NÆvrh algoritmu°l

IB002

Doplňující text k přednÆšce

Literatura

- *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein:* Introduction to Algorithms, 2nd ed., The MIT Press & McGraw-Hill, 2002
- *Thomas W. Parsons:* Introduction to Algorithms in Pascal, John Wiley & sons, 1995
- *Steven S. Skiena:* The Algorithm Design Manual, Springer, 1998
- `http://www.fi.muni.cz/~libor/vyuka/IB002/`
- zÆpisky z přednÆšek...

Program, algoritmus, funkce

Program je formální popis chování jakéhokoli systému (aplikační program, reaktivní systém, operační systém...). Skládá se z popisu algoritmů.

Algoritmus lze chápat jako parciální funkci z množiny všech možných vstupů do množiny všech možných výstupů.

$$\text{funkce} : \text{Vstupy} \dashrightarrow \text{Výstupy}$$

Pozor, není to definice: sice každému algoritmu odpovídá parciální funkce ze vstupů do výstupů, ale naopak ne každou takovou funkci lze vyjádřit algoritmem.

Algoritmus je parciÆlní funkce z množiny vstupů do množinyvýstupů, k níž existuje konečný popis v určitØm formalismu.

Tímto formalismem je typicky programovací jazyk, ale existují i další matematickØ formalismy: Turingův stroj, RAM, lambda kalkul, kombinÆtorový kalkul, vyčíslitelnØ funkce, Markovův algoritmus . . .

Slovo *algoritmus* je odvozeno ze jmØna starověkØho perskØho matematika a astronoma al-Chwarezmího (Abū ‘Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī).

Funkce z množiny vstupů do množiny výstupů, kterØ lze vyjÆdřit algoritmem, se nazývají *rekursivně spočetnØ*.

Reaktivní systØm je program, jehož provÆdění normÆlně nemusí končit (např. operační systØmy apod.), v obecnějším smyslu jsou to interaktivní programy.

Program, který je implementovaným algoritmem, při výpočtu přečte vstup, zpracuje ho (vstupní data transformuje na výstupní), vypíše výstup a skončí.

Většina současných programů je interaktivních, ale součÆstí všech jsou algoritmy.

Korektnost algoritmu

In	množina vstupních dat
Out	množina výstupních dat
A : *In* $\dashrightarrow$ *Out*	algoritmus (na vstupu z *In* vypisující výstup z *Out*)
φ : *In* $\rightarrow$ *Bool*	vstupní podmínka
ψ : *In* $\times$ *Out* $\rightarrow$ *Bool*	výstupní podmínka

Vstupní podmínka φ určuje, zda daný vstupní œdaj je pro algoritmus *přípustný*, vymezuje tedy jistou podmnožinu množiny vstupních dat *In*.

Výstupní podmínka ψ říkÆ, zda daný výstupní œdaj jěi není výsledkem výpočtu algoritmu na danØm vstupním œdaji.

Vstupní podmínka je unÆrní predikÆt na množině vstupních dat, výstupní podmínka je binÆrní predikÆt na množinÆch vstupních a výstupních dat.
Výstupní podmínka ψ říkÆ, zda daný výstupní œdaj jëi není (pro nedeterministickØ algoritmy zda může či nemůže být) výsledkem výpočtu algoritmu na danØm vstupním œdaji.

Příklad: Algoritmus A přečte tři čísla a vypíše je v neklesajícím pořadí; dejme tomu, že funguje korektně jen pro kladnÆčísla.

$\mathit{In} = \mathit{Out} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$,

$\varphi_A(x, y, z) \equiv x \geq 0 \wedge y \geq 0 \wedge z \geq 0$

$\psi_A(x, y, z, u, v, w) \equiv (u, v, w) \in \mathit{Permut}(x, y, z) \wedge u \leq v \leq w$

PoznÆmka: Připustíme i algoritmy, kterØ nenačítají žÆdný vstup. Pro ně položíme $\mathit{In} = \{\emptyset\}$.

Def: Řekneme, že A je *konvergentní* vzhledem k φ, když množina $\{x \in In \mid \varphi(x)\}$ je podmnožinou definičního oboru funkce A, tj. když pro každou vstupní hodnotu x, pro niž platí $\varphi(x)$, je výsledek výpočtu podle algoritmu A definovÆn (výpočet se zastaví).

Def: Řekneme, že A je *parciÆlně korektní* vzhledem k φ a ψ, když pro každou vstupní hodnotu x z definičního oboru funkce A splňující vstupní podmínku φ (tj. pro každØ x, pro něž je $A(x)$ definovÆno a $\varphi(x)$) platí $\psi(x, A(x))$.

Def: Řekneme, že A je *totÆlně korektní* vzhledem k φ a ψ, když je konvergentní vzhledem k φ a parciÆlně korektní vzhledem k φ a ψ.

Důsledek: Algoritmus A je totÆlně korektní vzhledem k φ a ψ, prÆvě když pro každou vstupní hodnotu x splňující vstupní podmínku výpočet podle algorimu A skončí a jeho výsledek splňuje i výstupní podmínku.

Vstupní a výstupní podmínce se takØ říkÆ *specifikace algoritmu*. O totÆlně korektním algoritmu pak říkÆme, že *vyhovuje specifikaci*.

Výpočetní paradigmata

Paradigma (vzor, přístup, pojetí) udÆvÆ způsob, jakým je algoritmus vyjÆden.

Výpočetní paradigmata:

– imperativní

– funkcionÆlní

– logickØ

– procedurÆlní

– objektovØ

– sekvenční

– paralelní

⋮

Imperativní paradigma

NeformÆlní vymezení:

Výpočet je posloupností tzv. *výpočetních kroků*. Výpočetní krok je realizací instrukce (jednoduchØho příkazu) programu. Přenos informace mezi jednotlivými kroky výpočtu zprostředkovÆvÆ*stav*.

Jinak řečeno, příkazy programu jsou *stavovØ transformÆtory* — (parciÆlní) funkce z množiny stavů do množiny stavů.

Pro formÆlní definici je nutno zavØst abstraktní počítač. V sekvenčním imperativním paradigmatu jím nejčastěji bývÆ Turingův stroj nebo RAM (Random-Access Machine).

Může jím být takØ „vyšší“ imperativní jazyk, který kromě elementÆrních stavových transformÆtorů (typicky tzv. *přiřazovacího příkazu*) obsahuje složenØ stavovØ transformÆtory realizující sekvenci (begin-end), binÆrní větvení výpočtu (if-then-else), nÆsobnØ větvení (case), opakovÆní (while-do, for-do), apod.

Výpočetním krokem je pak přechod mezi dvěma konfiguracemi Turingova stroje, resp. provedení jednØ instrukce RAMu, resp. provedení jednoho *elementÆrního* příkazu.

Abstraktní počítače pro imperativní programy

- Turingův stroj
- RAM
- (abstraktní) programovací jazyk

⋮

Turingův stroj

- Konečná neprázdná množina Σ symbolů — tzv. *abeceda*
- Nekonečná *páska* — spočetná posloupnost políček. Každé políčko pásky obsahuje jeden symbol abecedy, přičemž skoro všechna políčka obsahují zvláštní symbol $\sqcup$.
- *Čtecí hlava*, jejíž pozice na pásce je dána přirozeným číslem.
- Konečná množina S vnitřních stavů s vyznačeným *počátečním stavem* σ_0 a *koncovými stavy* $F \subseteq S$.
- *Přechodová funkce* $\delta : S \times \Sigma \to S \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$

RAM

- SpočetnÆ posloupnost $\mathcal{M}$ registrů (paměťových míst) — tzv. *paměť*. Každý registr obsahuje celØčíslo, vždy však skoro všechny obsahují nulu.
- *Stav* $\sigma \in S = \mathbb{N} \times \mathbb{Z}^*$
- KonečnÆ množina instrukcí $\mathcal{I}$. KaždÆ instrukce $\imath \in \mathcal{I}$ mÆ danou sØmantiku — funkci $\iota : S \dashrightarrow S$.
- KonečnÆ posloupnost $P \in \mathcal{I}^*$ instrukcí, tzv. *program*

Příklad: Výpočet faktoriÁlu (funkcionÁlní verze – Pascal)

FaktoriÁl nezÁpornÉho celÉho čísla n je číslo $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.

SpeciÁlně $0! = 1$ (součin nulovÉho počtu činitelů).

```
function fact (n:Integer): Integer;
begin
 if n==0 then fact:= 1              {return 1}
         else fact:= n*fact(n-1)    {return n*fact(n-1)}
end
```

Věta: Funkce `fact` konverguje vzhledem ke vstupní podmínce $\varphi(n) \equiv n \in \mathbb{N}$.

Věta: Funkce `fact` je parciÁlně korektní vzhledem k φ a k výstupní podmínce $\psi(n,k) \equiv k = n!$.

Důsledek: `fact` je totÁlně korektní vzhledem k φ, ψ.

Věta: Funkce `fact` konverguje vzhledem ke vstupní podmínce $\varphi(n) \equiv n \in \mathbb{N}$.

Důkaz indukcí podle n. Pro $n = 0$ se výpočet zastaví; zastaví-li se výpočet pro n, zastaví se i pro $n + 1$.

Věta: Funkce `fact` je parciálně korektní vzhledem k φ a k výstupní podmínce $\psi(n, k) \equiv k = n!$.

Důkaz opět indukcí podle n. $\texttt{fact}(0) = 1 = 0!$.
Nechť $n > 0$ a tvrzení platí pro $n - 1$. Pak
$\texttt{fact}(n) = n \cdot \texttt{fact}(n - 1) = n \cdot (n - 1)! = n!$, tedy tvrzení platí i pro n.

Výpočet faktoriÆlu (imperativní verze – Pascal)

```
function factorial (n:Integer): Integer;
var i, k: Integer;
begin
  i := 0;
  k := 1;
  while  i < n  do
   begin i := i+1;
         k := k*i
   end;
  factorial:= k    {return k}
end
```

Věta: Funkce konverguje vzhledem ke vstupní podmínce $\varphi(n) \equiv n \in \mathbb{N}$.

Funkce je parciÆlně korektní vzhledem k φ a k výstupní podmínce $\psi(n, k) \equiv k = n!$.

Věta: Funkce `factorial` konverguje vzhledem ke vstupní podmínce $\varphi(n) \equiv n \in \mathbb{N}$.

Důkaz spočívÆ v nalezenčíselnØ hodnoty (zÆvislØ na obsahu programových proměnných), kterÆ v žÆdnØm okamžiku výpočtu nemůže být zÆpornÆ a přitom při každØm průchodu tělem cyklu klesne nejmØně o jedničku. V tomto příkladě je touto hodnotou číslo $n - i$ a důkaz nezÆpornosti i klesÆní je snadný.

Věta: Funkce `factorial` je parciÆlně korektní vzhledem k φ a k výstupní podmínce $\psi(n, k) \equiv k = n!$.

Důkaz spočívÆ v nalezení mezilehlých podmínek, zejmØna invariantu cyklu while. Invariant cyklu v místě testovÆní podmínky je $k = i! \ \wedge \ 0 \leq i \leq n$.

Invariant cyklu

je mezilehlÆ podmínka, kterÆ je splněna v danØm bodě výpočtu podle algoritmu Invariant cykluv každØm průchodu cyklem.

V našem příkladě bude invariantem cyklu while podmínka $k = i! \wedge 0 \leq i \leq n$ a bude se vztahovat k místu testovÆní podmínky cyklu. Důkaz parciÆlní korektnosti funkce `factorial` se rozloží do kroků:

1. Platí-li na začÆtku $\varphi(n)$, pak po provedení œvodních přiřazení bude splněn invariant cyklu.
2. Je-li v nějakØm okamžiku splněn invariant cyklu a provede se tělo cyklu, pak po jeho provedení bude opět splněn invariant.
3. Je-li splněn invariant cyklu a není splněna podmínka cyklu, pak je splněna mezilehlÆ podmínka za cyklem, tedy $\psi(n, k)$.

Pro místa v algoritmu ležící mezi příkazy stanovíme tzv. *mezilehlØ podmínky*. Podmínka na začÆtku bude $\varphi(n)$, podmínka na konci bude $\psi(n,k)$. Z ostatních bodů v algoritmu pro důkaz parciÆlní korektnosti obvykle stačí vybrat jen význačnÆ místa, v nichž se stav výpočtu významně mění. Takovými místy jsou body uvnitř cyklů, například na jejich začÆtku. MezilehlÆ podmínka pro místo uvnitř cyklu se nazývÆ *invariant cyklu*.

Příklad: Algoritmus umocňovÆnčísla na přirozený exponent (funkcionÆlní verze – Pascal)

```
function pow (z:Real; n:Integer): Real;
{ Předpokládá se z>0, n≥0 }
begin
  if n==0 then pow := 1
          else pow := z * pow(z,n-1)
end
```

Věta: Funkce pow konverguje vzhledem ke vstupní podmínce $\varphi(z,n) \equiv z \in \mathbb{R}, n \in \mathbb{N}, z > 0$.

Důkaz: Klesající hodnotou je zde exponent.

Věta: Funkce pow je parciÆlně korektní vzhledem ke vstupní podmínce φ a k výstupní podmínce $\psi(z,n,r) \equiv r = z^n$.

Důkaz věty: Indukcí podle exponentu.

Příklad: Algoritmus umocňovÆnčísla půlením exponentu (imperativní verze – Pascal)

```
function power (z:Real; n:Integer): Real;
{ Předpokládá se z>0, n≥0 }
var y,r:Real; k:Integer;
begin
  k := n;
  y := z;
  r := 1;
  while k > 0 do
    begin
      if odd(k) then r := r * y;
      k := k div 2;
      y := sqr(y)
    end;
  power := r
end
```

Věta: Funkce `power` konverguje vzhledem ke vstupní podmínce $\varphi(z, n) \equiv z \in \mathbb{R}, n \in \mathbb{N}, z > 0$.

