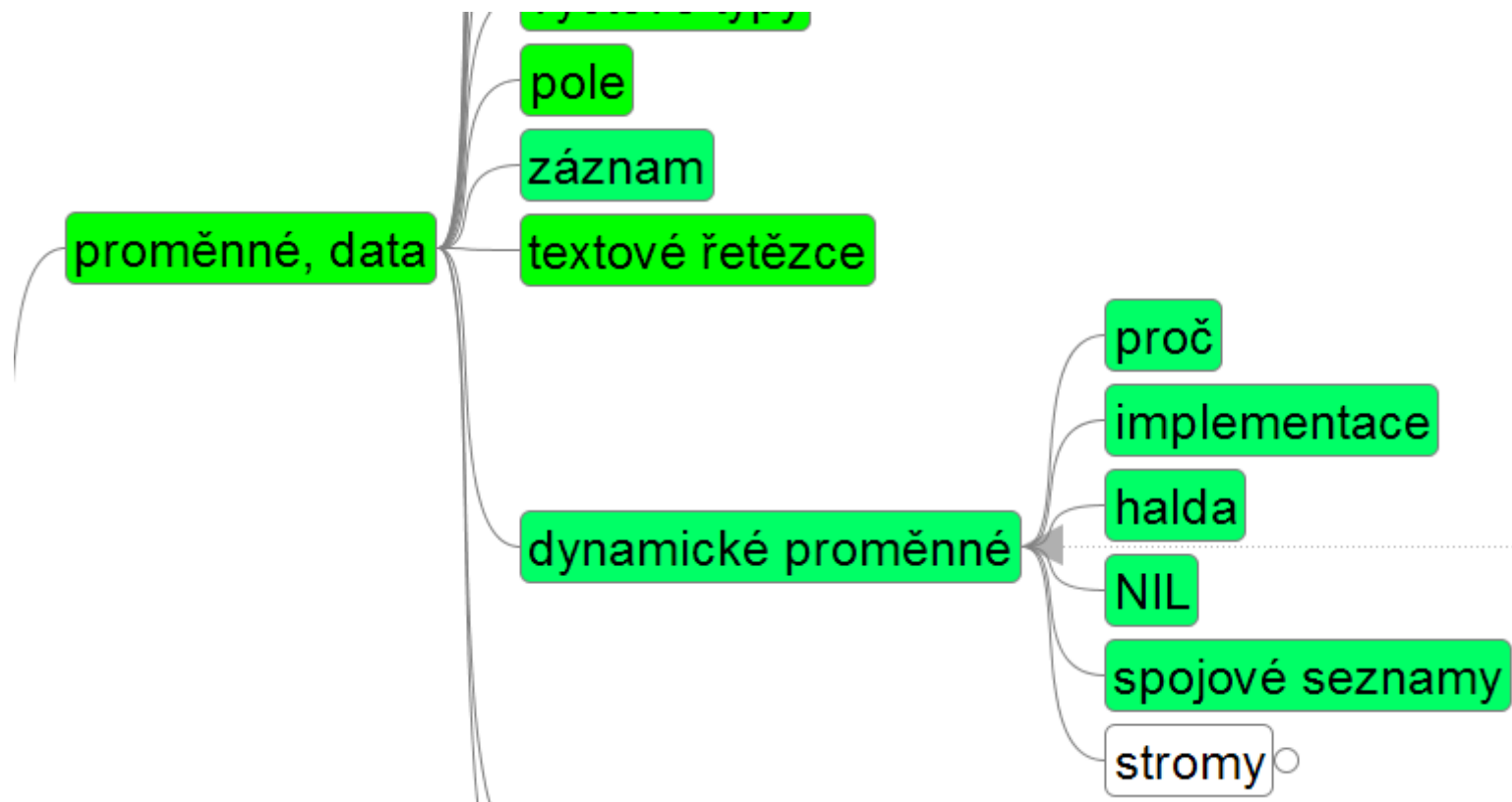
přidat na konec

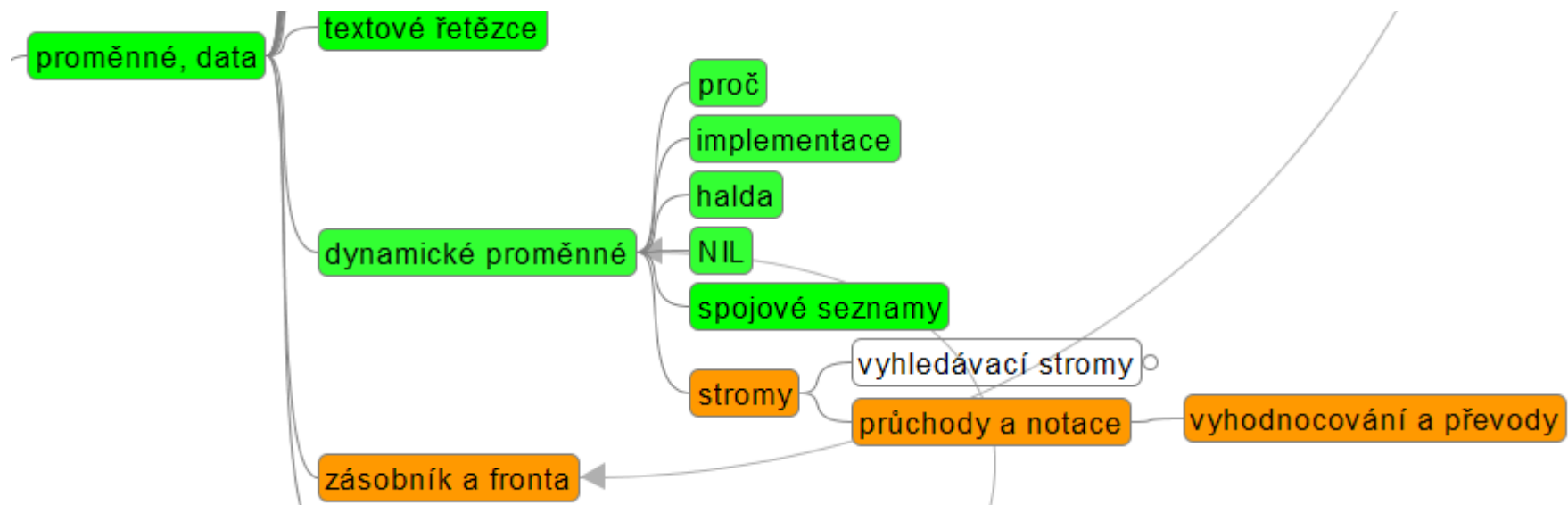
zrušit první

zrušit poslední

zrušit obecně...

vyrobit seznam pomocí funkce





Spojové seznamy: odstraňování potíží

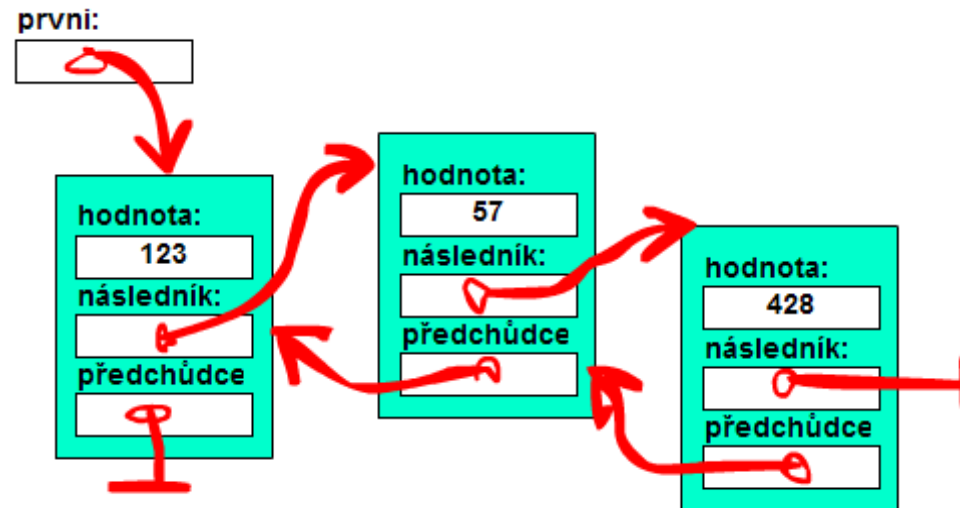
Potíž:

Nalezení předchozího prvku

Řešení:

Obousměrný seznam.

Kromě odkazu na následníka si ukládáme i odkaz na předchůdce.



Potíž:

Zvláštní zacházení s prvním prvkem

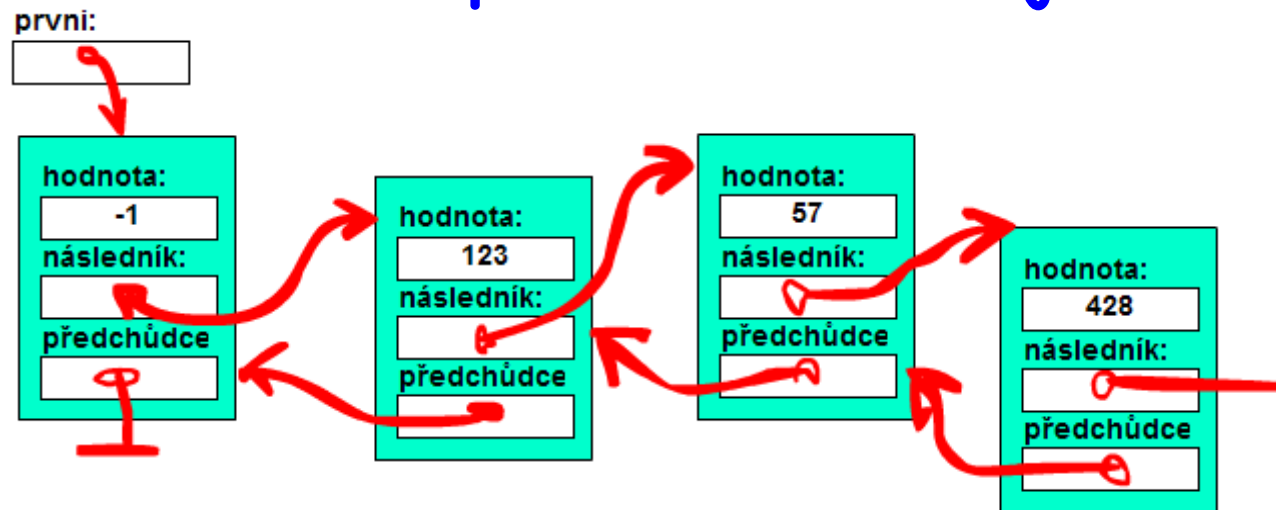
Řešení:

Seznam s hlavou.

Na začátek seznamu přidáme „nultý“ prvek HLAVU.

Každý (řádný) prvek má svého předchůdce.

Nikdy nemusíme měnit proměnnou ukazující na seznam.



Potíž:

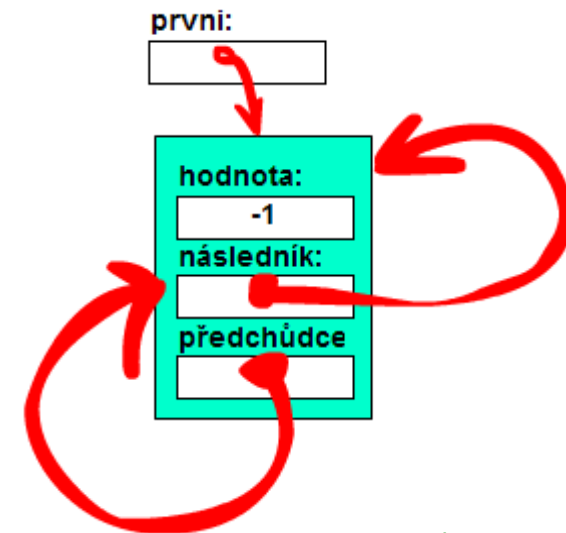
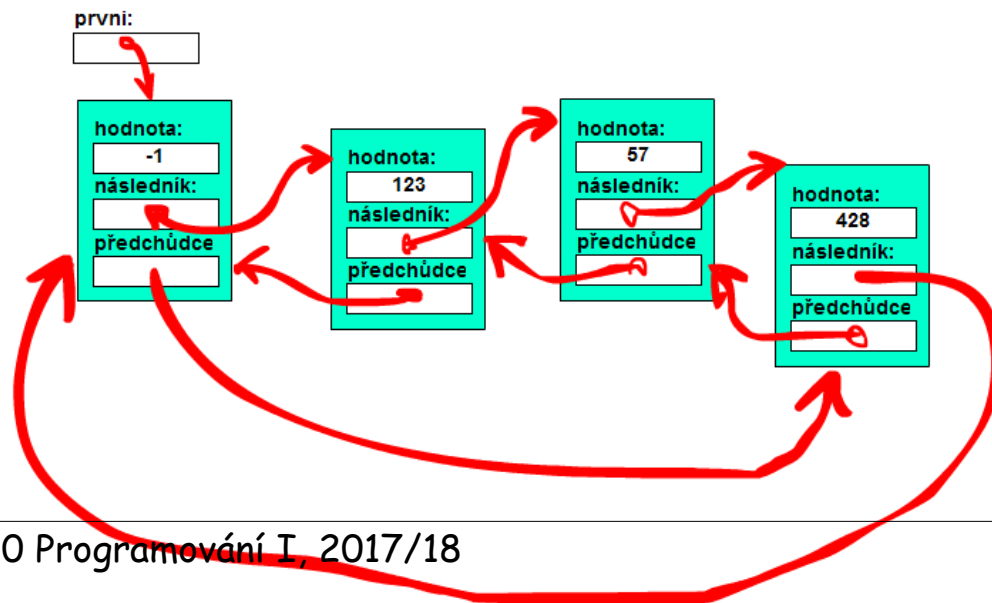
Zvláštní zacházení s posledním prvkem

Řešení:

Cyklický seznam.