Důkaz: Klesající hodnotou je zde exponent. KlesÆní je tentokrÆt rychlejší než v předchÆzejícím příkladě, ale pro $\mathtt{k} > 0$ je vždy ostrØ.

Věta: Funkce `power` je parciÆlně korektní vzhledem ke vstupní podmínce φ a k výstupní podmínce $\psi(z, n, r) \equiv r = z^n$.

Invariant: $\mathtt{r} \cdot \mathtt{y}^{\mathtt{k}} = z^n \ \wedge \ \mathtt{k} \geq 0$

Věta: Funkce `power` konverguje vzhledem ke vstupní podmínce $\varphi(z, n) \equiv z \in \mathbb{R}, n \in \mathbb{N}, z > 0$.

Důkaz: Klesající hodnotou je zde exponent. KlesÆní je tentokrÆt rychlejší než v předchÆzejícím příkladě, ale pro $\mathtt{k} > 0$ je vždy ostrØ.

Věta: Funkce `power` je parciÆlně korektní vzhledem ke vstupní podmínce φ a k výstupní podmínce $\psi(z, n, r) \equiv r = z^n$.

Důkaz: Invariant cyklu while, který platí vždy v okamžiku testovÆní podmínky cyklu ($\mathtt{k} > 0$), je

$$\mathtt{r} \cdot \mathtt{y}^{\mathtt{k}} = z^n \ \wedge \ \mathtt{k} \geq 0$$

Po poslední iteraci platí tento invariant v konjunkci spolu s negací podmínky cyklu ($\neg(\mathtt{k} > 0)$). Z tØto konjunkce vyplývÆ $\mathtt{r} = z^n$, přičemž `r` je výslednÆ hodnota funkce `power`. Platí tedy i výstupní podmínka $\psi(z, n, \mathtt{power(}z\mathtt{,}n\mathtt{)})$.

Příklad: Algoritmus řazení vklÆdÆním (imperativní verze – Pascal)

```
1   const MAX = 999;
2   type Elem = Integer;
3        Posl = array [1..MAX] of Elem;
4   procedure iInsSort (n:Integer; var A:Posl);
5     var i, j : Integer;  x : Elem;
6     begin
7         for i := 2 to n do
8         begin
9             x := A[i];  j := i-1;
10            while (j>0) && (A[j]>x) do
11            begin
12                A[j+1] := A[j];
13                j := j-1
14            end;
15            A[j+1] := x
16         end
17     end
```

Věta: Algoritmus `iInsSort` je totÆlně korektní vzhledem k podmínkÆm

$\varphi(n, A)$ $\equiv$ posloupnost A je neprÆzdnÆ a $n \geq 1$ je její dØlka

$\psi(n, A, A')$ $\equiv$ posloupnost A' je permutací posloupnosti A a je neklesající

Věta: Nechť $\mathbb{Z}^*$ označuje množinu všech konečných celočíselných posloupností, $In = \mathbb{N} \times \mathbb{Z}^*$, $Out = \mathbb{Z}^*$. Nechť $\varphi(n, A) \equiv n \geq 1 \wedge |A| = n$, $\psi(n, A, A') \equiv A' \in Permut(A) \wedge \forall i, j, 1 \leq i < j \leq n . A'_i \leq A'_j$, Pak `iInsSort` je totálně korektní vzhledem k podmínkám φ, ψ.

Důkaz rozložíme na důkaz konvergence a parciální korektnosti.

Lemma: Algoritmus `iInsSort` je konvergentní vzhledem k φ.

Důkaz: Vnější cyklus for vždy skončí, protože je vždy předem dán pevný počet jeho opakování ($n - 1$). Vnitřní cyklus while se provádí jen když $j > 0$. Ale při každém průchodu tělem tohoto cyklu hodnota proměnné `j` klesne. To znamená, že cyklus while se provede konečněkrát (nejvýše $(\texttt{i} - 1)$-krát).

Lemma: Algoritmus `iInsSort` je parciálně korektní vzhledem k podmínkám φ, ψ.

Důkaz: Vstupní posloupnost označme $(a_1, \ldots, a_n)$, obsah pole A (v danØm stavu výpœtu) označme $(A_1, \ldots, A_n)$.
Invariantem vnějšího cyklu for (splněným vždy na začÆtku tohoto cyklu) je podmínka
$A_1 \leq A_2 \leq \cdots \leq A_{i-1} \ \wedge \ (A_1, \ldots, A_{i-1}) \in \mathit{Permut}(a_1, \ldots, a_{i-1}) \ \wedge \ A_i = a_i, \ldots, A_n = a_n$, takže po posledním průchodu cyklem for platí $A_1 \leq A_2 \leq \ldots \leq A_n$.
Invariantem vnitřního cyklu while (opět splněným vždy na začÆtku cyklu) je podmínka
$A_1 \leq \cdots \leq A_{j-1} \leq A_j = A_{j+1} \leq A_{j+2} \leq \cdots \leq A_i \ \wedge \ (A_1, \ldots, A_j, \ a_i, \ A_{j+2}, \ldots, A_i) \in \mathit{Permut}(a_1, \ldots, a_i)$

(Důkaz, že jde skutečně o invarianty, je snadný.)

FunkcionÆlní paradigma

NeformÆlní definice:

Výpočet je posloupností tzv. *redukčních kroků*, z nichž každý je elementÆrní œpravou programu (výrazu). Výsledkem výpočtu je hodnota, kterou už nelze dÆle zjednodušit.

DØlka výpočtu je pak dØlkou tØto posloupnosti, tj. počet redukčních kroků vedoucích k výsledku.

Pro formÆlní definici je nutno zavØst formÆlní kalkul. Ve funkcionÆlním paradigmatu jím nejčastěji bývÆ jazyk vychÆzející z lambda kalkulu – *funkcionÆlní jazyk*.

Příklad: Funkce `maxim` najde největší číslo z konečnØho neprÆzdnØho seznamčísel

```
maxim [x]       =  x
maxim (x:y:s)   =  if x>y then maxim (x:s)
                          else maxim (y:s)
```

```
 maxim [4,3,5,1]
⇝ if 4>3  then maxim [4,5,1] else maxim [3,5,1]
⇝ if True then maxim [4,5,1] else maxim [3,5,1]

⇝ maxim [4,5,1]
⇝ if 4>5   then maxim [4,1] else maxim [5,1]
⇝ if False then maxim [4,1] else maxim [5,1]

⇝ maxim [5,1]
⇝ if 5>1  then maxim [5] else maxim [1]
⇝ if True then maxim [5] else maxim [1]

⇝ maxim [5]

⇝ 5
```

Příklad: Algoritmus řazení vklÆdÆním (funkcionÆlní verze – Haskell)

```
fInsSort []     =  []
fInsSort (x:s) =  ins x (fInsSort s)
  where ins x []     = [x]
        ins x (y:t) = if x<=y then x:(y:t)
                              else y:(ins x t)
```

Nechť *In* $=$ *Out* je množina všech seznamů (posloupností) celých čísel,

$\varphi(s) \equiv s$ je konečný,

$\psi(s,t) \equiv$ „t je permutací seznamu s a t je neklesající“.

Věta: AlgoritmusPak `fInsSort` je totÆlně korektní vzhledem k φ, ψ.

PoznÆmka: V Haskellu lze definici zapsat i kratčeji:

`fInsSort =  foldr ins []`.

Věta: AlgoritmusPak `fInsSort` je totÆlně korektní vzhledem k φ, ψ.

Důkaz: Nechť s je konečný seznam dØlky n. Konvergence obou funkcí se ukÆže indukcí přes dØlku seznamu.

Při důkazu parciÆlní korektnosti vyjdeme z vlastnosti funkce `ins`: Je-li seznam u neklesající, pak `ins` y u je neklesající permutace seznamu y`:`u. Zbytek indukcí přes dØlku seznamu s.

Složitost

Složitost *algoritmu* vyjadřuje náročnost algoritmu na různé zdroje v průběhu výpočtu: čas výpočtu, velikost paměti, počet procesorů apod. Podle toho rozlišujeme různé *míry složitosti*.

Míry složitosti:
- časová
- prostorová (paměťová)
- procesorová (hardwarová)
- ⋮

Pro definici časové resp. prostorové složitosti zavedeme pojmy *délka výpočtu* resp. *množství výpočtem spotřebované paměti*. K přesné definici obou pojmů však potřebujeme tzv. *výpočetní model*. Jeho definice závisí na *výpočetním paradigmatu*.

Například pro Turingův stroj dÉlkou výpočtu počet výpočetních kroků, tj. vnitřních přechodů. Pro RAM je to počet provedených instrukcí. Pro funkcionÁlní jazyk je to počet jednokrokových redukcí.

Složitost *problÉmu* je je zavedena jako složitost optimÁlního algoritmu řešícího tento problÉm. Na řešení nějakÉho problÉmu můžeme použít mnoho různých algoritmů. Časovou složitostí problÉmu pak rozumímečasovou složitost algoritmu, který řeší daný problÉm (i na „nejpomalejších datech“) nejrychleji. Prostorovou složitostí problÉmu rozumíme prostorovou složitost algoritmu, který řeší daný problÉm (i na nejmÉně příznivých datech) v nejmenší paměti apod.

Příklad:

- Řešíme problØm: seřadit vzestupně n-prvkovou posloupnost celých čísel.
- K řešení použijeme algoritmus řazení vklÆdÆním.
- Naše konkrØtní posloupnosti, kterØ budou vstupem, jsou $3, 6, 9, \ldots, 3n$ (tj. jsou už seřazenØ).

Rozlišujeme tři pojmy:

- časovou složitost problØmu
- časovou složitost algoritmu
- dØlku výpočtu

DØlka výpočtu algoritmem `iInsSort` na rostoucí posloupnosti dØlky n je $6n - 4$ (tolik se provede výpočetních kroků).

ČasovÆ složitost algoritmu `iInsSort` je vyjÆdřena kvadratickým polynomem $an^2 + bn + c$, jak později uvidíme. Je tedy větší než dØlka výpočtu na našich (příznivých) datech. Na jiných posloupnostech dØlky n stejný algoritmus strÆví čas více. V nejhorším případě až kvadraticky mnoho, proto je časovÆ složitost algoritmu řazení vklÆdÆním kvadratickÆ.

ČasovÆ složitost problØmu vzestupnØho seřazení posloupnosti je však lepší než kvadratickÆ: existují algoritmy, kterØ n-prvkovou posloupnost seřadí v čase $c\,n \log n$ (c je konstanta), a to i v nejhorším případě (tj. i pro „nejzlomyslněji zadanØ“ vstupní posloupnosti). Proto je časovÆ složitost problØmu řazení vyjÆdřena funkcí $c\,n \log n$ (v proměnnØ n).

DØlka výpočtu výrazů

DØlku výpočtu výrazu e označíme $\tau(e)$.

$\tau(e)$ je rovna součtu dØlek vyhodnocení všech operací, kterØ výraz e obsahuje.

Vyhodnocení elementÆrních aritmetických, logických, relačních operací nad skalÆrními operandy je konstantní. Proto i dØlku výpočtu výrazu, který obsahuje pouze tyto jednoduchØ operace, považujeme za konstantní.

Obsahuje-li však výraz e volÆní složitějších funkcí, je nutno započítat dØlku jejich výpočtu do $\tau(e)$.

Přesněji:
$\tau(c) = 0$, je-li c konstanta,
$\tau(v) = 1$, je-li c odkaz na hodnotu skalÁrní (přepisovatelnÓ) proměnnÓ. Tedy zpřístupnění obsahu proměnnÓ považujeme za jednotkovou operaci.

$\tau(op\ a) = 1 + \tau(a)$, kde *op* je primitivní unÁrní operÁtor, který vyhodnocuje svůj operand, například `not`.

$\tau(a \oplus b) = 1 + \tau(a) + \tau(b)$, kde $\oplus$ je některý z primitivních binÁrních operÁtorů, kterÓ vyhodnocují oba svoje operandy, například $+, -, *, /$, `div`, `mod`, $<, \leq, >, \geq, \ldots$.

Pozor, některØ operÆtory a jazykovØ konstrukce nemusí vždy vyhodnocovat všechny svoje operandy. Podle toho se pak určuje dØlka výpočtu výrazů:

$\tau(a \texttt{ \&\& } b) = 1 + \tau(a) + \tau(b)$ pro a pravdivØ
$\tau(a \texttt{ \&\& } b) = 1 + \tau(a)$ pro a nepravdivØ

$\tau(a \texttt{ || } b) = 1 + \tau(a) + \tau(b)$ pro a nepravdivØ
$\tau(a \texttt{ || } b) = 1 + \tau(a)$ pro a pravdivØ

$\tau(\texttt{if } a \texttt{ then } b \texttt{ else } c) = 1 + \tau(a) + \tau(b)$ pro a pravdivØ
$\tau(\texttt{if } a \texttt{ then } b \texttt{ else } c) = 1 + \tau(a) + \tau(c)$ pro a nepravdivØ

DØlka výpočtu volÆní funkce

Deklarace:
`f (`x_1`,...,`x_n`) =` e
VolÆní hodnotou (striktní aplikace):
$\tau(\texttt{f}(a_1,\ldots,a_n)) = 1 + \tau(a_1) + \cdots + \tau(a_n) + \tau(e'')$, kde výraz e'' vznikne z výrazu e nahrazením každØho výskytux_i *hodnotou* výrazu a_i.

VolÆní jmØnem (normÆlní aplikace):
$\tau(\texttt{f}(a_1,\ldots,a_n)) = 1 + \tau(e')$, kde výraz e' vznikne z výrazu e nahrazením každØho výskytu x_i *celým nevyhodnoceným výrazem* a_i.