Místo NIL ukazatel na první prvek
(je jedno, s čím porovnáváme).

Každý prvek má svého následníka.



Zarážka, NIL

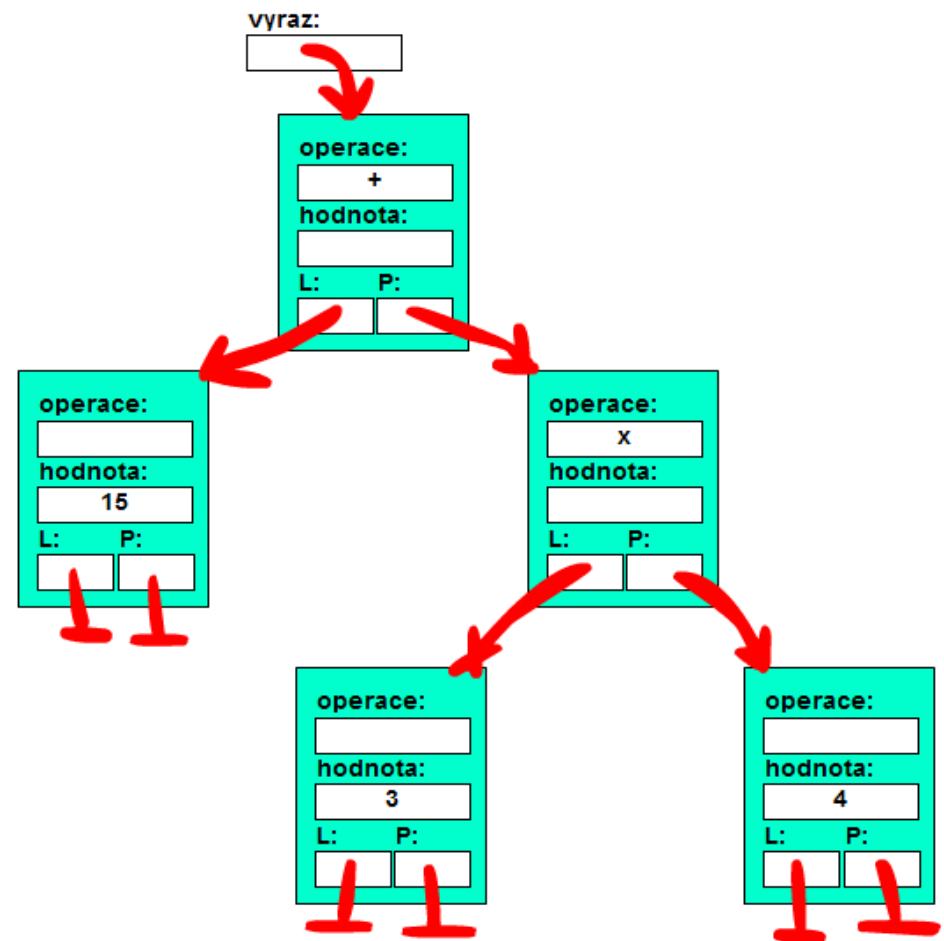
Složitější dynamické struktury stromy

Aritmetický výraz

```
type PExpr = ^TExpr;  
TExpr = record  
    operace: char;  
    hodnota: integer;  
    L, P: PExpr  
end;
```

vyhodnocení výrazu

výpis / průchod



Průchody a algebraické notace

```
procedure Projdi ( V: PVyraz );  
begin  
    if V^.L=NIL then write( V^.hodnota )  
        else  
            begin  
                write( V^.operace );  
                Projdi( V^.L );  
                write( V^.operace );  
                Projdi( V^.P );  
                write( V^.operace );  
            end  
        end;  
end;
```

PREORDER/PREFIXINORDER/INFIXPOSTORDER/POSTFIX

Vyhodnocení výrazu v notaci POSTFIX

Zásobník

Když přečteš...

...číslo: ulož do zásobníku

...operaci:

- 1) vyber ze zásobníku operandy
- 2) proved' na ně operaci
- 3) výsledek ulož do zásobníku

Vyhodnocení výrazu v notaci PREFIX

Rekurse

Když přečteš...

...číslo: vrat' jeho hodnotu

...operaci: proved' ji na dvě volání
sama sebe

```
if zn='+' then
```

```
    Hodnota := Hodnota + Hodnota
```

Vyhodnocení výrazu v notaci INFIX[1]

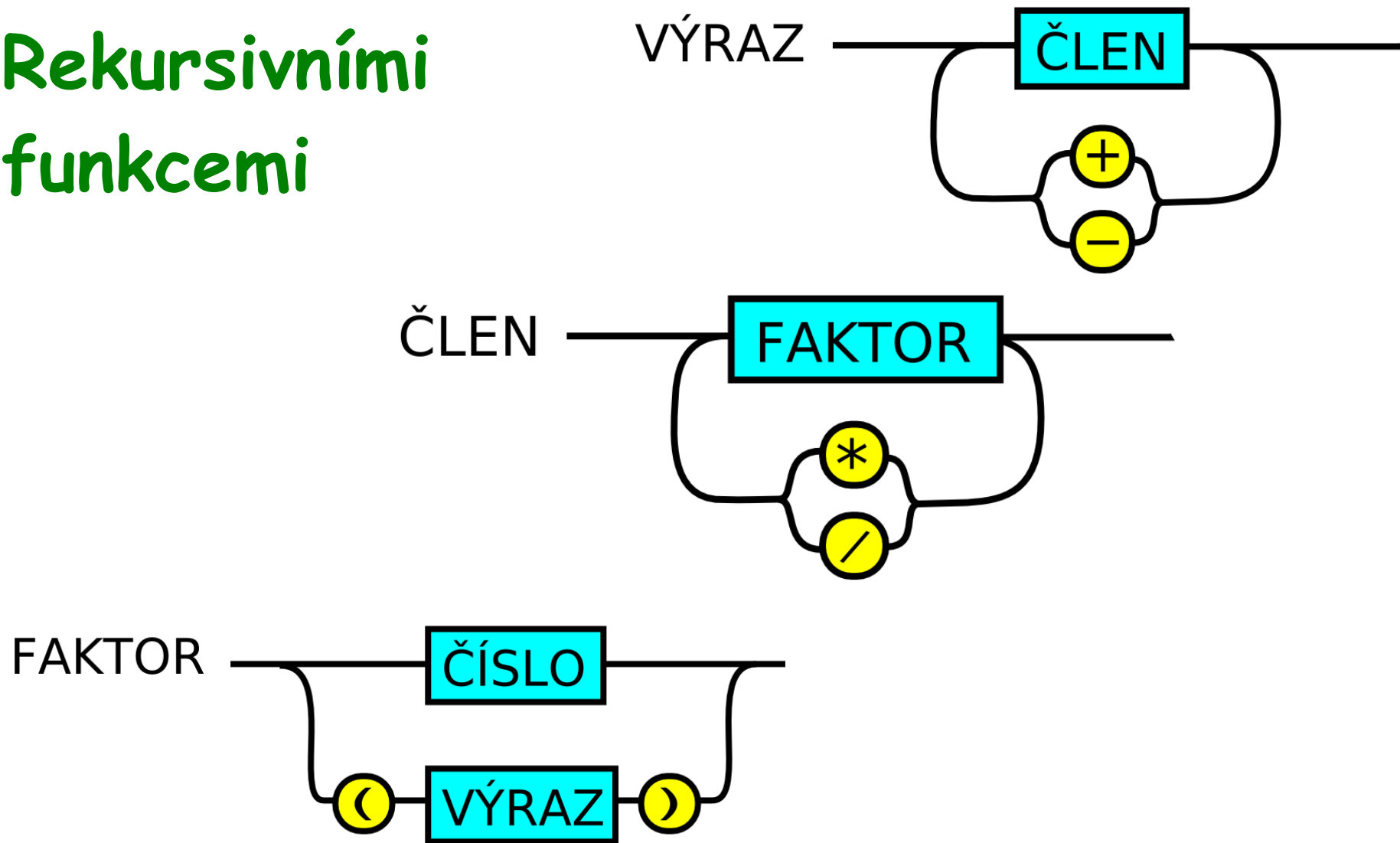
Rozkladem na podvýrazy.

Najít operaci s nejmenší prioritou
a tu provést na části výrazu:

$$\begin{array}{l} 12 * (7+3) - (2*4-5) / 8 + 19*7 \\ 12 * (7+3) - (2*4-5) / 8 + \underline{19} * \underline{7} \\ \underline{12} * (7+3) - (2*4-5) / \underline{8} + 19*7 \\ 12 * (\underline{7+3}) - (\underline{2*4-5}) / 8 + 19*7 \\ 12 * (\underline{7} + \underline{3}) - (\underline{2*4} - \underline{5}) / 8 + 19*7 \end{array}$$

Vyhodnocení výrazu v notaci INFIX[2]

Rekursivními
funkcemi



Abstraktní datový typ

je definován rozhraním (interface)

rozhraní může zahrnovat

- data
- procedury a funkce (souhrnně „metody“)

Příklad

typ Seznam:

- function JePrazdny: boolean
- procedure Pridej(prvek: TPrvek)
- procedure Vyber(var prvek: TPrvek)

Zajímá nás rozhraní a ne to,
jak je to rozhraní realizováno uvnitř
(pole, spojový seznam, soubor...).

Datová struktura FRONTA

FIFO - First In First Out

Datová struktura ZÁSObNÍK

LIFO - Last In First Out

...jsou zvláštní typy Seznamu

(stejně jako jezevčík je pes je savec je obratlovec je živočich).

=> mezi abstraktními datovými typy může být vztah „A je zvláštním případem (specializací) B“. („ISA“)

A.D.T. v různých jazycích různě, zatím neřešíme.

Realizace zásobníku a fronty

Pomocí pole

- zásobník
- fronta
 - kruhová fronta

Pomocí spojových seznamů

Pomocí souborů...

Prioritní fronta.

Obecný algoritmus prohledávání stavů

```
1. SeznamNeprozkoumanýchStavů  
   := { PočátečníStav }
```

```
2. while not KONEC do  
   if Seznam.Prázdný then  
     => KONEC, řešení neexistuje
```

```
   s := Seznam.Vyber;  
   pro každý t dostupný jedním krokem z s:  
     if t je cílový stav then  
       => KONEC, řešení nalezeno  
     else  
       Seznam.Přidej( t )
```

Zajímavé pozorování

Když Seznam je FRONTA:

Prohledávání do šířky.

(vlna)

Když Seznam je ZÁSOBNÍK:

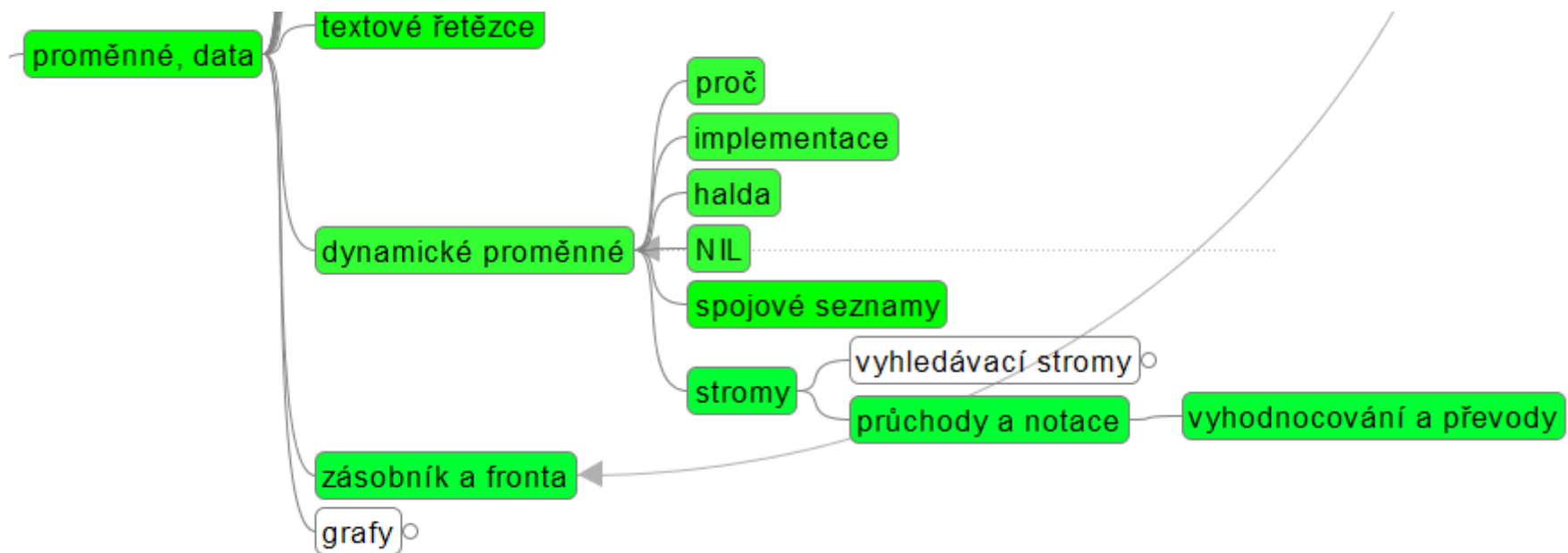
Prohledávání do hloubky.

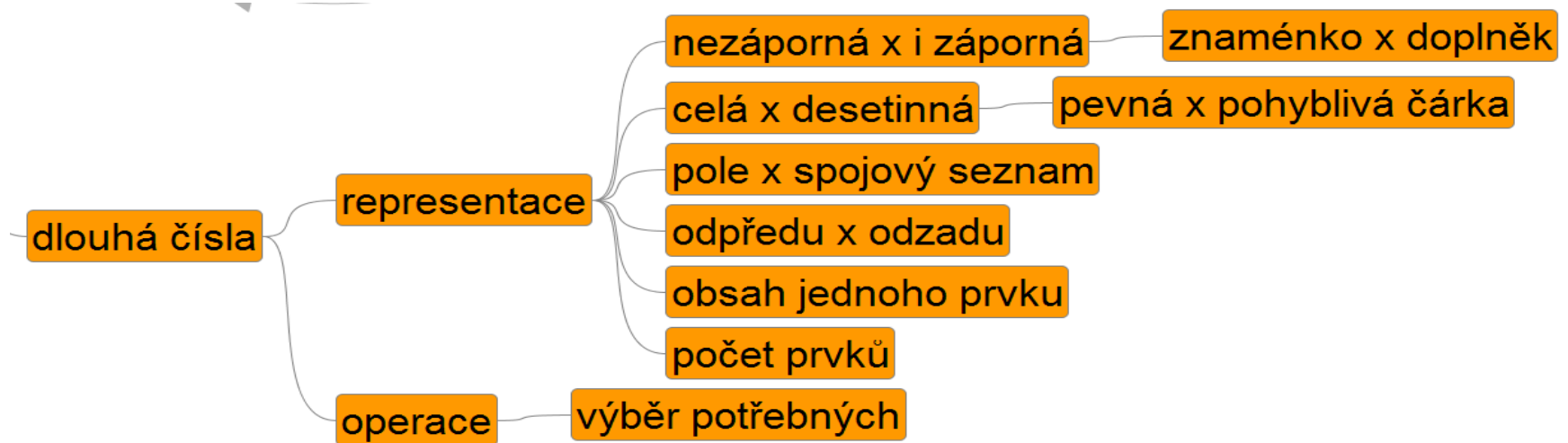
(backtracking)

Příklad: program zásobník, fronta, na šachovnici

Kdy prohledávat do šířky a kdy do hloubky

Prohledávání do rostoucí hloubky





Dlouhá čísla

když nám nestačí standardní číselné typy...

Krok 1: Návrh reprezentace

- a) nezáporná x i záporná
 - a2) se znaménkem x doplněk
- b) celá x desetinná
 - b2) pevná x pohyblivá řádová čárka
- c) pole x spojový seznam
- d) odpředu x odzadu
- e) obsah jednoho prvku
- f) délka
 - f1) kolik míst navíc
 - spočítat (a ještě zkusit)

Krok 2: Naprogramovat provádění operací

a) jaké operace potřebujeme

a2) jaké operace **DOOPRAVDY** potřebujeme

b) jak je naprogramovat

Příklad: e na 1000 desetinných míst

- odhad počtu kroků
- potřebné operace

Příklad: e na 1000 desetinných míst

- odhad počtu kroků
- potřebné operace
 - dosazení
 - sčítání
 - dělení integer-em

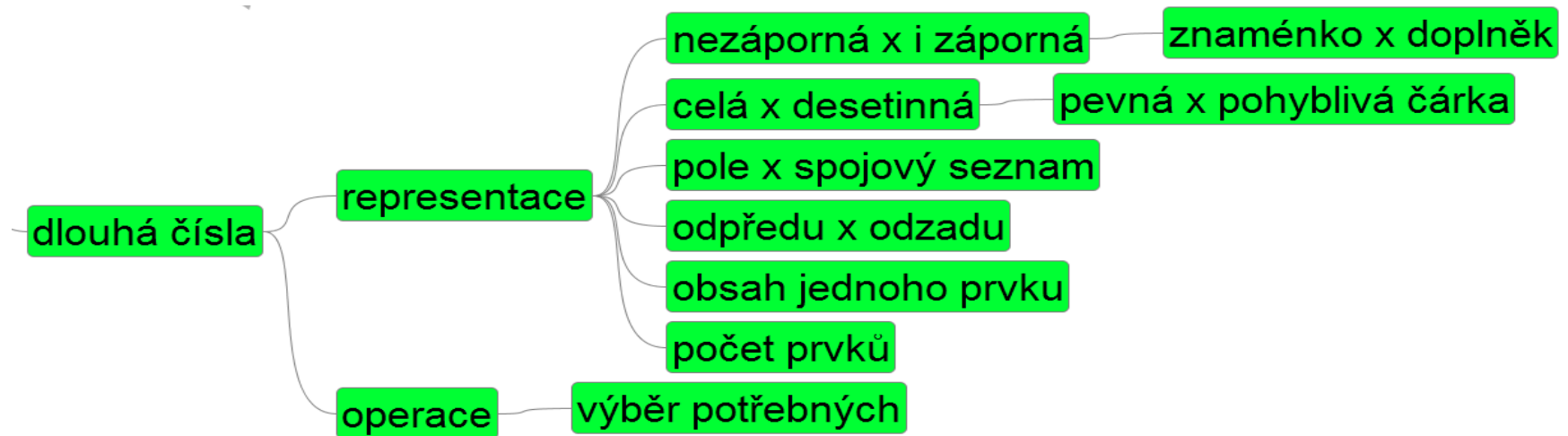
Příklad: e na 1000 desetinných míst

- odhad počtu kroků
- potřebné operace
 - dosazení
 - sčítání
 - dělení integer-em
 - přeskočení nul
 - test konce

odečítání
- doplněk

násobení

dělení



Třídění výběrem (přímý výběr, SelectSort)

Algoritmus:

Pole se dělí na setříděný úsek (vlevo) a neseříděný úsek (vpravo).

Na začátku tvoří všechny prvky neseříděný úsek.

V neseříděném úseku pole se vždy najde nejmenší prvek a vymění se s prvním prvkem tohoto úseku, tím se neseříděný úsek zleva zkrátí o jeden prvek.

Realizace: na místě v jediném poli – minima se postupně ukládají zleva, výměnou s původními hodnotami.

7 4 2 9 5

2 4 7 9 5

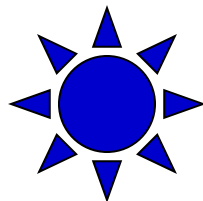
2 4 7 9 5

2 4 5 9 7

2 4 5 7 9

modře – hotovo (setříděný úsek)

červeně – minimum ze zbývajících hodnot



```

const N = 1000;
type Pole = array [1..N] of integer;
var A: Pole;                                {tříděné pole}
      i, j, k: integer;                       {indexy prvků}
      x: integer;                             {pro výměnu prvků}

...
for i:=1 to N-1 do
begin  {umístit číslo na pozici i v poli A}
  k:=i;
  for j:=i+1 to N do
    {vyhledání minima v úseku A[i..N]}
    if A[j] < A[k] then k:=j;
  if k > i then  {výměna prvků s indexy i, k}
    begin x:=A[k]; A[k]:=A[i]; A[i]:=x end
end

```


Totéž ve tvaru procedury:

```
procedure PrimyVyber(var A: Pole);  
var i, j, k: integer; {indexy prvků}  
    x: integer;        {pro výměnu prvků}  
begin  
    for i:=1 to N-1 do    {umístit číslo na pozici i}  
        begin  
            k:=i;  
            for j:=i+1 to N do    {vyhledání minima}  
                if A[j] < A[k] then k:=j;  
            if k > i then    {výměna prvků s indexy i, k}  
                begin x:=A[k]; A[k]:=A[i]; A[i]:=x end  
            end  
    end;    {procedure PrimyVyber}
```

Třídění vkládáním (přímé zatříd'ování, InsertSort)

Algoritmus:

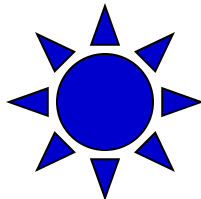
Pole se dělí na setříděný úsek (vlevo) a neseříděný úsek (vpravo). Na začátku je setříděný úsek tvořen pouze prvním prvkem pole. První prvek neseříděného úseku se vždy zařadí podle velikosti do setříděného úseku, tím se neseříděný úsek zleva zkrátí o jeden prvek.

Realizace: na místě v jediném poli – prvky setříděného úseku se posouvají o jednu pozici doprava, dokud je třeba.

7 4 2 9 5
4 7 2 9 5
2 4 7 9 5
2 4 7 9 5
2 4 5 7 9

modře – hotovo (setříděný úsek)

červeně – první ze zbývajících hodnot
(zatříd'ovaný prvek)



```

procedure PrimeVkladani(var A: Pole);
var i, j: integer;           {indexy prvků}
    x: integer;               {pro výměnu prvků}
begin
    for i:=2 to N do         {zatřídíme číslo z pozice i}
        begin
            x:=A[i];
            j:=i-1;
            while (j > 0) and (x< A [j]) do
                begin           {hledání správné pozice}
                    A[j+1]:=A[j];
                    j:=j-1;
                end;
            A[j+1]:=x
        end
    end; {procedure PrimeVkladani}

```

3. Bublínkové třídění (třídění záměnami, BubbleSort)

Pozorování:

ve vzestupně seřazeném poli stojí vždy menší prvek před větším

Algoritmus:

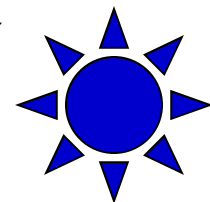
Opakovaně se prochází pole a porovnávají se dvojice sousedních prvků, v případě špatného uspořádání (větší prvek před menším) se tyto prvky spolu vymění – tím se v poli vytváří seřazený úsek.

Možnosti realizace:

průchody polem zleva doprava → seřazený úsek vzniká odzadu
(od největších prvků)

průchody polem zprava doleva → seřazený úsek vzniká odpředu
(od nejmenších prvků)

každý průchod může být vždy o jeden krok kratší než předchozí



```

procedure BublinskoveTrideni (var A: Pole);
var i, j: integer;           {indexy prvků}
    x: integer;               {pro výměnu prvků}
begin
    for i:=1 to N-1 do
        for j:=1 to N-i do
            if A[j] > A[j+1] then {vyměnit sousední prvky}
                begin x:=A[j]; A[j]:=A[j+1]; A[j+1]:=x end
        end; {procedure BublinskoveTrideni}

```

Možnosti zrychlení:

- při příštím průchodu polem stačí jít jen do místa poslední uskutečněné výměny
 (→ rychlejší zkracování průchodů, méně průchodů)
- třídění přetřásáním – pole se prochází střídavě zleva a zprava

Zásobník (LIFO) – realizace v poli

- zásobník reprezentován polem **Z: array [1..N] of T**
- N je kapacita zásobníku (kolik prvků může obsahovat najednou)
- dno zásobníku: $Z[1]$
- vrchol zásobníku (kde se přidávají a odebírají prvky): $Z[V]$
- prázdný zásobník (také inicializace): $V=0$

Vložení hodnoty X do zásobníku (předpokládáme, že je tam na ni místo, tzn. $V < N$, jinak by bylo třeba přidat příslušný test):

`inc(V) ; Z[V] := X ;`

Odebrání hodnoty ze zásobníku a vložení do proměnné X (předpokládáme, že tam nějaká hodnota je, tzn. $V > 0$, jinak by bylo třeba přidat příslušný test):

`X := Z[V] ; dec(V) ;`

Obě operace mají konstantní časovou složitost.

Fronta (FIFO) – realizace v poli

- fronta reprezentována polem **F: array [1..N] of T**
- N je kapacita fronty (kolik prvků může obsahovat najednou)
- konec fronty (místo příchodu do fronty): **F[Prich]**
- začátek fronty (místo odchodu z fronty): **F[Odch]**
- prázdná fronta (také inicializace): **Prich=0**

Vložení hodnoty X do fronty (předpokládáme, že je tam na ni místo, tzn. $\text{Prich} < N$, jinak by bylo třeba přidat příslušný test):

```
inc(Prich) ; F[Prich] := X;
```

Odebrání hodnoty z fronty a vložení do proměnné X (předpokládáme, že tam nějaká hodnota je, tzn. $\text{Odch} \leq \text{Prich}$, jinak by bylo třeba přidat příslušný test):

```
X := F[Odch] ; inc(Odch) ;
```

Problém: Obsazená část pole F se posouvá k vyšším indexům, časem není kam přidat další přicházející prvek, i když fronta není moc dlouhá a v poli F jsou volná místa po odebraných prvcích.

Řešení:

1. Při každém odebrání prvku z fronty se obsazená část pole F posune o 1 místo doleva (tzn. stále platí $Odch = 1$, proměnou $Odch$ tedy ani nepotřebujeme) \rightarrow časová složitost odebrání bude $O(N)$:

```
X := F[1];  
for I := 2 to Prich do F[I-1] := F[I];  
dec(Prich);
```


2. Posunutí obsazené části pole F se uskuteční jen když je to nutné, tzn. není-li místo na vkládaný prvek \rightarrow posunuje se méně často a na větší vzdálenost (vždy až do pozice $Odch = 1$).