Bude-li nÆs zajímat pouze tzv. *asymptotickØ chovÆní* složitosti algoritmů, budeme ztotožňovat složitosti lišící se jen kladnou multiplikativní konstantou. Pro takovØ případy bude užitečnÆ nÆsledující

Úmluva: Je-li e výraz obsahující jen skalÆrní konstanty a proměnnØ a elementÆrní operace (na skalÆrních hodnotÆch), klademe dØlku jeho výpočtu rovnu jednØ.

Výraz e zejmØna nesmí obsahovat vektorovØ operace (např. aritmetickØ operace na „dlouhých“ číslech) ani volÆní uživatelských funkcí.

Tuto œmluvu zavÆdíme proto, že pročeœ dy zjišťovÆní asymptotickØho chovÆní algoritmů je praktičtější například uvažovat dØlku přiřazení `x:=a*(b+c+d)` rovnu jednØ, než ji počítat jako $1 + 1 + 2 + 5$. Stejně víme, že součet je roven konstantě, a to nÆm stačí.

U tzv. „čistých“ výrazů předpokládáme, že nemají vedlejší efekty. To však nemusí vždy platit. Délka výpočtu výrazů, které obsahují volání funkcí, závisí nejen na samotném výrazu, ale i na *stavu*, v němž se výraz vyhodnocuje. Stavem rozumíme okamžitý obsah programových (přepisovatelných) proměnných. V jazycích s vedlejšími efekty se stav během výpočtu mění.

Příklad:

```
var a:Integer;
function f (n:Integer):Integer;
 var i,k:Integer;
 begin k := 0;
       for i:=0 to n*a do
           k := k+i;
       a := k;
       f:=a  {return a}
 end
```

GlobÆlní přepisovatelnÆ proměnnÆ a mÆ na začÆtku hodnotu 2, pak ve výrazu `f(4)+f(4)` trvÆ každØ vyhodnocení stejnØho podvýrazu `f(4)` různou dobu. (Jakou?)

DØlka výpočtu příkazů

Nechť S je množina stavů, $\sigma \in S$. Každý příkaz p funguje jako *stavový transformÆtor* $\nu(p) : S \dashrightarrow S$.

DØlku výpočtu příkazu p ve stavu σ označíme $\tau(p, \sigma)$.

PrÆzdný příkaz

$\tau(\texttt{skip}, \sigma) = 0$ pro libovolný stav σ

Přiřazovací příkaz

$\tau(\texttt{v:=}e, \sigma) = 1 + \tau(e, \sigma)$, je-li v jednoduchÆ přepisovatelnÆ proměnnÆ

Sekvence

$$\tau(\texttt{begin}\, p_1\,\texttt{;}\,\ldots\,\texttt{;}\, p_k\,\texttt{end}, \sigma_0) = \sum_{i=1}^{k} \tau(p_i, \sigma_{i-1}), \quad \text{kde } \sigma_i = \nu(p_i)(\sigma_{i-1})$$

Větvení

Nechť σ je stav, p, q příkazy, e výraz a $\sigma' = \nu(e)(\sigma)$.

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma')$ pro pravdivØ e,

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma')$ pro nepravdivØ e.

Pokud výraz e nemÆ vedlejší efekty, pak $\sigma = \sigma'$ a

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma)$ pro pravdivØ e,

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma)$ pro nepravdivØ e.

Cyklus while

Nechť $t = (\texttt{while}\ \ e\ \ \texttt{do}\ \ s)$ je příkaz cyklu a σ_n je stav takový, že ze stavu σ_0 příkaz cyklu t zopakuje právě n-krát svůj tělo s.

Pro $0 \leq i \leq n$ označme $\sigma'_i = \nu(e)(\sigma_i)$,

pro $0 \leq i < n$ označme $\sigma_{i+1} = \nu(s)(\sigma_i)$,

a nechť pro $0 \leq i < n$ je $\mu(e)(\sigma_i) =$ pravda, $\mu(e)(\sigma_n) =$ nepravda. Pak

$$\tau(t, \sigma_0) = \sum_{0 \leq i < n} \Big(\tau(e, \sigma_i) + \tau(s, \sigma'_i)\Big) + \tau(e, \sigma_n)$$

Pokud výraz e nemá vedlejší efekty, pak $\sigma_i = \sigma'_i$ a

$$\tau(t, \sigma_0) = \sum_{0 \leq i < n} \Big(\tau(e, \sigma_i) + \tau(s, \sigma_i)\Big) + \tau(e, \sigma_n)$$

$\hat{T}$ značí množinu všech hodnot typu T, tzv. *domØnu*

StejnØ rozšíření definice dØlky výpočtu se vztahuje i na výrazy s vedlejšími efekty.

Příkazy můžeme považovat za výrazy, kterØ mají speciÆlní hodnotu $\bullet$ typu $\mathtt{Com}$. SØmantika každØho příkazu je pak konstantní funkce, tj. pro libovolný příkaz p a stav σ je $\mu(p)(\sigma) = \bullet$.

Nechť `f` je funkce definovanÆ
`f (`x_1`,...,`x_n`) =` e
a σ_0 je stav, ve kterØm začne výpočet.
Pak buďto, v případě volÆní jmØnem,
$\tau(\texttt{f}(a_1,\ldots,a_n),\sigma_0) = 1 + \tau(e',\sigma_0)$, kde výraz e' vznikne z výrazu e (těla funkce `f`) nahrazením každØho výskytux_i *výrazem* a_i
anebo, v případě volÆní hodnotou,
$\tau(\texttt{f}(a_1,\ldots,a_n),\sigma_0) = 1 + \tau(a_1,\sigma_0) + \tau(a_2,\sigma_1) + \cdots + \tau(a_n,\sigma_{n-1}) + \tau(e'',\sigma_n)$,
kde $\sigma_i = \nu(a_i)(\sigma_{i-1})$ (pro $0 < i \leq n$) a výraz e'' vznikne z těla e takto:

$$
\begin{array}{lll}
e'' = & \texttt{begin} & x_1 \texttt{:=} \mu(a_1)(\sigma_0)\texttt{;} \\
 & & \vdots \\
 & & x_n \texttt{:=} \mu(a_n)(\sigma_{n-1})\texttt{;} \\
 & & e \\
 & \texttt{end} &
\end{array}
$$

DØlku výpočtu příkazu cyklu for můžeme odvodit tak, že si cyklus for vyjÆdíme pomocí cyklu while.

Nechť $t = (\texttt{for i:=}e\ \texttt{to}\ e'\ \texttt{do}\ s)$ je příkaz cyklu a σ je stav takový, že ze stavu σ příkaz cyklu t zopakuje prÆvě n-krÆt svØtělo s.

Označíme $\sigma_0 = \nu(\texttt{i:=}e)(\sigma)$,
pro $0 \leq i \leq n$ označíme $\sigma'_i = \nu(\texttt{i} \leq e')(\sigma_i)$,
a pro $0 \leq i < n$ označíme $\sigma''_i = \nu(s)(\sigma'_i)$, $\sigma_{i+1} = \nu(\texttt{i:=succ(i)})(\sigma''_i)$

Pak

$$\tau(t,\sigma) = \tau(\texttt{i:=}e)(\sigma) + \sum_{0 \leq i < n} (\tau(\texttt{i} \leq e', \sigma_i) + \tau(s, \sigma'_i) + \tau(\texttt{i:=succ(i)}, \sigma''_i)) + \tau(\texttt{i} \leq e', \sigma_n)$$

Většinou nÆs víc než dØlka jednoho konkrØtního výpočtu zajímÆ, jak se daný algoritmus chovÆ obecně, tj. jak vypadÆ množina dØlek všech možných výpočtů — pro množinu všech přípustných vstupních dat (vyhovujících vstupní podmínce).

ČasovÆ složitost algoritmu

Je-li *In* množina všech vstupních dat pro algoritmus A, pak pro $x \in$ *In* zavedeme číslo $|x| \in \mathbb{N}$, kterØ nazveme *velikost* vstupní hodnoty x.

PočÆteční stav výpočtu, který odpovídÆ umístění hodnoty x na vstupu, označíme σ_x.

Časovou složitost algoritmu A zavedeme jako jako funkci $T_A : \mathbb{N} \to \mathbb{N}$ takto:

$$T_A(n) = \max\{\tau(A, \sigma_x) \mid |x| = n\}$$

ČasovÆ složitost algoritmu je tedy funkce, kterÆ pro každou velikost vstupních dat je rovna dØlce *nejdelšího* výpočtu na všech možných datech tØto velikosti.

Velikost vstupních dat odpovídÆ počtu bitů, kterými je vstupní œdaj vyjÆřen. V praxi mÆ často tyto významy:

číslo n	dØlka jeho bitovØho zÆpisu, tj. $\lceil \log_2 n \rceil$
posloupnost čísel	počet jejích prvků
graf	počet uzlů $+$ počet hran

Protože množina v definici složitosti je většinou nekonečnÆ, musíme hledanØ maximum určit *analýzou nejhoršího případu*.

V některých případech je výhodnější rozdělit velikost dat na dvě čísla (graf) a upravit definici složitosti $T_A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

Příklad: Algoritmus řazení vklÆdÆním (imperativní verze – Pascal)

```
1   const MAX = 999;
2   type Elem = Integer;
3        Posl = array [1..MAX] of Elem;
4   procedure iInsSort (n:Integer; var A:Posl);
5     var i, j : Integer;  x : Elem;
6     begin
7         for i := 2 to n do
8         begin
9             x := A[i];  j := i-1;
10            while (j>0) && (A[j]>x) do
11            begin
12                A[j+1] := A[j];
13                j := j-1
14            end;
15            A[j+1] := x
16         end
17     end
```

Příklad: Analýza nejhoršího případu

Algoritmus: `iInsSort`
Vstup: posloupnost celých čísel $A = (a_1, \ldots, a_n)$
dØlky n.
Výstup: posloupnost celých čísel B, kterÆ je permutací posloupnosti A a přitom je neklesající.

Označme c_i dØlku výpočtu příkazu nebo testu na i-tØmřÆdku algoritmu `iInsSort`, přičemž na sedmØmřÆdku (v zÆhlaví cyklu for) jsou uvedeny elementÆrní akce $c_{7,\text{init}}$, $c_{7,\text{test}}$, $c_{7,\text{inc}}$.

Pak dØlka výpočtu je

$$T(n) = c_{7,\text{init}} + \sum_{i=2}^{n} \left(c_{7,\text{test}} + c_9 + \sum_{j=\xi}^{i-1} (c_{10} + c_{12} + c_{13}) + c_{10} + c_{15} + c_{7,\text{inc}} \right) + c_{7,\text{test}}$$

kde ξ je hodnota, při kterØ se vnitřní cyklus algoritmu zopakuje nejvícekrÆt, tj. nejmenší možnÆ hodnotaj, při níž je ještě splněna podmínka za while (na 10. řÆdku algoritmu).

TakovØ minimÆlní možnØ ξ je zřejmě rovno 1, a to v každØm průchodu vnějším cyklem for. Situace, kdy v každØm průchodu vnějším cyklem je $\xi = 1$, nastane, když vstupní posloupnost A je klesající.

Po dosazení $\xi = 1$ a po úpravě dostáváme

$$T(n) = \frac{c_{10} + c_{12} + c_{13}}{2} n^2 + (c_{7,\text{test}} + c_{7,\text{inc}} + c_9 + \frac{c_{10}}{2} - \frac{c_{12}}{2} - \frac{c_{13}}{2} + c_{15}) n + (c_{7,\text{init}} - c_{7,\text{inc}} - c_9 - c_{10} - c_{15})$$

což je kvadratický polynom v proměnné n.

Podle dřívější úmluvy lze všechny konstanty považovat za jedničky (resp. $c_9 = 2$), takže po dosazení lze pracovat s polynomem $\frac{3}{2} n^2 + \frac{9}{2} n - 4$.

Abychom mohli při porovnÆvÆní složitostí algoritmů abstrahovat od rychlosti konkrØtního počítače, budeme chtít srovnÆvat funkce podle toho, „jak rychle rostou“, přičemž za významný rozdíl v rychlosti růstu nepovažujeme například případ, kdy jedna funkce mÆ c-krÆt větší hodnoty než druhÆ (pro nějakou kladnou konstantu c).