Časová složitost odebrání prvku z fronty zůstane tedy konstantní, časová složitost vkládání bude v nejhorším případě lineární (ale velmi často se provede vložení prvku v konstantním čase).

```
if Prich = N then {plno, nejdřív posunout}  
  begin  
    for I := Odch to Prich do  $F[I - Odch + 1] := F[I]$  ;  
    Prich := Prich - Odch + 1 ;  
    Odch := 1  
  end ;  
inc(Prich) ;  
 $F[Prich] := X$  ;
```

3. Pole F se chápe **cyklicky**, za $F[N]$ následuje opět prvek $F[1]$, obsazená část pole se nikdy neposouvá \rightarrow časová složitost obou operací zůstává ***konstantní***.

Vložení hodnoty X do fronty (předpokládáme, že je tam na ni místo):

```
if Prich < N then inc(Prich) else Prich:=1;  
F[Prich]:=X;
```

Odebrání hodnoty z fronty a vložení do proměnné X :

```
X:=F[Odch];  
if Odch < N then inc(Odch) else Odch:=1;
```

Dlouhá čísla

Chceme například počítat s kladnými celými čísly s max. 100 ciframi (podobně pro čísla se znaménkem, čísla desetinná apod.)

Reprezentace: číslo uložíme po cifrách do prvků pole

```
type Cislo = array [1..100] of byte;
```

```
var A, B, C: Cislo;
```

```
    PA, PB, PC: byte;    {počet cifer}
```

Nutno zvolit, kam dáme kterou cifru – je to jedno, ale potom důsledně dodržovat v celém programu!

Např.: A[1] – jednotky, A[2] – desítky, A[3] – stovky, ...,

A[PA] – cifra nejvyššího řádu

(příjemnější počítání)

nebo obráceně:

A[1] – cifra nejvyššího řádu, ..., A[PA] – jednotky

(snadnější načítání ze vstupu)

Operace:

- po cifrách – jako sčítání, odčítání, násobení či dělení víceciferných čísel na základní škole
- počítání modulo 10, přenosy do vyšších řádů

Modifikace:

číslo uložíme po dvojicích cifer do pole prvků typu byte, nebo po čtveřicích do prvků pole typu word, nebo dokonce po devíti cifrách do prvků pole typu longint
→ úspora paměti, rychlejší výpočet
(počítání modulo 100, nebo 10000, apod.)

Desetinné číslo:

stačí doplnit evidenci polohy desetinné čárky (buď speciální hodnota v prvku pole, nebo speciální proměnná s indexem nulového řádu, nebo dvě pole – celá a desetinná část čísla)

Příklad:

součet kladných celých čísel A a B dáme do C
(zvolená reprezentace čísla: A[1] = cifra jednotek)

```
if PA < PB then M:=PA  
           else M:=PB; {M - délka kratšího čísla}  
Prenos:=0;  
for I:=1 to M do  
begin  
    X := A[I] + B[I] + Prenos;  
    C[I] := X mod 10;  
    Prenos := X div 10  
end;
```

{dále podobně ošetřit „přečnívající“ část delšího
čísla a nakonec ještě případný nenulový přenos}

Vícerozměrné pole

```
type Obdelnik = array [1..3, 1..4] of integer;  
var M: Obdelnik;  
...  
M[2,3] := 3145;
```

- počet indexů není omezen (v praxi obvykle nejvýše tři)
- více indexů → pomalejší přístup k prvku (počítá se jeho adresa)
- zkratka za „pole polí“

```
type Obdelnik = array [1..3] of  
                  array [1..4] of integer;  
...  
M[2][3] := 3145;  
{toto je rovněž povolený korektní zápis,  
 ekvivalentní význam}
```

Příklad: součin čtvercových matic

```
const N = 10;  
type Matice = array [1..N, 1..N] of real;  
var A, B, C: Matice;  
...  
{počítáme součin A*B, výsledek uložit do C}  
for I:=1 to N do  
  for J:=1 to N do  
    begin  
      X:=0;  
      for K:=1 to N do X:=X + A[I,K] * B[K,J];  
      C[I,J] := X  
    end;
```

Inicializované pole

- inicializovaná proměnná může být typu pole
- hodnoty prvků se vypisují do závorek oddělené čárkami
- u vícerozměrných polí více úrovní závorek

```
const Mesice: array[1..12] of byte =  
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31) ;
```

```
const C: Obdelnik = ( (1, 2, 3, 4) ,  
                     (0, 1, 1, 0) ,  
                     (9, 8, 7, 6) ) ;
```


Dekompozice problému

– rozdělení problému na logicky ucelené, relativně samostatné části, každé části řešeného problému odpovídá samostatná část programu (realizace: procedura, funkce, modul, objekt).

Návrh **rozhraní** – způsob komunikace jednotlivých částí programu se svým okolím, s ostatními součástmi programu (např. v případě procedur a funkcí jaké mají parametry).

Programujeme vždy proti rozhraní,
bez znalosti (resp. bez využívání znalosti) toho, co je za ním.

Přístup k rozhraní **shora**: v zápisu programu voláme procedury a funkce, o nichž víme, jak se volají a co dělají, ale nevíme, jak to dělají

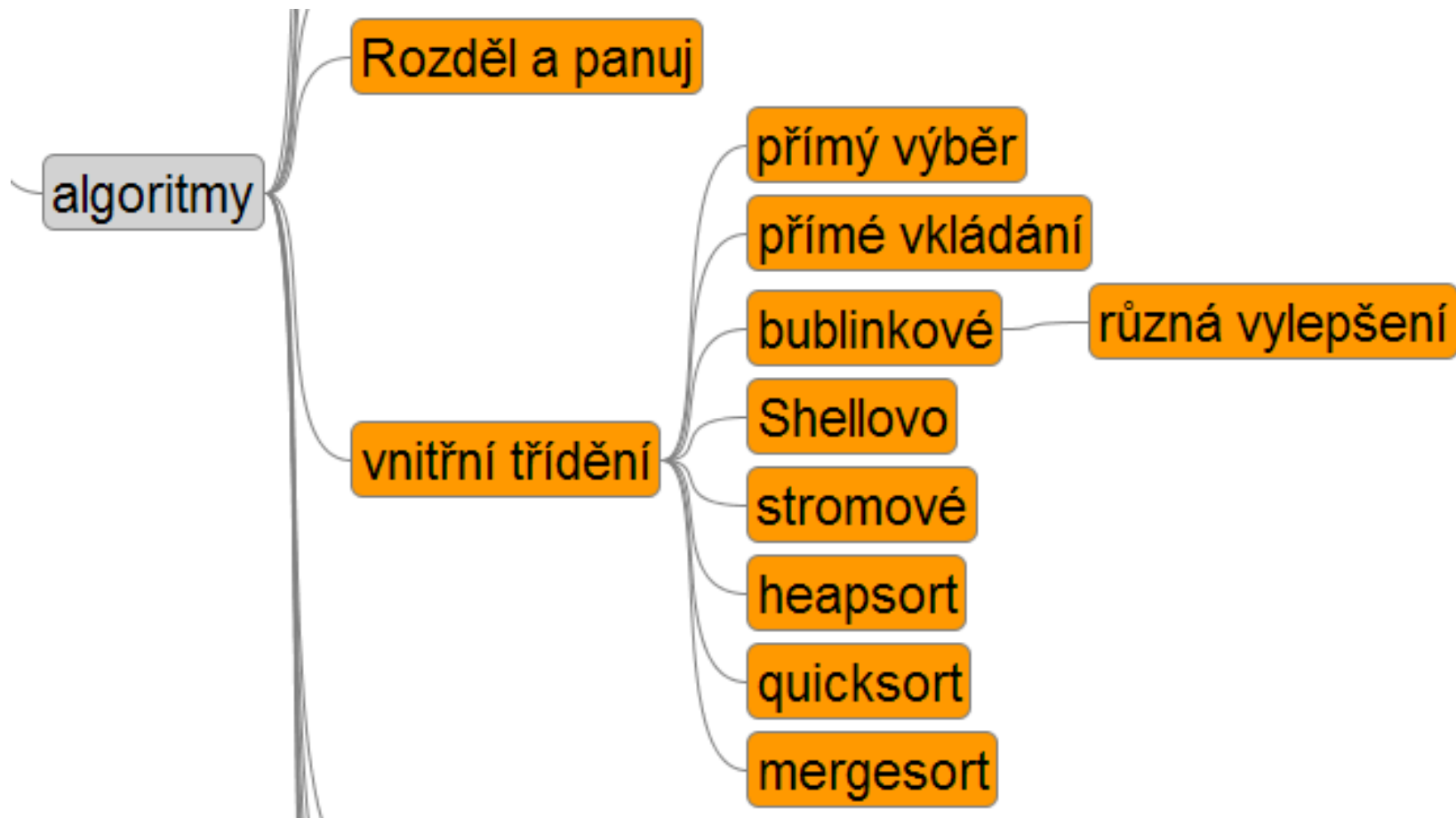
... a pokud to náhodou víme, tak tuto znalost nevyužíváme
(v budoucnu je možná změna implementace těchto procedur a funkcí)

Přístup k rozhraní **zdola**: píšeme procedury a funkce, o nichž víme, jak se budou volat a co mají dělat, ale nevíme, kdo a v jakém kontextu je bude používat

... a pokud to náhodou víme, tak tuto znalost nevyužíváme
(v budoucnu je možná bude chtít volat také někdo další)

Idea **dekompozice** a **programování proti rozhraní** se využívá opakovaně:

- *strukturované programování* – použití procedur a funkcí, rozhraním je hlavička procedury resp. funkce (název, parametry, typ)
- *modulární programování* – modul má definováno své rozhraní, uživatelé jsou přístupné pouze složky výslovně uvedené v rozhraní, ostatní nejsou dostupné a slouží pouze pro implementaci (pro vnitřní potřeby modulu)
- *objektové programování* – rozhraní třídy určuje ochranu přístupu ke složkám objektu resp. třídy
zvenčí jsou vidět pouze veřejné složky objektu (public), soukromé složky slouží pouze pro implementaci a nejsou zvenčí přístupné (private)
v některých jazycích existuje více úrovní ochrany (protected, internal,...)



Vnitřní třídění

Zadání: Uspořádejte pole délky N podle hodnot prvků

Měřítko efektivity:

- * počet porovnání
- * počet přesunů

Třídění přímým výběrem

Algoritmus:

1. Vyber prvek s nejmenší hodnotou (nejmenším klíčem)
2. Vyměň ho s prvkem na prvním místě pole
3. Totéž opakuj pro zbývající prvky pole

Třídění přímým vkládáním

Algoritmus:

Pole rozdělíme na dvě části - zdrojová a cílová (setříděná).

Na začátku cílová část je $a[1]$, zdrojová $a[2]..a[N]$

Krok - rozšíření cílové části $a[1]..a[i-1]$ o jeden prvek:

- zjistí, kam patří prvek $a[i]$
- zařad' ho tam

(tj. Případně odsuň prvky cílové (setříděné) části,
které budou za ním)

Bublinkové třídění

Algoritmus:

1. Jsou-li v poli dva sousední prvky ve špatném pořadí, vyměň je.
2. Opakuj krok 1.
 - N-krát
 - (N-1)-krát
 - , dokud je co vyměňovat.

Třídění přetřásáním

Algoritmus:

Stejný jako u bublinkového třídění, jen jiný způsob vyhledávání chybných dvojic.

Další zlepšení Bublínkového třídění:

hranice setříděných okrajů...

Stromové třídění

Idea: Získávat informace tak, aby se nevztahovaly všechny k jednomu prvku.

Algoritmus:

1. Sestrojit strom => nejmenší prvek
2. Vyloučit nejmenší prvek a obnovit strom.

Třídění haldou

Cíl: Nepotřebovat $2N-1$ paměťových jednotek

Definice HALDA:

- binární strom
- všechny hladiny jsou zcela zaplněny, až na poslední, která je zaplněná zleva
- pro každý uzel: hodnota \leq hodnota v synovském uzlu

Třídění haldou:

- ze všech prvků vytvoř haldu
- dokud není halda prázdná, odeber prvek z kořene

Uložení haldy v poli...

Vytvoření haldy I.

```
procedure VytvorHaldu( L,R: index );
var  i,j: index; x: Prvek;
begin
    i := L; j := 2*i; x := a[i]; { hodnota otce }
    while j <= R do { levý syn je ještě v haldě }
    begin
        if j+1 <= R then { pravý syn také... }
            if a[j].klic > a[j+1].klic then
                { ...a je menší! }
                j := j+1;
            if x.klic <= a[j].klic then break; { OK }
            a[i] := a[j]; i := j; j := 2*i { vyměním }
        end;
        a[i] := x      { až teď zapíšeme hodnotu otce }
    end;
```

Vytvoření haldy II.

```
L := n div 2+1;  
while L > 1 do  
begin  
    Dec( L );  
    VytvorHaldu( L,n )  
end
```

Třídění sléváním

Idea:

1. Pomocí N porovnání dovedeme spojit dvě setříděné posloupnosti po $N/2$ do setříděné posloupnosti (délky N)
2. 1-prvková posloupnost JE setříděná.

Algoritmus:

Posloupnost rozdělíme na dvě části, ty setřídíme*)
a potom slijeme.

*) stejným algoritmem

Quicksort

Idea: Posloupnost rozdělíme na části, které setřídíme týmž algoritmem. Jednoprvková posloupnost je setříděná.

Algoritmus na setřídění úseku pole:

- menší prvky dát do levé části
- větší prvky dát do pravé části
- setříd' levou část (není-li prázdná)
- setříd' pravou část (není-li prázdná)

Co znamená „menší/větší prvky“?

menší/větší než určená hodnota (PIVOT)

metoda „Rozděl a panuj“ („Divide & Impera“)

Quicksort - kód

```
procedure QuickSort( zac,kon: index );
var  M,pom: Hodnota;
begin
  x := zac; y := kon; M := ....(pivot)
  repeat
    while A[x] < M do Inc( x );
    while A[y] > M do Dec( y );
    if x<=y then
      begin
        pom := A[x]; A[x] := A[y]; A[y] := pom;
        Inc( x ); Dec( y )
      end
  until x > y;
  if zac < y then QuickSort( zac,y );
  if x < kon then QuickSort( x, kon )
end;
```

QuickSort III. - složitost

Volba pivota

Složitost v nejhorším případě

Volba pivota prakticky

Rozděl a panuj (Divide & Impera)

Řešení úlohy tím,
že ji rozdělíme na stejné úlohy menšího rozměru
a ty zase dělíme... až k úlohám, které mají snadné řešení.

Příklady

Quicksort

výpočet hodnoty výrazu rozdělením na podvýrazy

(rychlé) násobení matic

(rychlá) Fourierova transformace

Hledání k-tého nej...šího prvku z N

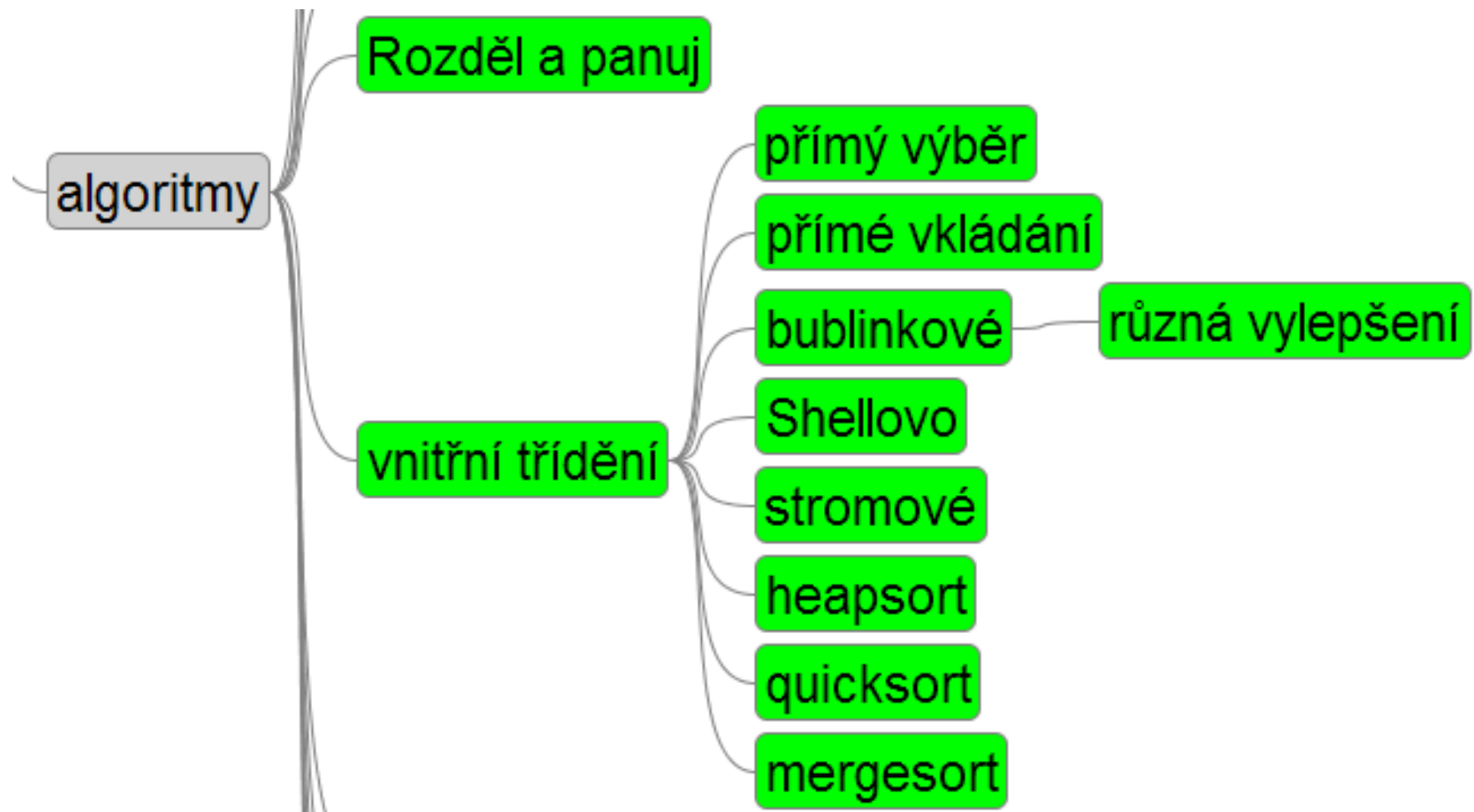
- postupným vybíráním minima/maxima
- pomocí haldy
- lineární algoritmus

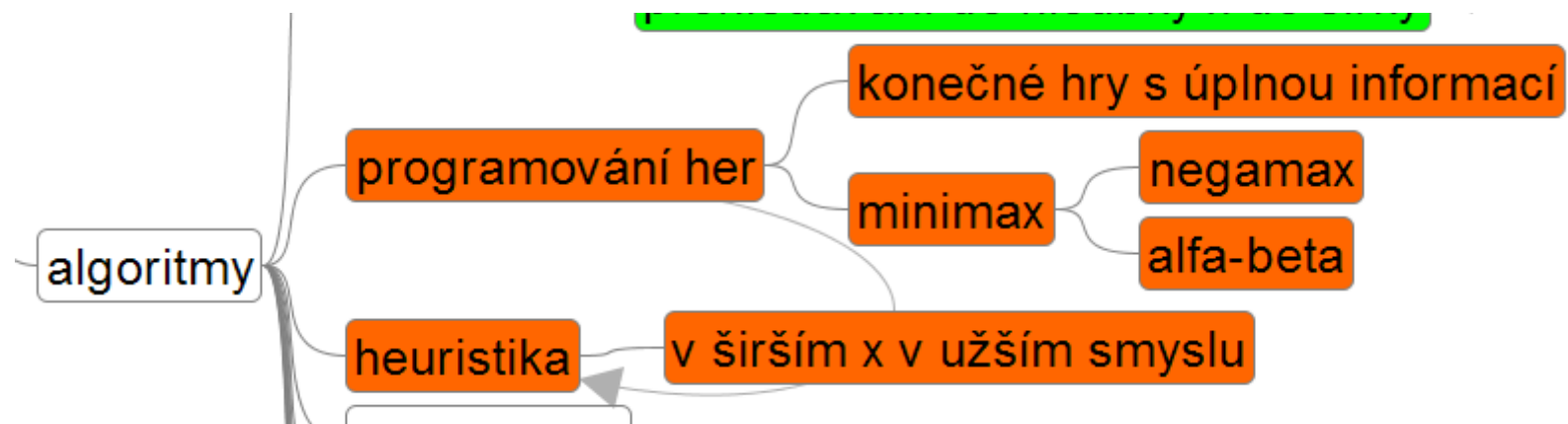
. Složitost úlohy

- . je nejmenší složitost algoritmu
ze všech algoritmů řešících tuto úlohu

Složitost úlohy vnitřního třídění
založeného na porovnávání dvou prvků
je $O(N \times \log N)$.

- . Přihrádkové třídění (bucket sort)
-s více průchody (radix sort).





Konečné hry s úplnou informací

hra dvou hráčů

konečná....skončí po konečném počtu kroků

s úplnou informací....oba hráči mají

všechny informace o stavu hry

Příklad: odebírání zápalek, piškvorky na konečné ploše, dáma, šachy, go...

Vyhrávající a prohrávající pozice

(kdyby neexistovala remíza)

- koncová pozice, ve které hráč na tahu prohrál,
je **prohrávající**
 - koncová pozice, ve které hráč na tahu vyhrál,
je **vyhrávající**
-

- pozice, ze které **lze** hráče dostat do prohrávající pozice,
je **vyhrávající**
- pozice, ze které **nelze** hráče dostat do prohrávající pozice,
je **prohrávající**

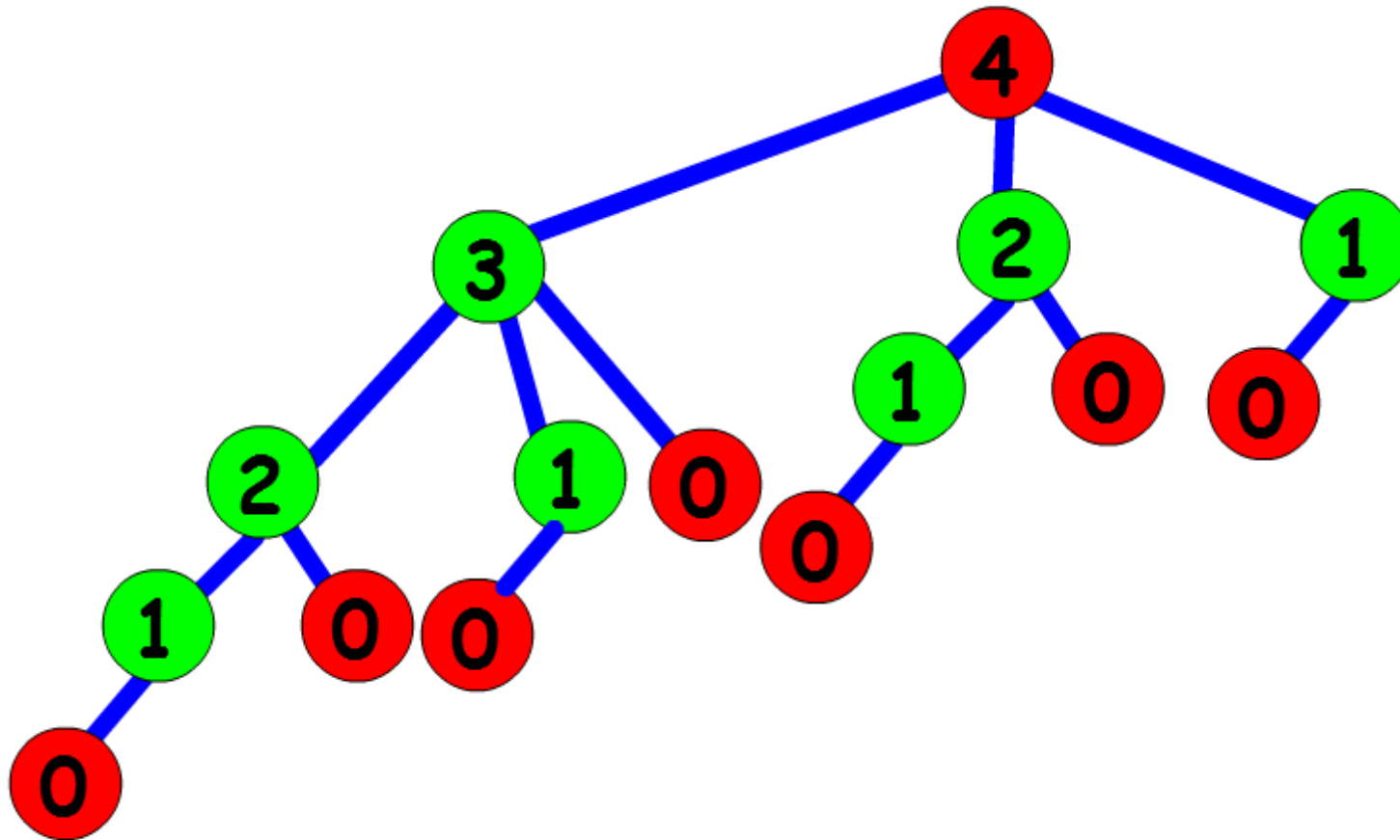
=> (kdyby neexistovala remíza)

každá dostupná pozice je buď vyhrávající nebo prohrávající

Strom hry

(nemusí být strom)

= Graf, vrcholy - stavy hry, hrany - možné tahy



Pokud existuje remíza, jsou kromě prohrávajících a vyhrávajících pozic ještě **neprohrávající** pozice.

Věta:

U konečné hry s úplnou informací alespoň pro jednoho hráče existuje neprohrávající strategie.

příklad: zápalky 1..3

příklad: zápalky z více hromádek

Hry s ohodnocením

= Každá cílová pozice je ohodnocena číslem
(namísto vyhrál-prohrál).

Jeden hráč se snaží dosáhnout maximálního výsledku,
druhý minimálního.

Hra s nulovým součtem

= Hra, ve které zisk jednoho hráče
je roven ztrátě druhého hráče.

Algoritmus MINIMAX

Strom hry se ohodnocuje od listů – koncových pozic,
(tam už víme, jak hra dopadla a ohodnocení známe).