Rychlost růstu funkcí

Nechť $g : \mathbb{N} \to \mathbb{N}$. Zavedeme množiny funkcí:

Množina funkcí rostoucích nejvýše tak rychle jako g :

$$O(g) = \{f \mid \exists c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) \leq c \cdot g(n)\}$$

Množina funkcí rostoucích aspoň tak rychle jako g :

$$\Omega(g) = \{f \mid \exists c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) \geq c \cdot g(n)\}$$

Množina funkcí rostoucích stejně rychle jako g :

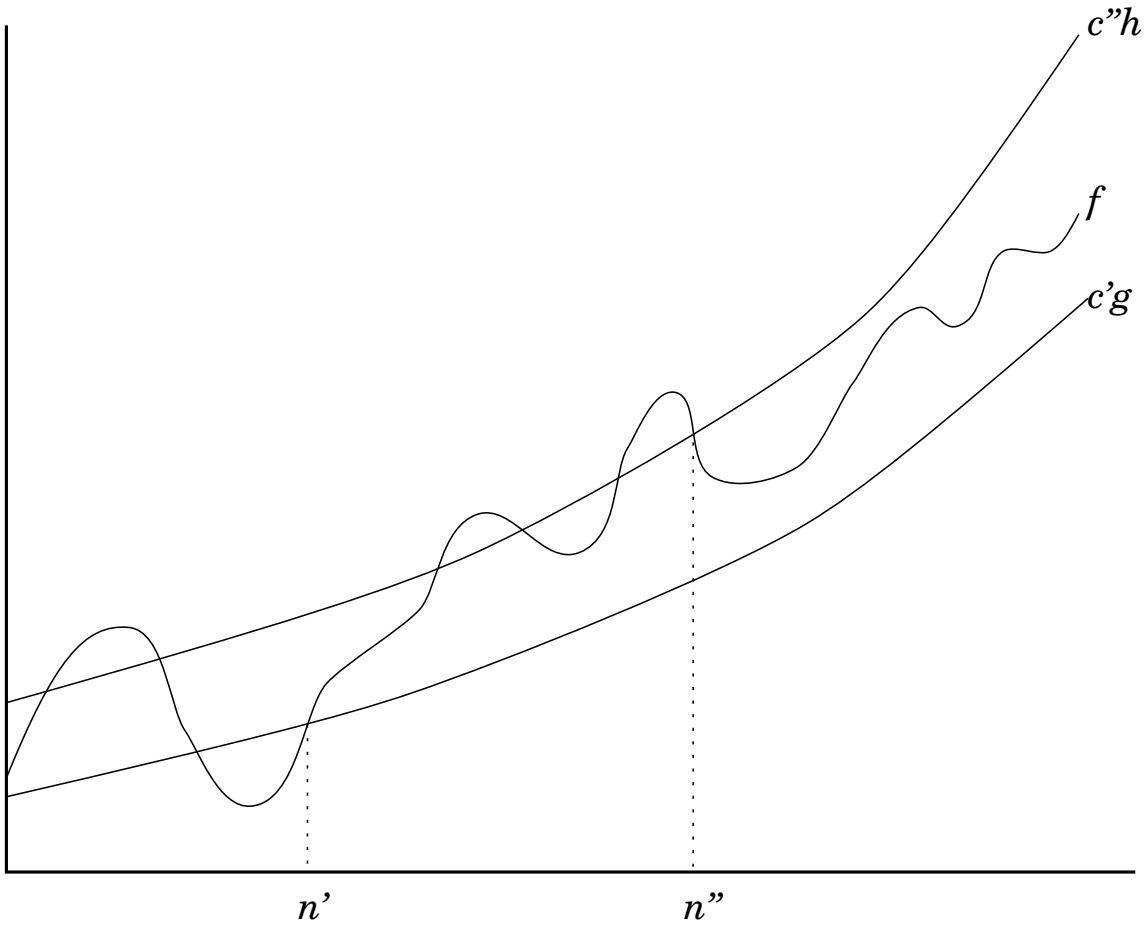
$$\Theta(g) = O(g) \cap \Omega(g)$$

Množina funkcí rostoucích pomaleji než g :

$$o(g) = \{f \mid \forall c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) < c \cdot g(n)\}$$

Množina funkcí rostoucích rychleji než g :

$$\omega(g) = \{f \mid \forall c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq c \cdot g(n) < f(n)\}$$



$$f \in \Omega(g) \Leftrightarrow \forall n \geq n'.\, 0 \leq c'g(n) \leq f(n)$$

$$f \in O(h) \Leftrightarrow \forall n \geq n''.\, f(n) \leq c''h(n)$$

Protože se budeme zabývat především funkcemi vyjadřujícími složitost algoritmu (kterÆ je vždy nezÆpornÆ), omezíme se dÆle na nezÆpornØ funkce.

Vlastnosti množin O, Ω, Θ, o, ω

Věta: Jsou-li $f, g : \mathbb{N} \to \mathbb{N}$ dvě funkce, pak $f \in \Theta(g)$ právě když

$$\exists c_1 > 0 \exists c_2 > 0 \exists n_0 \forall n \geq n_0 .\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Věta: Pro každé dvě kladné funkce $f, g : \mathbb{N} \to \mathbb{N}$ platí:

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$
$$f \in o(g) \Leftrightarrow g \in \omega(f)$$
$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$
$$f \in o(g) \Rightarrow f \in O(g)$$
$$f \in \omega(g) \Rightarrow f \in \Omega(g)$$

V důkazu prvních dvou tvrzení stačí obrÆtit nerovnost a uvÆžit kladnou konstantu $\frac{1}{c}$.

Třetí tvrzení je triviÆlní.

ČtvrtØ tvrzeníříkÆ, že jistÆ vlastnost platí pro všechna kladnÆ. Ale kladnÆčísla existují, například $c = 1$. Tedy platí takØ slabší vlastnost pro *nějakØ* $c > 0$.

Důkaz pÆtØho tvrzení je analogický.

Věta: Pro každØ dvě kladnØ funkce $f, g : \mathbb{N} \to \mathbb{N}$ platí:

$$f \in o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f \in \omega(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Rovnost $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ je ekvivalentní s podmínkou $\forall c > 0 \exists n_0 \forall n \geq n_0 . \left| \frac{f(n)}{g(n)} \right| < c$. Obě funkce jsou kladnØ, tedy absolutní hodnotu nemusíme uvažovat a nerovnost lze vynÆsobit číslem $g(n)$, čímž dostaneme podmínku pro $f \in o(g)$. Úprava byla ekvivalentní, takže ekvivalentní jsou i obě podmínky v prvním tvrzení.

Rovnost $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ je ekvivalentní s podmínkou $\forall c > 0 \exists n_0 \forall n \geq n_0 . \left| \frac{f(n)}{g(n)} \right| > c$. Obě funkce jsou kladnØ, tedy absolutní hodnotu nemusíme uvažovat a nerovnost lze vynÆsobit číslem $g(n)$, čímž dostaneme podmínku pro $f \in \omega(g)$. Úprava byla ekvivalentní, takže ekvivalentní jsou i obě podmínky v prvním tvrzení.

Jsou-li $f, g : \mathbb{N} \to \mathbb{N}$ dvě kladnØ funkce, pak platí

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \;\Rightarrow\; f \in O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \;\Rightarrow\; f \in \Omega(g)$$

ObrÆcenØ implikace však obecně neplatí, protože uvedenØ limity nemusí existovat.

Pro funkce f, g však platí silnější tvrzení:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \;\Leftrightarrow\; f \in O(g)$$

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \;\Leftrightarrow\; f \in \Omega(g)$$

Cvičení: Rozhodněte a zdůvodněte, zda pro každou kladnou funkci g platí
$O(g) = o(g) \cup \Theta(g),\ \Omega(g) = \omega(g) \cup \Theta(g)$.

Věta: Nechť $f, g, h : \mathbb{N} \to \mathbb{N}$ jsou tři kladnØ funkce a nechť funkce $q : \mathbb{N} \to \mathbb{N}$ je definovÆna $q(n) = f(n) + g(n)$.

Pak platí:

$$f \in O(h) \wedge g \in O(h) \Rightarrow q \in O(h)$$

$$f \in \Omega(h) \wedge g \in \Omega(h) \Rightarrow q \in \Omega(h)$$

$$f \in \Theta(h) \wedge g \in \Theta(h) \Rightarrow q \in \Theta(h)$$

$$f \in o(h) \wedge g \in o(h) \Rightarrow q \in o(h)$$

$$f \in \omega(h) \wedge g \in \omega(h) \Rightarrow q \in \omega(h)$$

PoznÆmka: Platí řada podobných odvozených vlastností, například

$$f \in \Omega(h) \wedge g \in O(h) \Rightarrow q \in \Omega(h)$$
$$f \in \Theta(h) \wedge g \in O(h) \Rightarrow q \in \Theta(h)$$
$$f \in \Theta(h) \wedge g \in o(h) \Rightarrow q \in \Theta(h)$$

apod.

Cvičení: Formulujte a dokažte další vlastnosti, kterØ z věty vyplývají.

Věta: Nechť $f, h : \mathbb{N} \to \mathbb{N}$ jsou dvě kladnØ funkce, c, d konstanty, $c > 0$, a nechť funkce $p : \mathbb{N} \to \mathbb{N}$ je definovÆna $p(n) = c \cdot f(n) + d$.

Pak platí:

$$f \in O(h) \Rightarrow p \in O(h)$$

$$f \in \Omega(h) \Rightarrow p \in \Omega(h)$$

$$f \in \Theta(h) \Rightarrow p \in \Theta(h)$$

$$f \in o(h) \Rightarrow p \in o(h)$$

$$f \in \omega(h) \Rightarrow p \in \omega(h)$$

Důkaz: Nejprve nechť $d = 0$; tvrzení obdržíme vhodnou volbou kladnØ konstanty v definiích tříd O, Ω, Θ, o, ω. Pro obecnØ $d \neq 0$ jde o speciÆlní případ předchozí věty, kdy funkce g z jejích předpokladů je konstantní.

Věta: Nechť $f, g, h : \mathbb{N} \to \mathbb{N}$ jsou tři funkce, pak

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$
$$f \in o(g) \wedge g \in O(h) \Rightarrow f \in o(h)$$
$$f \in O(g) \wedge g \in o(h) \Rightarrow f \in o(h)$$
$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$$
$$f \in \omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \omega(h)$$
$$f \in \Omega(g) \wedge g \in \omega(h) \Rightarrow f \in \omega(h)$$

Důsledek: Relace *„růst nejvýše tak rychle“* tvoří *předuspořádání* na množině všech funkcí $\mathbb{N} \to \mathbb{N}$.

Poznámka: Relace *„růst právě tak rychle“* tvoří *ekvivalenci* na množině všech funkcí $\mathbb{N} \to \mathbb{N}$.

Tyto relace však nejsou úplné. Existují dvojice funkcí, které nejsou porovnatelné.

PoznÆmka: *PředuspořÆdÆní* (též *polouspořÆdÆní*) je binÆrní relace, kterÆ je reflexivní a transitivní. *UspořÆdÆní* je binÆrní relace, kterÆ je reflexivní, antisymetrickÆ a transitivní. *Ekvivalence* je binÆrní relace, kterÆ je reflexivní, symetrickÆ a transitivní.

Cvičení: Najděte příklad dvojice funkcí $f, g : \mathbb{N} \to \mathbb{N}$, kterØ nejsou porovnatelnØ, tj. $f \notin O(g), f \notin \Omega(g)$.

Cvičení: Dokažte tvrzení:
Je-li kladnÆ funkce f skoro všude majorizovÆna funkcí g (tj. $f(n) \leq g(n)$ pro skoro všechna n), pak $f \in O(g)$.

Cvičení: Dokažte tvrzení:
Pro každou kladnou funkci g je $o(g) \cap \omega(g) = \emptyset$.

Růst jednoduchých funkcí

Def: *Kladný polynom* je polynom s kladným vedoucím koeficientem.

Věta: Kladný polynom vyššího stupně roste vždy rychleji než polynom nižšího stupně. Dva kladnØ polynomy stejnØho stupně rostou stejně rychle.

PoznÆmka: Předchozí větu lze zobecnit i na mocninnØ funkce s neceločíselnými exponenty: jestliže $0 \leq a < b$, pak $n^a \in o(n^b)$.

Věta: ExponenciÆlní funkce se zÆkladem $a > 1$ roste rychleji než libovolný polynom. LogaritmickØ funkce rostou pomaleji než funkce lineÆrní.

Cvičení: Dokažte, že pro každé přirozené číslo k platí $(n+1)^k \in O(n^k)$.
(Vhodnou kladnou konstantu c z definice množiny O určete podle binomické věty.)

Věta: Nechť $1 < a < b$. Pak

$$\log_a \in \Theta(\log_b)$$

$$a^n \in o(b^n)$$

PoznÆmka: Roste-li funkce logaritmicky, pak na zÆkladu logaritmu nezÆleží. (Jen musí být větší než 1, což je však nutnÆ podmínka k tomu, aby funkce vůbec rostla.)

Věta: Platí $n! \in O(n^n)$, $n! \in \Omega(2^n)$.

Důkaz plyne z definice faktoriÆlu.

PoznÆmka: Místo „$\Theta(\log)$“ se často píše „$\Theta(\log n)$“, podobně jako „$o(n^2)$“ místo „$o(f)$, kde $f(n) = n^2$ pro každØn“,
a dokonce „$O(1)$“ místo „$O(g)$, kde $g(n) = c > 0$“.
Mnozí autoři zachÆzejí ve zneužívÆní notace ještØdÆle (např. pracují s množinami funkcí, jako by to byla čísla).

PoznÆmka: Z tzv. Stirlingovy aproximace

$$\sqrt{2\pi n}(n/\mathrm{e})^n \leq n! \leq \sqrt{2\pi n}(n/\mathrm{e})^n \mathrm{e}^{1/(12n)}$$

vyplývají silnější tvrzení o růstu faktoriÆlu:

$$n! \in o(n^n), \qquad n! \in \omega(2^n)$$

Def: Řekneme, že funkce f *roste nejvýše polylogaritmicky*, když existuje číslo $k > 0$ tak, že $f \in O\left((\log n)^k\right)$.

Def: Řekneme, že funkce f *roste nejvýše polynomiálně*, když existuje číslo $k > 0$ tak, že $f \in O(n^k)$.

Funkce f roste *roste aspoň polynomiálně*, když existuje číslo $\varepsilon > 0$ tak, že $f \in \Omega(n^\varepsilon)$.

Def: Řekneme, že funkce f *roste nejvýše exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in O(a^n)$.

Funkce f *roste aspoň exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in \Omega(a^n)$.

Def: Řekneme, že funkce f *roste aspoň dvojitě exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in \Omega\left(a^{a^n}\right)$.

PoznÆmka: Existují funkce, kterØ rostou rychleji než všechny k-nÆsobnØ exponenciÆlní funkce, a to dokonce pro libovolnØ k.
Naopak existují kladnØ funkce, kterØ rostou (tedy rychlejinež konstantní), ale rostou pomaleji než funkce $\log \circ \cdots \circ \log$, kde logaritmus můžeme složit sama se sebou libovolněkrÆt.

Def: *Fibonacciho funkce* $F : \mathbb{N} \to \mathbb{N}$ je definovanÆ rekursivnČ:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n+2) &= F(n) + F(n+1) \end{aligned}$$

pro všechna $n \in \mathbb{N}$.

Věta: Nechť $\varphi = \dfrac{1+\sqrt{5}}{2}$, $\hat{\varphi} = \dfrac{1-\sqrt{5}}{2}$. Pak pro každØ n platí $F(n) = \dfrac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$.

Důsledek: Pro Fibonacciho funkci F platí $F \in \Theta(\varphi^n)$, kde $\varphi = \dfrac{1+\sqrt{5}}{2}$.

Důkaz Věty – indukcí podle n.

(i) Pro $n = 0$ a $n = 1$ tvrzení platí:

$$F(0) = \frac{\varphi^0 - \hat{\varphi}^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0, \quad F(1) = \frac{\varphi^1 - \hat{\varphi}^1}{\sqrt{5}} = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = 1$$

(ii) Předpokládejme, že pro nějaké $n \geq 0$ platí $F(n) = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$,

$F(n+1) = \frac{\varphi^{n+1} - \hat{\varphi}^{n+1}}{\sqrt{5}}$, a ukážeme, že $F(n+2) = \frac{\varphi^{n+2} - \hat{\varphi}^{n+2}}{\sqrt{5}}$.

$$F(n+2) = F(n) + F(n+1) = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}} + \frac{\varphi^{n+1} - \hat{\varphi}^{n+1}}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n + \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}} =$$

$$\frac{4\left(1+\sqrt{5}\right)^n - 4\left(1-\sqrt{5}\right)^n + 2\left(1+\sqrt{5}\right)^{n+1} - 2\left(1-\sqrt{5}\right)^{n+1}}{2^{n+2}\sqrt{5}} =$$

$$\frac{\left(4+2(1+\sqrt{5})\right)\left(1+\sqrt{5}\right)^n-\left(4+2(1-\sqrt{5})\right)\left(1-\sqrt{5}\right)^n}{2^{n+2}\sqrt{5}}=$$
$$\frac{\left(1+2\sqrt{5}+5\right)\left(1+\sqrt{5}\right)^n-\left(1-2\sqrt{5}+5\right)\left(1-\sqrt{5}\right)^n}{2^{n+2}\sqrt{5}}=$$
$$\frac{\left(1+\sqrt{5}\right)^{n+2}-\left(1-\sqrt{5}\right)^{n+2}}{2^{n+2}\sqrt{5}}=\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+2}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+2}}{\sqrt{5}}=\frac{\varphi^{n+2}-\hat{\varphi}^{n+2}}{\sqrt{5}}$$

Lemma: Funkce $\log(n!)$ roste stejně rychle jako funkce $n \cdot \log n$.

Důkaz: Nejdříve ukÆžeme, že $\log_2 n! \in O(n \log n)$.

$\log_2 n! = \sum_{k=1}^{n} \log_2 k \leq \sum_{k=1}^{n} \log_2 n = n \log n$, takže $\log_2 n! \in O(n \log n)$.

DÆle funkci $\log_2 n!$ ohraničíme i zdola: $2 \log_2 n! = 2 \sum_{k=1}^{n} \log_2 k \geq 2 \sum_{k=\lfloor n/2 \rfloor}^{n} \log_2 k \geq$

$2 \sum_{k=\lfloor n/2 \rfloor}^{n} \log_2 \left\lfloor \frac{n}{2} \right\rfloor = 2 \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor \in \Theta(n \log n)$, takže $\log_2 n! \in \Omega(n \log n)$.

Dohromady $\log n! \in \Theta(n \log n)$.

Cvičení: Dokažte, že dØlky výpočtů podle nějakØho algoritmu A jsou v množině $\Theta(g)$, prÆvě když časovÆ složitost algoritmu A je v $O(g)$ a dØlky nejkratších výpočtů patří do $\Omega(g)$.

Cvičení: Seřaďte podle rychlosti růstu funkce (v proměnnØ n): log n^n, log(log n), $n\sqrt{n}$, $2^{\log_3 n}$.

Algoritmus řazení slučovÆním (funkc. verze)

```
mergeSort      ::  [Int] → [Int]
mergeSort []    =  []
mergeSort [x]   =  [x]
mergeSort s     =  merge (mergeSort u) (mergeSort v)
                   where (u,v) = splitAt (n `div` 2) s
                         n     = length s

merge s []           =  s
merge [] t           =  t
merge (x:u) (y:v)    =  if x ≤ y  then  x : merge    u  (y:v)
                                 else  y : merge (x:u)    v
```

Příklad:

```
MS [4,3,1,6,7,2,8,5]

mr (MS [4,3,1,6]) (MS [7,2,8,5])

mr (mr (MS [4,3]) (MS [1,6])) (mr (MS [7,2]) (MS [8,5]))

mr (mr (mr (MS [4])(MS [3]))(mr (MS [1])(MS [6]))) (mr (mr (MS [7])(MS [2]))(mr (MS [8])(MS [5])))

mr (mr (mr [4] [3]) (mr [1] [6])) (mr (mr [7] [2]) (mr [8] [5]))

mr (mr [3,4] [1,6]) (mr [2,7] [5,8])

mr [1,3,4,6] [2,5,7,8]

[1,2,3,4,5,6,7,8]
```

Věta: Nechť *In* $=$ *Out* je množina všech seznamů (posloupností) celých čísel, $\varphi(s) \equiv s$ je konečný, $\psi(s,t) \equiv t$ je permutací seznamu s a t je neklesající. Pak `mergeSort` je totálně korektní vzhledem k podmínkám φ, ψ.

Lemma: Časová složitost pomocné funkce `merge` je $T_{\texttt{merge}} \in \Theta(n)$.

Věta: Časová složitost funkce `mergeSort` je $T_{\texttt{mergeSort}} \in \Theta(n \cdot \log n)$.

Důkaz totÆlní korektnosti: Konvergence i parciÆlní korektnost se dokÆže indukcí podle dØlky seznamu s, parciÆlní korektnost pomocnØ funkce `merge` se dokÆže indukcí podle součtu dØlek obou slučovaných seznamů.

LineÆrní složitost funkce `merge` je zřejmÆ: konstantní dØlka porovnÆní a připojení menšího prvku na začÆtek posloupnosti se dohromady provede tolikrÆt, kolik je součet dØlek obou vstupních posloupností (argumentů funkce `merge`).

Odvození složitosti funkce `mergeSort` je na nÆsledujících stranÆch.

Algoritmus řazení slučovÆním (imp. verze)

```
type Elem = Integer;  Posl = array [1..MAX] of Elem;

procedure mergeSort (n:Integer; var A:Posl);

  var B : Posl; { pomocné pole }

  procedure msort (p,r:Integer); { seřadí pole od p do r }
     var q : Integer;
     begin if p < r then begin
                           q := ⌊(p+r)/2⌋ ;
                           msort (p,q) ;
                           msort (q+1,r) ;
                           merge (p,q,r)
                         end
     end
```

```
procedure merge (p,q,r:Integer);
   { sloučí úsek A[p],...,A[q] s úsekem A[q+1],...,A[r] }
   var i,j : Integer;
   begin
     i := p ;  j := q + 1 ;
     for k := p to r do
       if        (i≤q) && (j≤r) && (A[i]≤A[j]) || (j>r)
         then     begin B[k] := A[i] ; i := i+1 end
         else if (i≤q) && (j≤r) && (A[i]>A[j]) || (i>q)
         then     begin B[k] := A[j] ; j := j+1 end ;
     for k := p to r do  A[k] := B[k]
   end


begin { mergeSort }
   msort (1,n)
end   { mergeSort }
```

PoznÆmka: Podmíněný příkaz v těle cyklu for procedury `merge` lze ekvivalentně zapsat kratčeji takto (dokažte)

```
if ( i≤q ) && ( j>r || A[i]≤A[j] )
   then  begin B[k] := A[i] ; i := i+1 end
   else  begin B[k] := A[j] ; j := j+1 end ;
```

ČasovÆ složitost algoritmuřazení slučovÆním

Analýza nejhoršího případu

Pro danou velikost n je dØlka výpočtu vždy stejnÆ.

ČasovÆ složitost

$$T(1) = a$$
$$T(n) = b + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + D(n), \quad D \in \Theta(n)$$

$$T'(1) = a$$
$$T'(n) = b + 2\,T'(n/2) + n \cdot c, \quad T' \in \Theta(T).$$

Analýza nejhoršího případu je zde triviÆlní: pro každou vstupní posloupnost dØlky n algoritmus dělÆ stejnou prÆci. Pro složitostní $T : \mathbb{N} \to \mathbb{N}$ tedy platí, že $T(n)$ je rovno dØlce výpočtu na *libovolnØ* vstupní posloupnosti dØlky n.

Je-li $n \leq 1$, pak $T(n) = a$, kde a je konstanta.

Je-li $n > 1$, pak $T(n) = b + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + D(n)$, kde D je složitost zbytku těla procedury `msort` (bez rekursivního volÆní), tedy zejmØna pomocnØ procedury `merge`.

Procedura `merge` je tvořena cyklem s pevným počtem opakovÆní (n), takže $D(n) = c \cdot n$, kde c je konstanta. Tedy $D \in \Theta(n)$.

Protože $\lfloor n/2 \rfloor$ a $\lceil n/2 \rceil$ se liší nejvýše o 1 (tj. o konstantu), můžeme tento rozdíl zanedbat. Takto upravenÆ funkce T' roste stejně rychle jako T:

$T'(1) = a$

$T'(n) = b + 2\,T'(\lceil n/2 \rceil) + n \cdot c, \quad T' \in \Theta(T).$

Pro několik hodnot n dostÆvÆme

$$
\begin{array}{lcll}
T'(1) & = & a & \\
T'(2) & = & b + 2 \cdot T'(1) + 2c & = & 2a + b + 2c \\
 & \vdots & & \\
T'(4) & = & b + 2 \cdot T'(2) + 4c & = & 4a + 3b + 8c \\
 & \vdots & & \\
T'(8) & = & b + 2 \cdot T'(4) + 8c & = & 8a + 7b + 24c \\
 & \vdots & & \\
T'(16) & = & b + 2 \cdot T'(8) + 16c & = & 16a + 15b + 64c \\
 & \vdots & & \\
T'(32) & = & b + 2 \cdot T'(16) + 32c & = & 32a + 31b + 160c \\
 & \vdots & &
\end{array}
$$

Všimneme-li si hodnot T' na mocninách dvou, můžeme vyslovit tvrzení

Věta: Pro každé $k \in \mathbb{N}$ je

$$T'(2^k) = 2^k a + (2^k - 1)b + 2^k kc$$

Vyjádřeno pomocí n:

$$T'(n) = a \cdot n + b \cdot (n - 1) + c \cdot n \log_2 n$$

Protože lineární funkce $a \cdot n$, $b \cdot n$ rostou nejvýše tak rychle jako $n \cdot \log n$, tak $T' \in \Theta(n \log n)$.

T roste stejně rychle, takže i $T \in \Theta(n \log n)$.

Rovnosti $T(1) = c$
$T(n) = 2\,T(\lceil n/2 \rceil) + d \cdot n$
jsou speciÆlním případem situace, v níž je
$T(1) = c$
$T(n) = a\,T(\lceil n/b \rceil) + f(n)$, kde $a \geq 1, b > 1, c > 0$ a f je kladnÆ funkce. Například pro algoritmus řazení slučovÆním je $a = 2, b = 2, f \in \Theta(n)$, tedy případ „2“ nÆsledující věty.

Věta: Nechť $a \geq 1, b > 1$, f je kladnÆ funkce a
$T(1) = c$
$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$
Potom:
1. Pokud $f \in O(n^{\log_b a - \varepsilon})$ pro nějakÉ $\varepsilon > 0$, pak $T \in \Theta(n^{\log_b a})$.
2. Pokud $f \in \Theta(n^{\log_b a})$, pak $T \in \Theta(n^{\log_b a} \log n)$.
3. Pokud $f \in \Omega(n^{\log_b a + \varepsilon})$ pro nějakÉ $\varepsilon > 0$ a pokud $a \cdot f(n/b) \leq d \cdot f(n)$ pro nějakÉ $d < 1$ a skoro všechna n, pak $T \in \Theta(f(n))$.

Dolní odhad složitosti řadicích algoritmů

Def: *Asociativní řadicí algoritmus* je algoritmus, jehož množinu přípustných vstupních dat tvoří všechny konečnØ posloupnosti celýchčísel, a takový, že jeho výsledkem je neklesající permutace vstupní posloupnosti.

PoznÆmka: Asociativní řadicí algoritmus tedy nemůže předpoklÆdat nic o velikosti řazených prvků a jedinou možností testovÆní je srovnÆvatprvky mezi sebou.

Věta: Každý sekvenční asociativní řadicí algoritmus mÆ složitost v$\Omega(n \log n)$.

PoznÆmka: Věta říkÆ, že*dolní odhad* (časovØ) složitosti problØmu asociativního řazení je v množině $\Omega(n \log n)$. Protože však znÆme konkrØtnířadicí algoritmy s touto složitostí, vidíme, že tento odhad je dokonce *těsný*: ČasovÆ složitost problØmu asociativního řazení je v množině $\Theta(n \log n)$.

Např. složitost řazení vkládáním, výběrem, bublinkami či rozdělováním je $\Theta(n^2) \subseteq \Omega(n \log n)$, složitost řazení slučováním či haldou je $\Theta(n \log n) \subseteq \Omega(n \log n)$.

Pro určení složitosti uvažujeme zápis algoritmu v jednoduchém funkcionálním nebo imperativním jazyce, tj. ekvivalentní *sekvenční* implementaci RAMem.