Hodnota vrcholu-stavu se spočte jako

maximum/minimum hodnot synů,

podle toho,

zda je na tahu

maximalizující nebo minimalizující hráč.

příklad: řada čísel – součet dolů vs. součet nahoru

příklad: tabulka čísel – součet dolů vs. součet nahoru

Negamax

= Alternativa MINIMAXu,
namísto střídání MIN a MAX se počítá $-MAX(-...)$.
Snazší naprogramování.

$H := \max(\min(\max(\min(...))))$

vs.

$H := \max(-\max(-\max(-\max(-...))))$

alfa-beta prořezávání

Představme si, že

V uzlu U vybíráme maximum

a už umíme získat hodnotu ALFA

V jeho synu V vybíráme minimum...

a našli jsme hodnotu x menší nebo rovnu ALFA

=> nemá smysl zkoumat další tahy z uzlu V !

Protože:

- hodnota uzlu V bude určitě $\leq x$ (a $x \leq \text{ALFA}$)
...a dalším hledáním ji můžeme jedině zmenšit
- hodnota uzlu U bude určitě $\geq \text{ALFA}$
...takže ji hodnota uzlu V už nezvýší

Analogicky hodnota BETA a výběr minimum-maximum.

"Skutečné" hry

- omezená hloubka a statická ohodnocovací funkce
- hloubka ne všude stejná
- alfa-beta prořezávání
- okénko
- procházení do rostoucí hloubky
- příklad: BP70\EXAMPLES\CHESS

Heuristika

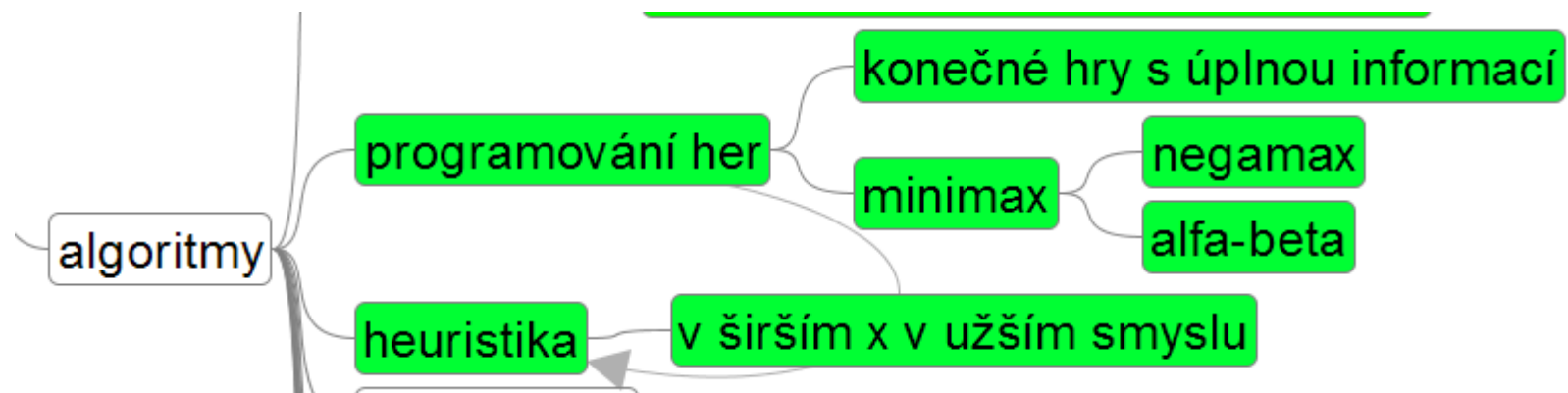
= rada, něco, co „obvykle dává dobré výsledky“

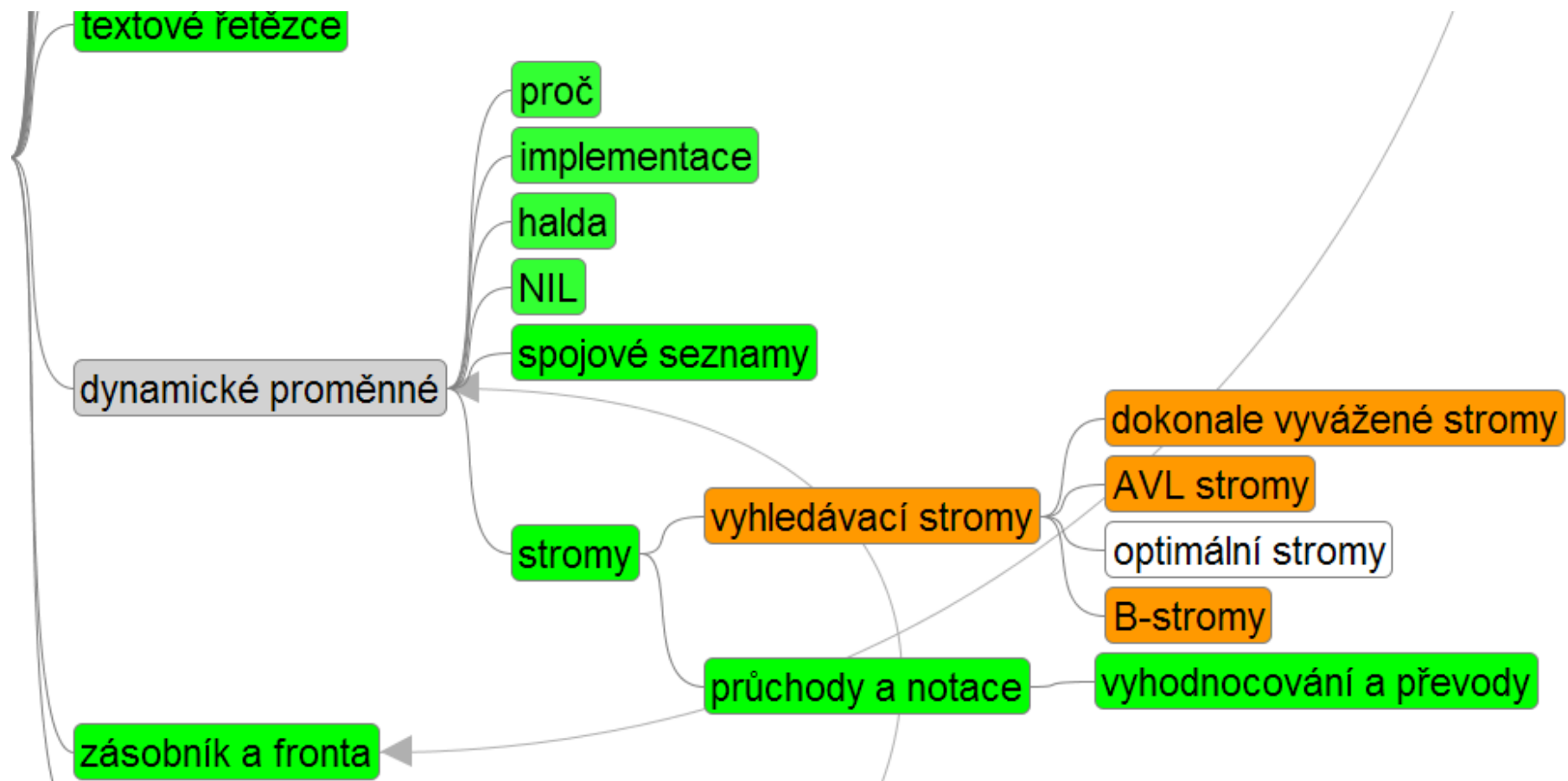
V užším smyslu (nic nezkazí)

pořadí prozkumávání tahů...

V širším smyslu (náhrada úplného řešení).

příklad s loupežníky...





Binární vyhledávací stromy

```
type PVrchol = ^TVrchol;  
    TVrchol = record  
        hodnota: typ;  
        L, P: PVrchol  
    end;
```

Pro každý vrchol platí:

- všechny hodnoty v levém podstromě jsou menší
- všechny hodnoty v pravém podstromě jsou větší

Operace s BVS

- najít prvek / zjistit, že tam není
- přidat prvek
- vymazat prvek

Vyvážené stromy

PROČ: OMEZIT VÝŠKU a tím SLOŽITOST

Dokonale vyvážené stromy

Pro každý vrchol platí:

Počet prvků levého a pravého podstromu
se liší nejvýše o 1.

Obtížné udržovat.

AVL-stromy (Adelson-Velskij, Landis)

Pro každý vrchol platí:

Výška levého a pravého podstromu
se liší nejvýše o 1.

AVL-stromy mají výšku $O(\log n)$.

Důkaz: Fibonacciho stromy.

Adelson-Velskij, G.; E. M. Landis:

An algorithm for the organization of information, 1962.

AVL-stromy - vyvažování

```
type PVrchol = ^TVrchol;  
    TVrchol = record  
        hodnota: typ;  
        b: -1..+1; { příznak vyvážení }  
        L, P: PVrchol  
    end;
```

Rotace

jednoduchá x dvojitá
levá x pravá

// existují (dva) různé názory, co je LEVÁ a co PRAVÁ

B-stromy (Bayer, McCreight)

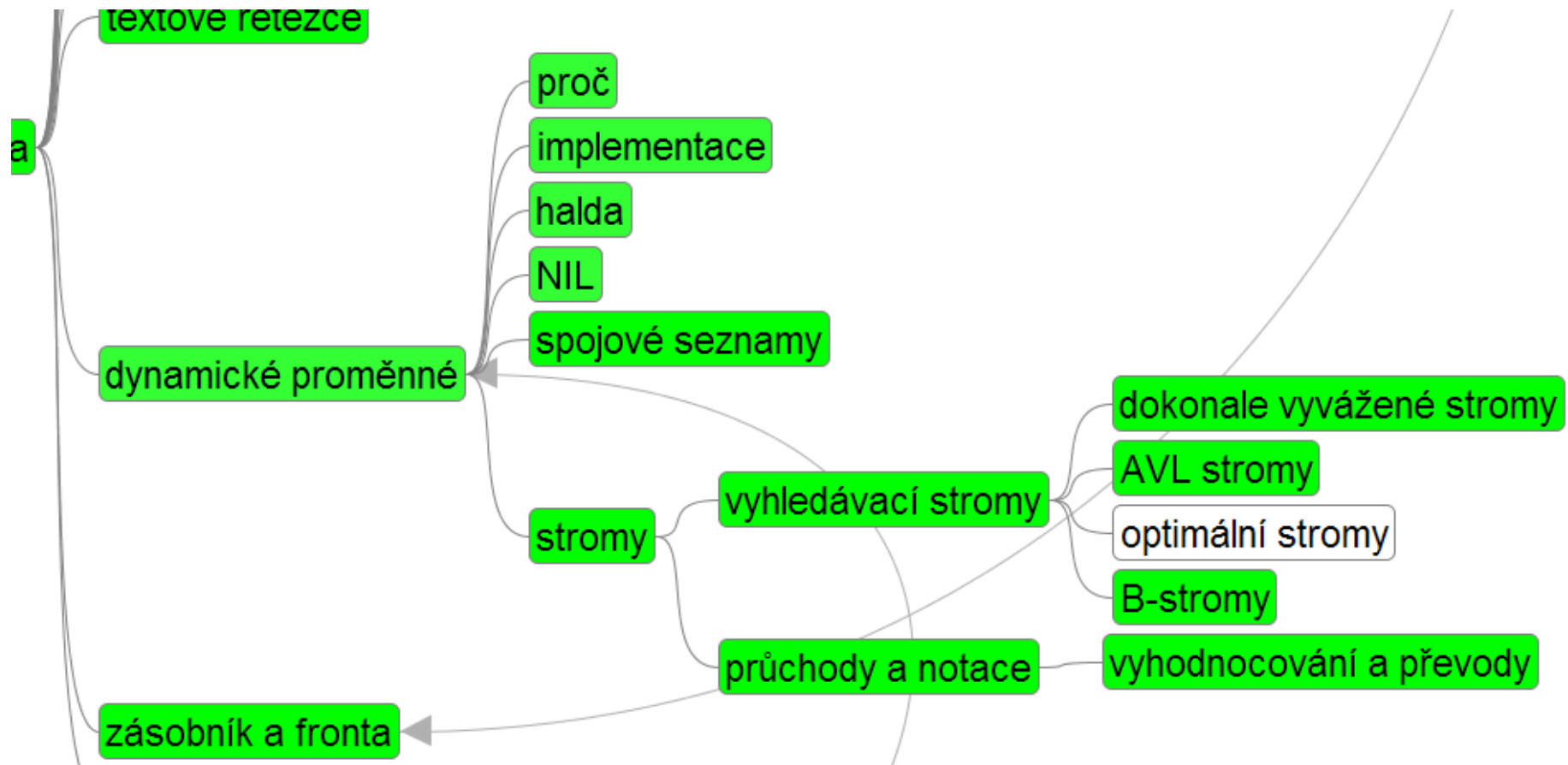
B-Strom stupně N:

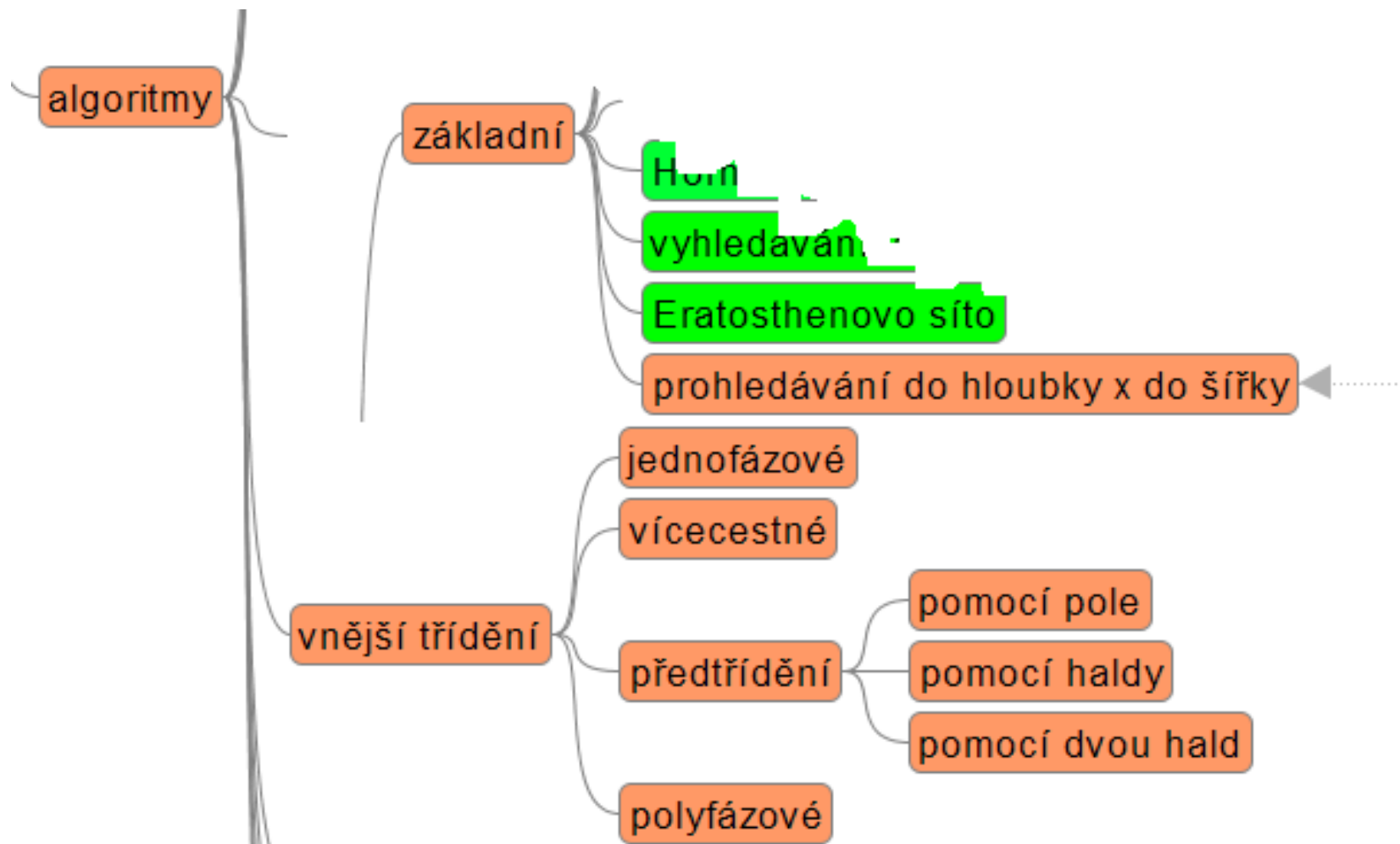
1. Každý vrchol kromě kořene má alespoň N hodnot
2. Každý vrchol má nejvýše $2N$ hodnot
3. Když má vrchol k hodnot, má $k+1$ synů
(kromě listů)
4. Všechny listy leží ve stejné hladině

// existují různé verze definic !

Bayer, R.; McCreight, E.: (1972)

Organization and Maintenance of Large Ordered Indexes





Vnější třídění

Zadání:

- setřídít data uložená v souborech,
nemáme přímý přístup k i-tému prvku.
- složitost měřená v počtu I/O operací

Základní operace:

slučování (merge)

dvoufázové dvoucestné přirozené slučování

POZOR (časté nedorozumění):

NEVYTVÁŘÍME SPOUSTU SOUBORŮ!!

Jenom dva...

Vnější třídění - vylepšení

- jednofázové
- vícecestné
- předtřídění
- obyčejně
- pomocí haldy
- pomocí dvou hald
- polyfázové

Zbytky...

set of ...

funkce jako parametry, proměnné pro funkce a jiné ošklivosti.

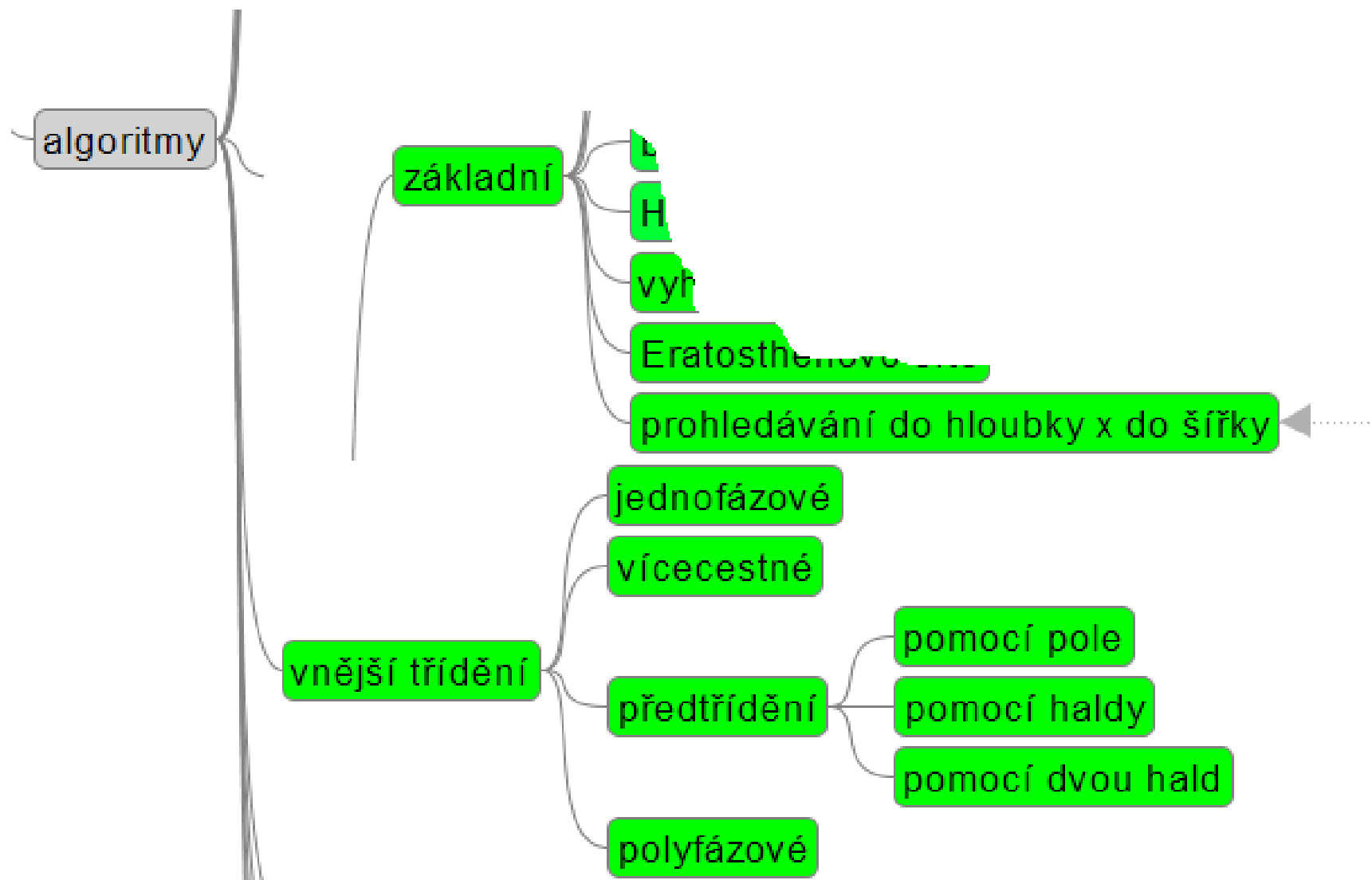
Příklady prohledávání

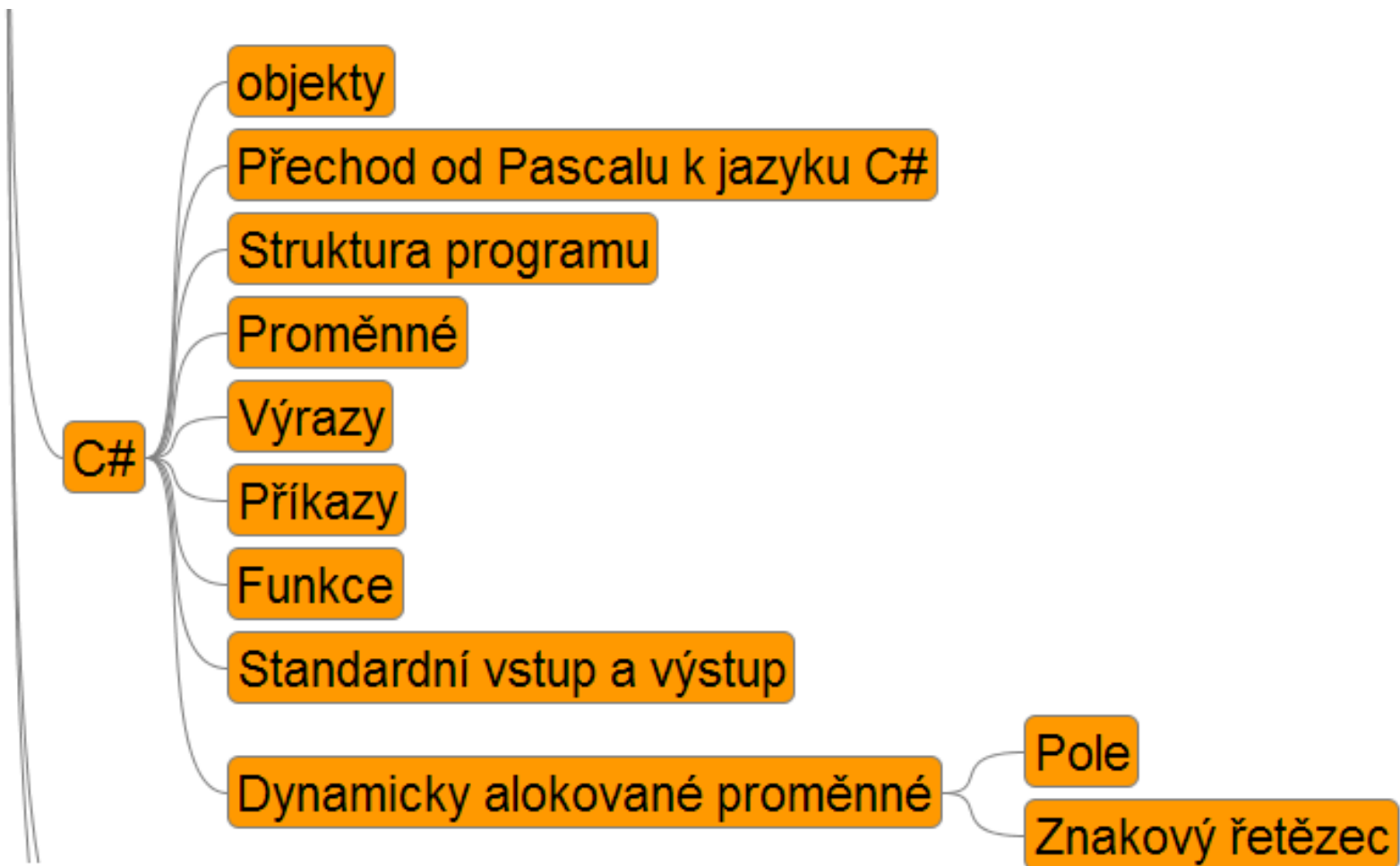
heuristiky (výběr stavu)

soubory

- append
- proměnná FileMode

Mark & Release





Objekty

Svět se skládá z objektů!

Objekt = data + funkce (metody)

konkrétní x abstraktní
hmatatelné x nehmatatelné
(letadlo) x (chyba v programu)

Objekty URČITÝM ZPŮSOBEM PODOBNÉ můžeme
považovat za instance jedné třídy (*pes*).

Objekty

Další pokus oddělit

CO x JAK

VENKU x UVNITŘ

INTERFACE x IMPLEMENTACE

Strukturované programování

blok, funkce

Modulární programování

modul, unit

Objektové programování

objekt a třída

Objekty v programu

Způsob jak izolovat část kódu
(příkaz-blok-procedura-modul-objekt).

Způsob jak uvažovat o problému
Objekt sdružuje DATA (datové složky, vlastnosti)
i KÓD (funkce+procedury=METODY)
= ČLENY (members)

OBJEKT = exemplář, instance TŘÍDY.

Zapouzdření
ukrývání vnitřku, díky tomu konsistentní stav

Příklad:

Napište program, který čte ze vstupu slova
a tiskne je na řádky dané délky.

Jazyk C#

- * C#

- Java 1995, bytecode
- C# 2002, .NET
 - Anders Hejlsberg

- * Microsoft Visual Studio, „Community“ verze zdarma
- * Projekt MONO (macOS, Linux, Windows)

Přechod od Pascalu k jazyku C#

Používání mezer a řádkování

- volné, stejně jako v Pascalu
- doporučená a prostředím podporovaná indentace

Identifikátory

- case-sensitivní
- možnost používat diakritiku
- klíčová slova malými písmeny
- konvence (zvyklosti):
 - proměnné malými písmeny, konstanty velkými písmeny
 - jména prostorů*), tříd, metod a vlastností, veřejné členy
 - > „PascalskáNotace“
 - např. `Math`, `DivideByZeroException`, `Main`, `WriteLine`
 - soukromé metody začínají malým písmenem
 - > „velbloudíNotace“

Struktura programu

Celý program se skládá ze tříd,
vše se deklaruje a používá uvnitř tříd
(proměnné, konstanty, funkce, ...).

Položky deklarované ve třídě:

- datové složky třídy = **členské proměnné**
- metody = **členské funkce**

Prozatím: celý program je tvořen jedinou statickou*)
metodou (její obsah tedy odpovídá celému programu)

Někdy přístě: jak jinak to může vypadat se třídami

Proměnné

- zápis deklarace

- **syntaxe:** `int alfa;`

- umístění deklarace:

BUĎ členská proměnná třídy (tzn. datová složka objektu)

NEBO lokální kdekoliv ve funkci, ale nesmí zakrýt jinou stejnojmennou deklaraci uvedenou v téže funkci

(pozor na kolize!)

- lokální platnost deklarace v bloku, kde je uvedena


- možnost inicializace v rámci deklarace: `int alfa = 15;`

- v programu nelze použít nedefinovanou hodnotu proměnné
(kontrola při překladu) 

- hodnotové a referenční typy

- všechno*) je objekt (instance nějaké třídy)

Konstanty

- syntaxe jako inicializované proměnné, specifikátor `const`:
`const int ALFA = 15;`
- číselné konstanty podobné jako v Pascalu (různé typy)
- konstanty typu `char` v apostrofech: `'a'`,
typu `string` v uvozovkách: `"aaa"` 

Typy

Hodnotové

celé číslo

`int` `System.Int32` 32 bitů

další typy: `byte`, `sbyte`, `short`, `ushort`, `uint`, `long`, `ulong`

desetinné číslo

`double` `System.Double` 64 bitů

další typy: `float`, `decimal`

logická hodnota

`bool`

znak

`char` `System.Char` 16 bitů Unicode

výčtový typ

`enum`

struktura

`struct`

Referenční

pole

[] System.Array

znakový řetězec

string System.String

třída


class

(standardní třídy, např. ArrayList,
StringBuilder, List<>)

Hlavní rozdíl:

Dosazuje se hodnota nebo reference.

Aritmetické výrazy

- obvyklé symboly operací i priority stejné jako v Pascalu
+ - * /
- **POZOR:** symbol / představuje reálné i celočíselné dělení (zvolí se podle typu argumentů) = zdroj chyb! 
- znak % pro modulo (zbytek po celočíselném dělení)
- klíčová slova checked, unchecked - určení, zda se má kontrolovat aritmetické přetečení v celočíselné aritmetice
- použití jako checked(výraz) nebo checked{blok}
- standardní matematické funkce
= statické*) metody třídy Math

Středník

- ukončuje každý příkaz
(musí být i za posledním příkazem bloku!)
- nesmí být za blokem ani za hlavičkou funkce
- odděluje sekce v hlavičce for-cyklu

Čárka

- odděluje deklarace více proměnných téhož typu
- odděluje parametry v deklaraci funkce i při volání funkce
- odděluje indexy u vícerozměrného pole

Komentáře

- jednořádkové `// xxx` do konce řádku
- víceřádkové `/* xxx */`
- dokumentační `///`

Blok (složený příkaz)

- závorky `{ }` místo pascalského `begin - end`

Dosazovací příkaz

- syntaxe: `proměnná = výraz` např. `i = 2*i + 10;`

Příkaz modifikace hodnoty

```
i++; ++i;  
i--; --i;
```

```
i += 10;  
i -= 10;  
i *= 10;  
i /= 10;  
i %= 10;
```

Podmíněný příkaz

- podmínka = výraz typu bool v závorkách

```
if (a == 5)    b = 17;
```

```
if (a == 5)    b = 17;
```

```
    else      b = 18;
```

- relační operátory: == != < > <= >=

- logické spojky

&& and (zkrácené vyhodnocování)

|| or (zkrácené vyhodnocování)

& and (úplné vyhodnocování)

| or (úplné vyhodnocování)

! not

^ xor

For-cyklus

- **syntaxe:**

for (inicializace; podmínka pokračování; příkaz iterace)
příkaz těla

```
for (int i=0; i<N; i++)    a[i] = 3*i+1;
```

- některá sekce může být prázdná (třeba i všechny)

(pokud víc příkazů, oddělují se čárkou)

Cykly while a do-while

- cyklus **while** stejný jako while-cyklus v Pascalu (podmínka je opět celá v závorce a nepíše se „do“)

```
while (podmínka) příkaz;
```

- cyklus **do-while** má podmínku na konci jako cyklus repeat-until v Pascalu, ale význam podmínky je proti Pascalu obrácený,
tzn. dokud podmínka platí, cyklus se provádí

```
do příkaz while (podmínka) ;
```

- více příkazů v těle cyklu musí být uzavřeno v bloku { }

Ukončení cyklu

- příkazy

`break;`

`continue;`

- stejný význam jako v Pascalu

Příkaz switch

- analogie pascalského příkazu case
- varianta se může rozhodovat podle výrazu celočíselného, podle znaku nebo také stringu
- sekce **case**, za každým case jediná konstanta, ale pro více case může být společný blok příkazů
- poslední sekce může být default:
- je povinnost ukončit každou sekci **case** (i sekci default, neboť ta nemusí být uvedena poslední) příkazem

`break, příp. return nebo goto`

```
int j, i = ...;  
switch (i)  
{  
    case 1:  
        i++; break;  
    case 2:  
    case 3:  
        i--; break;  
    default:  
        i=20; j=7; break;  
}
```

(pozor: v C, C++, Java, PHP... se může propadat mezi
sekcemi = zdroj chyb, v C# opravený)





Funkce

- pascalské procedury a funkce
 - > metody (členské funkce) nějaké třídy
- procedura
 - > funkce typu void
- v deklaraci i při volání vždy píšeme (),
i když nemá žádné parametry
- ve funkci nelze lokálně definovat*) jinou funkci,
strukturu nebo třídu,
Ize tam ale deklarovat lokální proměnné
(ve třídě lze deklarovat jinou třídu
ta může mít své metody)



*) V C#7 už lze...

Funkce...

- mohou vracet i složitější*) typy
- `return <hodnota>;`
 - definování návratové hodnoty a ukončení funkce
- v případě funkcí typu `void` pouze `return;`
- předávání parametrů:
 - standardně hodnotou
 - odkazem - specifikátor `ref` v hlavičce i při volání *) 
 - výstupní parametr
 - specifikátor `out` v hlavičce i při volání 
 - (`out` je také odkazem, nemá ale vstupní hodnotu)

*) V C#7 už lze vracet "tuple"

Výchozí metoda Main()

- plní funkci hlavního programu
(určuje začátek a konec výpočtu)
- je to statická*) metoda nějaké třídy
(nic „mimo třídy“ neexistuje),
často se pro ni vytváří samostatná třída
- obvykle jediná v aplikaci
→ je tak jednoznačně*) určeno, kde má začít výpočet
- *)může jich být i více, pak se ale při kompilaci musí
přepínačem specifikovat, ze které třídy se má použít
Main() při spuštění programu
- **syntaxe:** `static void Main(string[] args)`

Standardní vstup a výstup

`Console.Read();`

vrací `int` = jeden znak ze vstupu (jeho kód)

`Console.ReadLine();`

vrací `string` = jeden řádek ze vstupu

`Console.Write(výraz);`

vypíše hodnotu zadaného výrazu

`Console.WriteLine(výraz);`

vypíše hodnotu zadaného výrazu a odřádkuje

Formátovaný výstup

`Console.WriteLine(string);`

do stringu se dosadí hodnoty výrazů po řadě na místa vyznačená pomocí {0}, {1}, {2}, atd., případně i s požadovaným formátováním {0:N}

```
Console.WriteLine(  
    "x0={0} x1={1} x2={2} ...a to je vše",  
    x0, x1, x2  
);
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            XXXXXXXXXXXXXXXXXXXXXXXX
        }
    }
}
```

Příklad: Eukleidův algoritmus

```
static void Main(string[] args)
{
    Console.WriteLine(
        "Zadej dvě kladná celá čísla:");
    int a = int.Parse(Console.ReadLine());
    int b = int.Parse(Console.ReadLine());
    while (a != b)
    {
        if (a > b) a -= b;
        else b -= a;
    }
    Console.WriteLine(
        "Nejv. spol. dělitel: {0}", a);
    Console.ReadLine();
}
```

Příklady:

Prvočíselný rozklad

Hornerovo schéma - vstup čísla po znacích


```
static void Main(string[] args)
{
    int v;
    int c;
    c = Console.Read();
    // preskocit ne-cislice:
    while ((c < '0') || (c > '9'))
    {
        c = Console.Read();
    }
    // nacitat cislice:
    v = 0;
    while ((c >= '0') && (c <= '9'))
    {
        v = 10 * v + (c - '0');
        c = Console.Read();
    }
    Console.WriteLine(v);
    Console.ReadLine();
}
```

Dynamicky alokované proměnné


- vytvářejí se pomocí zápisu
new + konstruktor*) vytvářeného objektu
- new je funkce, vrací vytvořenou instanci
(ve skutečnosti ukazatel na ni)
- v odkazech se nepíšou ^
- string, pole, třídy - referenční typy
- konstanta null (jako NIL v Pascalu)
- automatická správa paměti
nedostupné objekty jsou automaticky uvolněny z paměti
(ne nutně úplně okamžitě, až to bude potřeba)

Příklad

```
prvni = null;  
    // to je korektní zrušení celého spojového seznamu
```

```
class Uzel  
{  
    public int info;  
    public Uzel dalsi;  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        Uzel prvni = new Uzel();  
        prvni.info = 123;  
        prvni.dalsi = null;  
        // ...  
    }  
}
```

Pole

- deklarace: `int[] aaa;`
- referenční typ, je nutné vytvořit pomocí `new`:
`int[] aaa = new int[10];`
- každé pole je instancí třídy odvozené*) z abstraktní statické třídy `System.Array`
- indexování vždy od 0
- možnost inicializace:
`int[] aaa = new int[3] { 2, 6, 8 };`
`int[] aaa = { 2, 6, 8 };`
- počet prvků: `aaa.Length`
- vždy se provádějí kontroly přetečení mezí 
při indexování `aaa[i]`

POZOR !

```
static void Main(string[] args)
{
    int[] aaa = { 2, 6, 8 };
    int[] bbb;
    bbb = aaa;
    aaa[0] = 27;

    Console.WriteLine(bbb[0]);
}
```

dosazuje se ukazatel !!

Pole...

- připravené metody, např. `CopyTo`, `Sort`, `Reverse`, `BinarySearch`, `Array.Reverse(aaa)` ;
- vícerozměrné pole
obdélníkové `[,]` a nepravidelné `[][]`

Nepravidelné dvourozměrné pole je ve skutečnosti pole polí (tzn. pole ukazatelů na řádky, což jsou pole jednorozměrná),

- každý řádek je třeba zvlášť vytvořit pomocí `new`
- řádky mohou mít různou délku

```
int[][] aaa = new int[3][];  
aaa[0] = new int[4];  
aaa[1] = new int[6];  
aaa[2] = new int[2];
```

Příklad: Třídění čísel v poli - přímý výběr

```
static void Main(string[] args)
{
    Console.Write("Počet čísel: ");
    int pocet = int.Parse(Console.ReadLine());
    int[] a;
    a = new int[pocet];
    int i = 0;
    while (i < a.Length)
        a[i++] = int.Parse(Console.ReadLine());
    i = 0;

    while (i < a.Length)
    {
        int k = i;
        int j = i+1;
        while (j < a.Length)
        {
```

```

        if (a[j] < a[k]) k = j;
        j++;
    }
    if (k != i)
    {
        int x = a[i];
        a[i] = a[k];
        a[k] = x;
    }
    i++;
}

i = 0;
while (i < a.Length)
    Console.Write(" {0}", a[i++]);
Console.WriteLine();
}

```


Znakový řetězec

- deklarace: `string sss;`
- `typ string` - referenční typ, alias pro třídu `System.String`
- vytvoření instance: `string sss = "abcdefg";`
- nulou-ukončené řetězce, nemají omezenou délku
- indexování znaků od 0
- délka = `sss.Length`
- obsah nelze měnit (na to je třída `StringBuilder`)
- všechny objekty mají konverzní metodu `ToString()`,
pro struktury a objekty je vhodné předdefinovat ji
(jinak se vypisuje jenom jejich jméno)

Struktura - struct

- „zjednodušená třída“
- má podobný význam a použití jako pascalský záznam (record)
- navíc může mít metody (jako třída)
- může mít i konstruktor
(vlastní konstruktor musí inicializovat všechny datové složky, jinak má i implicitní bezparametrický konstruktor)
- je to **hodnotový** typ
(na rozdíl od instance třídy se nemusí alokovat)
- některá omezení oproti třídám (např. nemůže dědit)

```
struct Bod
{
    public int x, y;
    public Bod(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