Idea důkazu Věty o dolním odhadu složitosti

Bez újmy na obecnosti můžeme předpokládat, že prvky uspořádané posloupnosti jsou navzájem různé: chceme-li nejhorší případ, musíme seřazovat co nejvíce prvků. Dokonce se můžeme omezit jen na permutace množiny $\{1, \ldots, n\}$: asociativní algoritmus si všímá jen srovnání dvojic prvků.

Poněvadž existuje celkem $n!$ možných permutací a pro každou takovou permutaci na vstupu musí podle toho algoritmu proběhnout jiný výpočet, existuje aspoň $n!$ různých výpočtů. Ve všech těchto výpočtech je aspoň $n! - 1$ binárních testů, jejichž výsledky odliší jednotlivé výpočty. Ale to znamená, že v *nejdelším* výpočtu (jde nám o nejhorší případ) musí být aspoň $\lfloor \log_2 n! \rfloor$ testů. Pro složitost T *každého* algoritmu tedy musí platit $T(n) \geq \lfloor \log_2 n! \rfloor$. To znamená, že $T \in \Omega(\log n!) = \Omega(n \log n)$

Cvičení: V důkazu věty o dolním odhadu složitosti řadicích algoritmů se mlčky předpokládá, že složitost jednoho porovnání dvou čísel a_i a a_j je konstantní. Můžete uvést lepší (realističtější) odhad časové složitosti jednoho porovnání?

Cvičení: Ukažte, že $\log\big(n \cdot \log n\big) \in \Theta\big(\log n\big)$.

Cvičení: Uvážíme-li realistickou složitost jednoho porovnání, jak vyjde dolní odhad složitosti řadicích algoritmů? Je v rozporu s dokázanou větou? Je jejím zesílením?

Algoritmus řazení rozdělovÆním

(funkcionÆlní verze)

```
quickSort :: [Int] → [Int]
quickSort []     =  []
quickSort (p:t) =  quickSort lt ++ [p] ++ quickSort ge
                   where lt = [ x | x←t, x<p ]
                         ge = [ x | x←t, x≥p ]
```

Věta: Nechť *In* = *Out* je množina všech seznamů (posloupností) celých čísel,

$\varphi(s) \equiv$ „s je konečný“,

$\psi(s,t) \equiv$ „t je permutací seznamu s a t je neklesající“.

Pak `quickSort` je totÆlně korektní vzhledem k podmínkÆm φ, ψ.

Věta: ČasovÆ složitost algoritmu `quickSort` je v $\Theta(n^2)$.

Důkaz korektnosti: Konvergence i parciÆlní korektnost seukÆže indukcí vzhledem k dØlce seznamu s.

Řazení rozdělovÆním (imperativní verze)

```
type  Element = Integer;
      Posl = array [1..MAX] of Element;

procedure QuickSort (n:Integer, var A:Posl);

  procedure Q (l,r:Integer);
    var p: Integer;
    begin if l<r then begin p := Part (l,r);
                            Q (l,p-1);  Q (p+1,r)
                      end
    end;

  procedure Part (l,r:Integer);
    var i,j: Integer; x,y: Element;
    begin x := A[r];  i := l - 1;
          for j:=l to r-1 do
              if A[j]≤x then begin i := i+1;
                                   y := A[i];  A[i] := A[j];  A[j] := y;
                             end;
          y := A[i+1];  A[i+1] := A[r];  A[r] := y
          Part := i + 1
    end;

begin  Q (1,n)  end
```

Řazení rozdělovÆním (modif. imp. verze)

```
type  Element = Integer;
      Posl = array [1..MAX] of Element;

procedure QuickSort (n:Integer, var A:Posl);

  procedure Q (l,r:Integer);
    var p: Integer;
    begin if l<r then begin p := Part (l,r);
                            if p-l<r-p then begin Q (l,p-1);  Q (p+1,r) end
                                       else begin Q (p+1,r);  Q (l,p-1) end
                      end
    end;

  procedure Part (l,r:Integer);
    var i,j: Integer; x,y: Element;
    begin x := A[r];  i := l - 1;
          for j:=l to r-1 do
              if A[j]≤x then begin i := i+1;
                                   y := A[i];  A[i] := A[j];  A[j] := y;
                             end;
          y := A[i+1];  A[i+1] := A[r];  A[r] := y
          Part := i + 1
    end;

begin  Q (1,n)  end
```

Věta: Algoritmus řazení rozdělovÆním mÆ složitost v množině $\Theta(n^2)$.

PoznÆmka: *PrůměrnÆ dØlka* výpočtu podle algoritmu řazení rozdělovÆním (tzv. *průměrnÆ* nebo *očekÆvanÆ časovÆ složitost*) je v množině $\Theta(n \log n)$.

Důkaz věty o kvadratickØ složitosti: Není tĕžkØ zjistit, že dØlka výpočtu cyklu (repeat) je $c \cdot n$ (celØ pole se projde jedním průchodem a výměna prvků trvÆ konstantní dobu).

Označíme-li $T(n)$ složitost seřazení pole dØlky n, dostaneme

$$T(n) = c \cdot n + \max\{T(q) + T(n-q-1) \mid 0 \leq q < n\}$$

Při nevyvÆženØm dĕlení posloupnosti na podposloupnosti dØlek $n-1, 0$ je dØlka výpočtu kvadratickÆ. Tedy $T \in \Omega(n^2)$. Ale výraz $T(q) + T(n-q-1)$ nabude maxima pro $q = 0$ nebo $q = n-1$ (o tomto případu víme, že je kvadratický) – druhÆ derivace $T(n)$ vzhledem ke q pro kvadratický odhad $T(n) \leq cn^2$ je kladnÆ.
Dohromady, $T \in \Omega(n^2)$, $T \in O(n^2)$, tj. $T \in \Theta(n^2)$.

Neformální zdůvodnění poznámky o průměrné složitosti:
Při každém vyvolání procedury Q se posloupnost délky n rozdělí na dva úseky délek q a $n - q - 1$. Předpokládáme-li rovnoměrné rozložení čísel (položek) v seřazované posloupnosti, pak pravděpodobnost, že $q < rn$, je rovna číslu r, $0 \leq r \leq 1$. Tedy pravděpodobnost, že se při výpočtu „nepříznivá“ hodnota hranice q vybere ve významném počtu případů, klesá s číslem r k nule. Proto stačí uvažovat jen případy, kdy se posloupnost dělí na úseky délek větších než n/r_0 pro nějaké pevné $r_0 > 0$ (ostatní případy mají nevýznamnou pravděpodobnost). Ale pak je největší hloubka zanoření rovna $\log_{\frac{1}{r_0}} n = -\log_{r_0} n$. Součet délek seřazovaných úseků v každé hloubce je nejvýše n, takže *průměrná* délka výpočtu je nejvýše $-c \cdot n \cdot \log_{r_0} n$, tedy v $O(n \log n)$. Dále se snadno ověří (například pro $r_0 = 1/2$), že $n \log n$ je i dolním odhadem, takže průměrná délka výpočtu je v $\Theta(n \log n)$.

ProstorovÆ složitostřadicích algoritmů

Definujeme *prostorovou složitost* algoritmu analogicky jako časovou složitost, ale za míru složitosti místo počtu kroků výpočtu zvolíme maximÆlní množství paměti, kterÆ bude během výpočtu obsazena.

Def: ProstorovÆ složitost algoritmu A je funkce, kterÆ pro každou velikost vstupních dat je rovna velikosti obsazenØ paměti při výpočtu, jenž tØto paměti spotřebuje nejvíc.

PoznÆmka: Označíme-li $\rho(A, \sigma_x)$ množství paměti spotřebovanØ výpočtem podle algoritmu A z počÆtečního stavu σ_x (odpovídajícího umístění dat x na vstupu), potom prostorovÆ složitost algoritmu A je

$$S_A(n) = \max\{\rho(A, \sigma_x) \mid |x| = n\}$$

Prostorová složitost všech zmiňovaných řadicích algoritmů je lineární, tj. $S \in \Theta(n)$. Důvod je ten, že součástí dat, s nimiž algoritmus pracuje, je samařazená posloupnost, a ta má délku n. Ostatní data (mimo tuto posloupnost, tj. „extrasekvenční“) mají v uvážených algoritmech velikost nejvýše $O(n)$.

Def: Zavedeme *extrasekvenční* prostorovou složitost řadicího algoritmu jako funkci zobrazující dØlky vstupní posloupnosti na velikost paměti obsazenØ při výpočtu v nejhorším případě, ale *nezapočítÆvÆme paměť obsazenou seřazovanou posloupností*.

Řadicí algoritmy, kterØ mají extrasekvenční prostorovou složitost konstantní, se nazývají *in situ*.

Řazení rozdělovÆním není *in situ*.

Věta: První varianta algoritmu `quickSort` mÆ extrasekvenční prostorovou složitost $\Theta(n)$, druhÆ (modifikovanÆ) varianta tohoto algoritmu mÆ extrasekvenční prostorovou složitost $\Theta(\log n)$.

PoznÆmka: Hovořit o řadicích algoritmech, zda jsou in situ, a zavÆdět jejich extrasekvenční prostorovou složitost mÆ praktický význam jen tehdy, je-li posloupnost reprezentovÆna polem a uložena v souvislØm œseku paměti. Při jiných datových reprezentacích posloupnosti (např. seznamem) se extrasekvenční prostorovÆ složitost nezavÆdí.

Řazení haldou

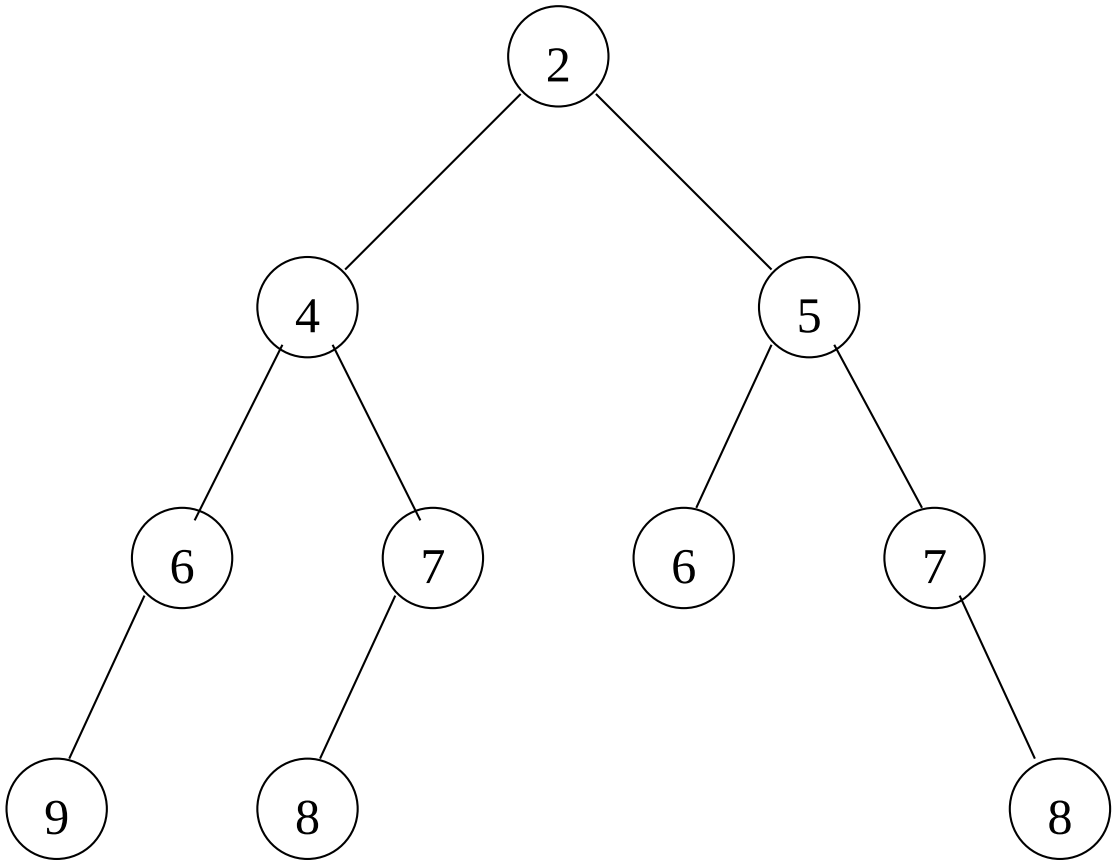
Def: Nechť K je úplně uspořádaná množina tzv. *klíčů* (typicky množina čísel s přirozeným uspořádáním $\leq$). *Binární halda* je binární strom, jehož uzly jsou ohodnoceny prvky množiny K, a který splňuje tyto vlastnosti:

1. Délky všech větví se liší nejvýše o 1: mají délku k, případně $k-1$. Číslu k pak říkáme *hloubka* haldy.
2. Hodnoty uzlů na každé větvi jsou vzestupně (sestupně) uspořádány.

Jsou-li hodnoty uzlů na každé větvi seřazeny vzestupně (čím dále od kořene, tím větší hodnoty), je v kořenu haldy nejmenší prvek. Hovoříme pak o *minimové* haldě (*min-heap*).

Jsou-li hodnoty uzlů na každé větvi seřazeny sestupně (čím dále od kořene, tím menší hodnoty), je v kořenu haldy největší prvek. Hovoříme pak o *maximové* haldě (*max-heap*).

Ve funkcionÆlní implementaci algoritmuřazení haldou, kde se pracuje s explicitní haldou (datovou strukturou binÆrní strom), použijeme minimovou haldu. Naopak v imperativní implementaci, kde se pracuje s implicitní haldou (reprezentovanou polem), je výhodnější použít maximovou haldu.


2
4
5
6
7
6
7
9
8
8

Operace s binÆrní haldou

```
data Heap = Empty | Node Elem Heap Heap
```

ZÆkladní operace – konstruktory a selektory – mají konstantní složitost.

Konstruktory

$\texttt{Empty} : \texttt{Heap}$

$\texttt{Node} : \texttt{Elem} \to \texttt{Heap} \to \texttt{Heap} \to \texttt{Heap}$

Selektory

$\texttt{rootVal} : \texttt{Heap} \dashrightarrow \texttt{Elem}$

$\texttt{leftHeap} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

$\texttt{rightHeap} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

$\texttt{isEmpty} : \texttt{Heap} \to \texttt{Bool}$

Další operace pro prÆci s binÆrní haldou

VyhledÆní minimÆlního prvku (v minimovØ haldØ)

$\texttt{minH} : \texttt{Heap} \dashrightarrow \texttt{Elem}$

$\texttt{minH} = \texttt{rootVal}$

(složitost $\Theta(1)$)

Odstranění minimÆlního prvku (z minimovØ haldy)

$\texttt{extractMinH} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

(složitost $\Theta(\log n)$)

Odstranění maximÆlního prvku (z maximovØ haldy)

$\texttt{extractMaxH} : \texttt{Heap} \dashrightarrow \texttt{Heap}$

(složitost $\Theta(\log n)$)

Přidání prvku

$\texttt{insertH} : \texttt{Elem} \to \texttt{Heap} \to \texttt{Heap}$

(složitost $\Theta(\log n)$)

Odstranění prvku

$\texttt{removeH} : \texttt{Heap} \to \texttt{Heap} \dashrightarrow \texttt{Heap}$

(složitost $\Theta(\log n)$)

Věta: NeprÆzdnÆ binÆrní halda s n uzlech mÆ hloubku $\lfloor \log_2 n \rfloor$.

Důkaz indukcí podle n.

Def: BinÆrní strom, v jehož každØm podstromu se počet uzlů levØho a pravØho podstromu liší nejvýše o jednu, se nazývÆ *uzlově vyvÆžený* binÆrní strom.

Věta: Každý uzlově vyvÆžený binÆrní strom je haldou (až na ohodnocení uzlů).

Důkaz: Je třeba dokÆzat, že dØlky všech větví uzlově vyvÆženØho stromu se liší nejvýše o jednu.

PředpoklÆdejme sporem, že tvrzení neplatí, a T je nejmenší strom, který tvrzení porušuje. Tedy v každØm podstromu stromu T se počet uzlů vlevo a vpravo liší nejvýše o jednu, ale existují zde dvě větve dØlek m a k, přičemž $m \geq k + 2$. Z minimality stromu T plyne, že jedna z těchto dvou větví, řekněme ta delší, dØlky m, leží v levØm podstromu T_l stromu T, a druhÆ, kratší, dØlky k, v pravØm podstromu T_r stromu T. Navíc, rovněž z minimality, všechny větve jdoucí do T_l mají dØlku m nebo $m - 1$, všechny větve jdoucí do T_r mají dØlku k nebo $k + 1$. To znamenÆ, že T_l mÆ aspoň o dva uzly víc než T_r. Ale podle předpokladu věty mÆ nejvýše o jeden uzel více, což je spor.

Algoritmus řazení binÆrní haldou (funkcionÆní verze)

```
data Heap  =  Empty  |  Node Int Heap Heap

insHeap :: Int → Heap → Heap
insHeap u Empty          =  Node u Empty Empty
insHeap u (Node v p q)   =  if u ≥ v  then  Node v (insHeap u q) p
                                      else  Node u (insHeap v q) p

toHeap :: [Int] → Heap
toHeap s = foldr insHeap Empty s

toList :: Heap → [Int]
toList Empty = []
toList (Node x l r) = x : merge (toList l) (toList r)

heapSort :: [Int] → [Int]
heapSort  =  toList · toHeap
```

Korektnost řazení haldou

Věta: Algoritmus `heapSort` je totÆlně korektní řadicí algoritmus.

Konvergence algoritmu je zřejmÆ. Důkaz parciÆlní korektnosti vyplývÆ z nÆsledujících lemmat.

Lemma: Funkce `toList` vytvoří z haldy h seznam s týmiž prvky jako měla halda h, ale uspořÆdaný vzestupně.

Lemma: Je-li u číslo a h uzlově vyvÆženÆ halda, pak `insHeap` u h je halda obsahující prÆvě prvky z haldy h a prvek u.

Lemma: Funkce `toHeap` vytvoří ze seznamu s haldu obsahující tytØž prvky jako seznam s.

Důkaz prvního lemmatu indukcí podle velikosti haldy.

Důkaz druhØho lemmatu: indukcí podle počtu prvků v h se ukÆže, že binÆrní strom `insHeap` u h bude opět uzlově vyvÆžený.
Podle věty o binÆrních stromech vyvÆžených vzhledem k počtu uzlů bude výsledek opět halda.
UsporÆdÆní hodnot podØétví vyplývÆ z definice funkce `insHeap`.

Třetí lemma indukcí podle dØlky seznamu a z definice kombinÆtoru `foldr`.

$$\texttt{foldr insHeap Empty [}x_1\texttt{,}x_2\texttt{,...,}x_n\texttt{]}$$
$$= \texttt{insHeap } x_1 \ (\texttt{insHeap } x_2 \ (\ldots(\texttt{insHeap } x_n \texttt{ Empty})\ldots))$$

Složitost řazení haldou

Lemma: Procedura `insHeap` mÆ složitost v $O(\log n)$.

Důkaz: Funkce `insHeap` se rekursivně vyhodnotí prÆvě tolikrÆt, jako je hloubka haldy. Ale ta je logaritmickÆ vzhledem k počtu jejích uzlů, jichž je vždy nejvýše n.

Věta: Algoritmus řazení haldou mÆ složitost v $\Theta(n \log n)$.

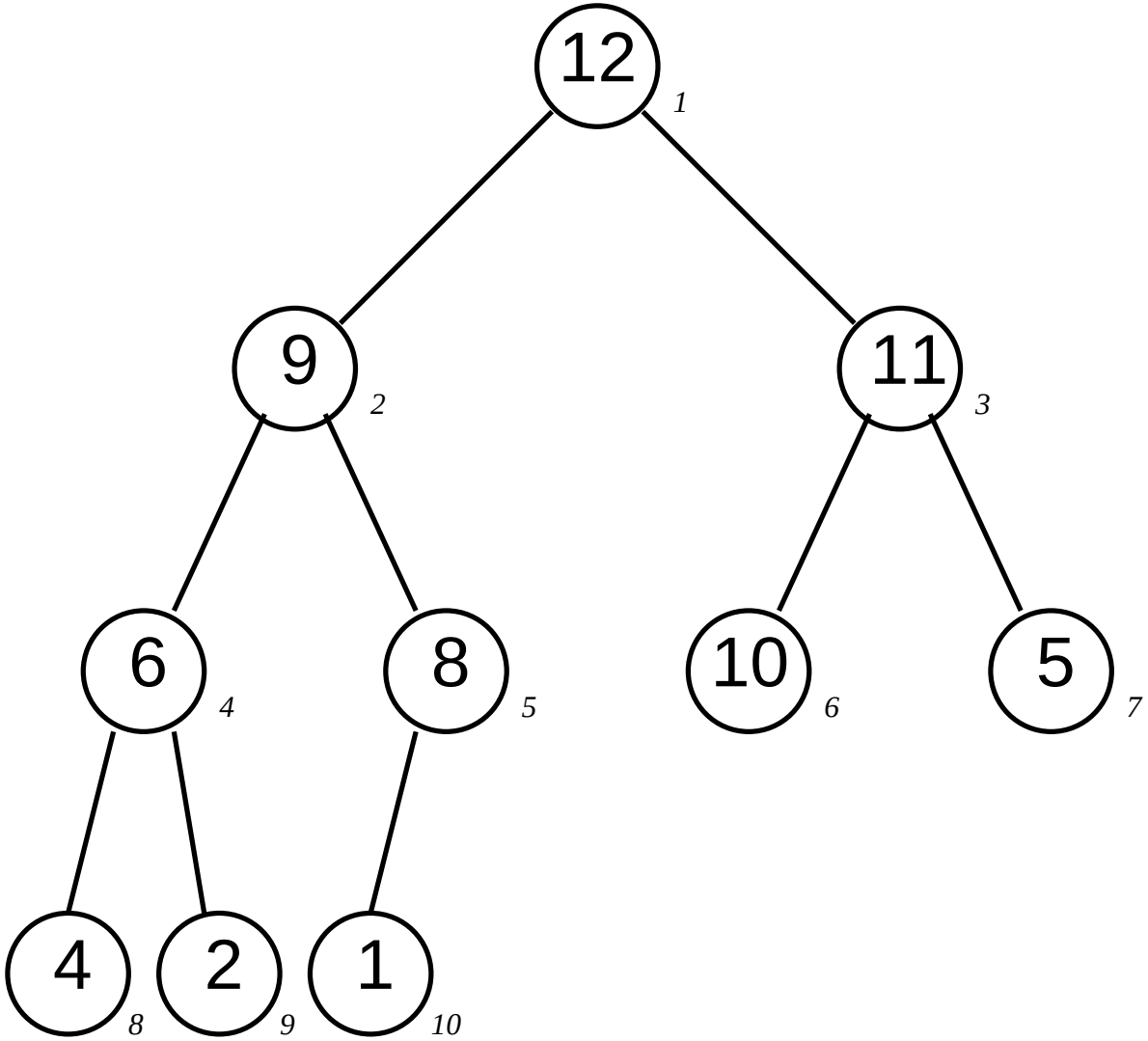
Důkaz: Z předchozího lemmatu vyplývÆ, že složitost funkce `toHeap` je v $O(n \log n)$. Ve funkci `toList` je hloubka rekursivního zanoření logaritmickÆ, protože seznamy se půlí. Složitost pomocnØ funkce `merge` je lineÆrní a celkovÆ dØlka výstupu všech volÆní funkce `merge` ve stejnØ hloubce k (tj. se stejně dlouhými seznamy) je $\frac{n}{2^k} \cdot 2^k = n$. Funkce `toList` mÆ tedy složitost takØ v $O(n \log n)$. Celkem je tedy horní odhad složitosti funkce `heapSort` $n \log n$, ale podle Věty o dolním odhadu složitosti řadicích je toto i dolním odhadem.

V imperativním algoritmu `heapSort` se používÆ maximovÆ halda, tj. binÆrní halda, kterÆ mÆ položky ve vìtvích seřazeny *sestupnì*. Tedy největší položka je vždy v kořeni.

Jelikož algoritmus mÆ haldu efektivnì reprezentovÆnu polem, je zde výhodnìjší uvažovat binÆrní haldu nikoliv uzlovì vyvÆženou, ale naopak s levým podstromem „tìžším“.

Def: *Vlevo zarovnanÆ halda* je binÆrní halda, jejíž všechny vìtve dØlky k leží nalevo od vìtví dØlky $k-1$.

Vlevo zarovnanÆ halda



Věta: Reprezentujme vlevo zarovnanou binÆrní haldu na n uzlech posloupností hodnot jejích uzlů $(a_1, \ldots, a_n)$ takto: a_1 reprezentuje kořen haldy a je-li uzel u reprezentovÆn prvkem a_i, pak levý nÆsledník uzlu u je reprezentovÆn prvkem a_{2i} a pravý nÆsledník uzlu u je reprezentovÆn prvkem a_{2i+1}. Pak posloupnost $(a_1, \ldots, a_n)$ je haldou určena jednoznačně.

Věta umožňuje pracovat s vlevo zarovnanou binÆrní haldou reprezentovanou v paměti polem. Vlevo zarovnanÆ binÆrní halda je totØž jako pole rozepsanØ „po vrstvÆch“.

Algoritmus řazení binÆrní haldou (imp. verze)

```
type  Element = Int;
      Posl = array [1..MAX] of Element;

procedure heapSort (n:Int, var A:Posl);
  var l,r: Int;  x: Element;

  procedure createHeap (l,r:Int);
    { vytvoření haldy A[l]...A[r] ze dvou podhald začínajících A[2*l] a A[2*l+1] }
    var i,j:Int; {pom. indexy}  y:Element; {zařazovaný prvek}  p:Bool; {sestupovat?}
    begin i := l;  j := 2*i;  y := A[i];  p := True;
          while  p  &&  (j≤r) do  { sestupujeme }
            begin if (j<r) && (A[j]<A[j+1]) then j:=j+1;
                  { A[j] je teď větší z kořenů podhald pod A[i] }
                  p := y<A[j];
                  if p then { A[j]>y, takže posuneme A[j] o patro výš }
                            begin
                              A[i] := A[j];
                              i := j;  j := 2*i
                            end
            end;
          A[i] := y  { prvek y zařazen na své místo }
    end {createHeap};
```

```
begin {heapSort}
   { postupně vytvoříme haldu A[1],...,A[n] }
        for l := n div 2 downto 1 do  createHeap(l,n);

   { V kořenu (A[1]) je prvek s největším ohodnocením.}
   { Vyměníme ho s koncem pole, haldu zkrátíme        }
   { a restaurujeme zařazením A[1] na správné místo   }
        for r := n downto 2 do
         begin
            x := A[1];  A[1] := A[r];  A[r] := x;
            createHeap (1,r-1)
         end
end {heapSort};
```

Korektnost a složitost imperativního algoritmu `heapSort`

Věta: Mějme vlevo zarovnanou binÆrní haldu reprezentovanou posloupností $(a_l, \ldots, a_r)$ a nechť oba podstromy pod kořenem splňují podmínky haldy. Pak posloupnost $(a'_l, \ldots, a'_r)$ ve stavu po provedení `createHeap`(a, l, r) splňuje podmínky (vlevo zarovnanØ) binÆrní haldy.

Důkaz indukcí podle hloubky haldy.

Věta: Provedení procedury `heapSort`(A) seřadí posloupnost A

Idea důkazu: JednoprvkovØ podstromy reprezentovanØ druhou polovinou posloupnosti A jsou triviÆlní podhaldy. První fÆze algoritmu vytvÁÆ z těchto podhald větší podhaldy. Na konci první fÆze je posloupnost$A$ binÆrní haldou.

DruhÆ fÆze algoritmu vytvÁÆ z haldy seřazenou posloupnost. To vyplývÆ z toho, že největší prvek haldy je vždy v jejím kořenu: největší prvky se sklÆdají na konec posloupnosti.

Věta: Algoritmus řazení haldou konverguje.

Důkaz: Tvrzení věty vyplývÆ z konečnØ velikosti a hloubky haldy.

Věta: Řazení haldou (imperativní algoritmus) mÆ složitost v $\Theta(n \log n)$.

Důkaz: První fÆze (vytvÆření haldy) mÆ složitost $O\left(\frac{n}{2} \log n\right)$, druhÆ fÆze mÆ složitost $O(n \log n)$, obě tedy dohromady $O(n \log n)$.

Podle Věty o dolním odhadu složitosti musí být rovněž v $\Omega(n \log n)$.

Z obou odhadů pak plyne tvrzení.

DatovØ struktury

Typy rozlišujeme *datovØ* a *funkcionÆlní*.

DatovØ typy lze zavØst pomocí funkcionÆlních, případně funkcionÆlní typy lze simulovat pomocí nekonečných datových typů, ale kvůli efektivnosti implementace se rozlišují a uvažují se zvlÆšť.

Hodnoty funkcionÆlních typů jsou *funkce* a *procedury*. JednotlivØ funkční hodnoty (či vedlejší efekty procedur) nejsou znÆmy předem, ale dospěje se k nim po výpočtu. Ten je spuštěn tzv. *volÆním* čili *aplikací na argumenty*.

Hodnoty datových typů jsou obvykle znÆmy a uloženy v paměti, ale jejich složky je třeba najít a zpřístupnit (například najít prvek pole podle indexu apod).

Datové typy mohou být *skalární* nebo *složené*.

Skalární datové typy v daném programovacím jazyce obvykle zahrnují číselné typy (celé nebo desetinné čísla z určité konečné množiny), znakové typy, typ pravdivostních hodnot apod. Data skalárního typu zabírají vždy konstantní a malé množství paměti (typicky do 10 B). Zpřístupnění hodnoty skalárního typu trvá konstantní dobu.

Složené datové typy jsou například záznamy (n-tice s pojmenovanými složkami), uniony, posloupnosti, množiny apod.
Data složených typů se nazývají *datové struktury*.

- Datové struktury pevné velikosti (n-tice, statické pole, ...) — tzv *statické datové struktury*
- Datové struktury proměnné velikosti — tzv. *dynamické datové struktury*

StatickØ datovØ struktury mají konstantní velikost a časovÆ složitost zpřístupnění libovolnØho prvku je konstantní.

DynamickØ datovØ struktury mají velikost danou nějakou proměnnou n, jejíž hodnota se může měnit. ČasovÆ složitost zpřístupnění libovolnØho prvku dynamickØ datovØ struktury je neklesající funkcí zÆvislou na n; často lineÆrní, u efektivních datových struktur lepší, typicky logaritmickÆ.

DynamickØ datovØ struktury

Seznam

Seznam nad bÆzovým typem B je lineÆrní datovÆ struktura typu S s nÆsledujícími operacemi:

$$\texttt{nil} : S$$

$$\texttt{cons} : B \times S \to S$$

$$\texttt{head} : S \dashrightarrow B$$

$$\texttt{tail} : S \dashrightarrow S$$

$$\texttt{null} : S \to \texttt{Bool}$$

Pro tyto operace a každou hodnotu x typu B a každý seznam s typu S musí platit

Axiomy seznamu

```
null(nil)  =  True
null(cons(x,s))  =  False
head(cons(x,s))  =  x
tail(cons(x,s))  =  s
```

ZÆsobník

ZÆsobník nad bÆzovým typem B je lineÆrní datovÆ struktura typu S s nÆsledujícími operacemi:

$$\texttt{empty} : S$$

$$\texttt{push} : B \times S \to S$$

$$\texttt{top} : S \dashrightarrow B$$

$$\texttt{pop} : S \dashrightarrow S$$

$$\texttt{isempty} : S \to \texttt{Bool}$$

Pro tyto operace a každou hodnotu x typu $\mathbb{B}$ a každý zÆsobníks typu $\mathbb{S}$ musí platit tzv.

Axiomy zÆsobníku

$$\texttt{isempty(empty)} = \texttt{True}$$

$$\texttt{isempty(push(}x\texttt{,}s\texttt{))} = \texttt{False}$$

$$\texttt{top(push(}x\texttt{,}s\texttt{))} = x$$

$$\texttt{pop(push(}x\texttt{,}s\texttt{))} = s$$

Jak je vidět, zÆsobník a seznam je tatÆž datovÆ struktura.

Liší se pouze v použití:

O *zÆsobníku* obvykle mluvíme, když na něm používÆme jen zÆkladní operace a pracujeme jen s jedním zÆsobníkem nebo pevně daným malým počtem zÆsobníků. ZÆsobníky bývají dÆny předem: během výpočtu se mění jejich obsah, ale nemění se počet zÆsobníků, tj. neruší se a nevznikají nové zÆsobníky. Proto se v imperativních jazycích zÆkladní operace často implementují jako procedury; navíc jejich parametr zÆsobník se vynechÆvÆ, pracuje-li se jen s jedním zÆsobníkem.

U *seznamů* často definujeme složitější operace (zpřístupnění či změnu n-tØho prvku, rozdělení seznamu na dva, spojení dvou seznamů,...) a během výpočtu seznamy vytvÆříme a rušíme.

Například změnu třetího prvku (alespoň tříprvkovØho) seznamu s na číslo 5 lze realizovat složenou operací $\texttt{cons}(a,\texttt{cons}(b,\texttt{cons}(5,t)))$, kde $a = \texttt{head}(s)$, $b = \texttt{head}(\texttt{tail}(s))$, $t = \texttt{tail}(\texttt{tail}(\texttt{tail}(s)))$.

Fronta

Fronta nad bÆzovým typem B je lineÆrní datovÆ struktura typu Q s nÆsledujícími operacemi:

$$\texttt{empty} : Q$$

$$\texttt{head} : Q \dashrightarrow B$$

$$\texttt{enqueue} : B \times Q \to Q$$

$$\texttt{dequeue} : Q \dashrightarrow Q$$

$$\texttt{isempty} : Q \to \texttt{Bool}$$

Pro tyto operace a každØ hodnoty x, y typu B a každou frontu q typu Q musí platit

Axiomy fronty

$$\texttt{isempty(empty)} = \texttt{True}$$

$$\texttt{isempty(enqueue(}x\texttt{,}q\texttt{))} = \texttt{False}$$

$$\texttt{head(enqueue(}x\texttt{,empty))} = x$$

$$\texttt{head(enqueue(}x\texttt{,enqueue(}y\texttt{,}q\texttt{)))} = \texttt{head(enqueue(}y\texttt{,}q\texttt{))}$$

$$\texttt{dequeue(enqueue(}x\texttt{,empty))} = \texttt{empty}$$

$$\texttt{dequeue(enqueue(}x\texttt{,enqueue(}y\texttt{,}q\texttt{)))} = \texttt{enqueue(}x\texttt{,dequeue(enqueue(}y\texttt{,}q\texttt{)))}$$

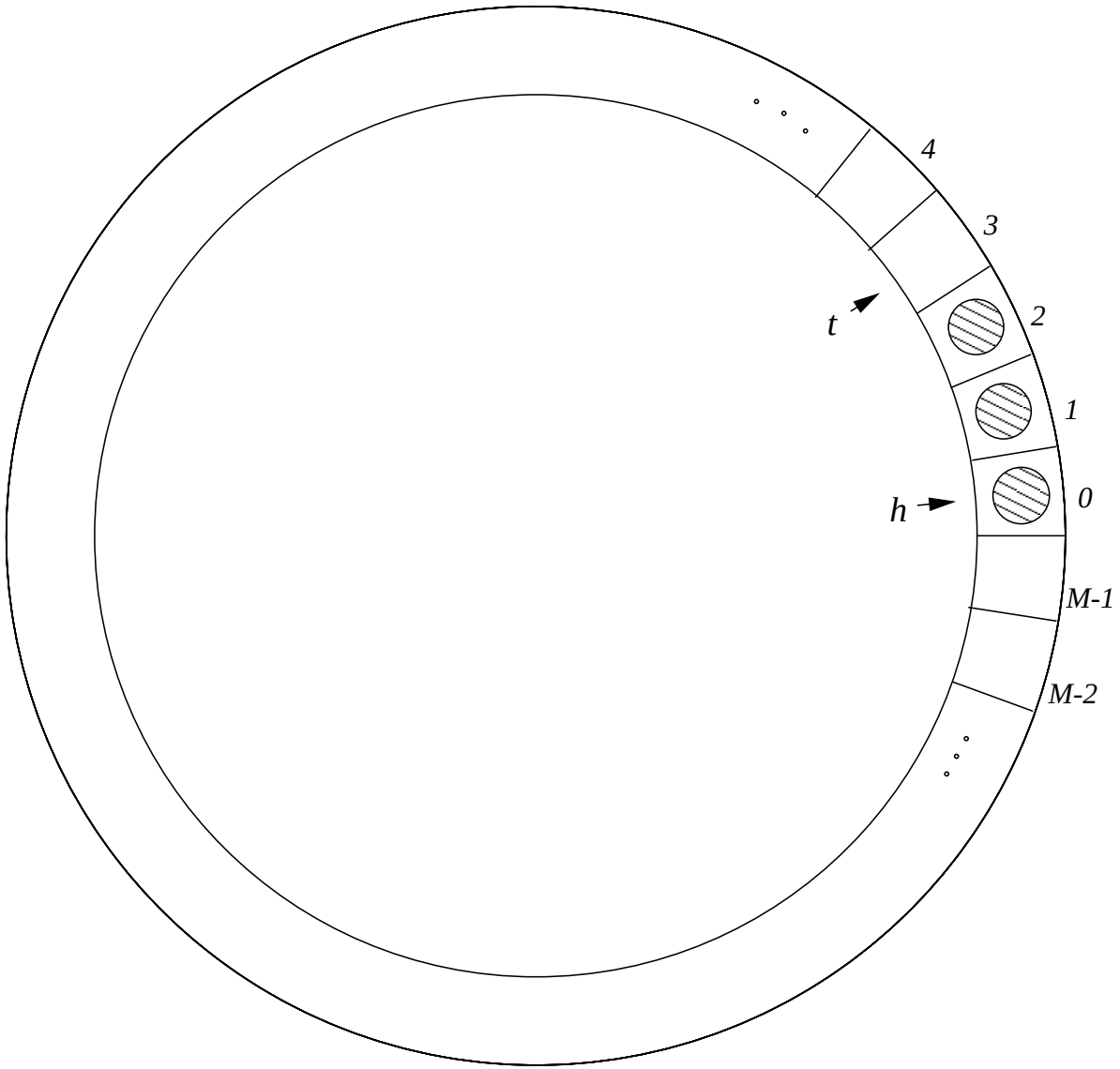
Implementace datových struktur

Jestliže typ datovÉ struktury není součÁstí jazyka, je nutnÉ vyjÁdřit datovou strukturu a operace nad ní pomocí jiných struktur a operací. TakovÉ kolekci definic říkÁme *implementace datovÉ struktury*.

Například zÁsobník (omezenÉ dÉlky) lze implementovat pomocí (statickÉho) pole,
frontu lze implementovat pomocí dvou zÁsobníků,
frontu omezenÉ dÉlky pomocí cyklickÉho pole apod.

PoznÆmka: Jazyky s malou podporou dynamických datových struktur (Pascal, C,...) často poskytují alespoň dynamickou datovou strukturu „nízkØ œrovnì“: *paměť* `M` spolu s typem *ukazatel* (adresa) `P`, nulÆrní operací `nil:P` prÆzdnØho ukazatele a unÆrní operací zpřístupnění (dereferencovÆní) čÆsti paměti `deref:P-->M`.

Pak například seznam se běžně implementuje pomocí zÆznamů v `M`, jejichž složky jsou ukazatele na další zÆznamy, apod.


4
3
2
1
0
M-1
M-2
t
h

Příklad: Implementace ohraničenØ fronty pomocí cyklickØho pole

```
{ procedury pracují pouze s jedinou globální frontou }

const M   = 256;  { délka pole }
      MAX = M-1;  { kapacita fronty }
type R = 0..MAX;
     Elem = Integer;
var Q : record
          a : array [R] of Elem;
          h, t : Integer
        end;
```

```
procedure mkemptyq;
  begin
    Q.h := 0;  Q.t := 0
  end;

function isempty : Boolean;
  begin
    isempty := Q.h = Q.t
  end;

function isfull : Boolean;
  begin
    isfull := Q.h = (Q.t + 1) mod M
  end;
```

```
function headq : Elem;
  begin
    if isemptyq then err("headq prázdné fronty")
                else headq := Q.a[Q.h]
  end;

procedure enqueue (x:Elem);
  begin
    if isfull
       then err("enqueue do plné fronty")
       else begin Q.a[Q.t] := x;
                  Q.t := (Q.t + 1) mod M
            end
  end;
```

```
procedure dequeue;
  begin
    if isemptyq then err("dequeue prázdné fronty")
                else Q.h := (Q.h + 1) mod M
  end;
```

Příklad: Implementace fronty pomocí dvou seznamů

```
data  Queue  =  Q [Int] [Int]

emptyq ::  Queue
emptyq  =  Q [] []

isemptyq  ::  Queue → Bool
isemptyq (Q [] [])  =  True
isemptyq _          =  False

enqueue :: Int → Queue → Queue
enqueue x (Q h t)  =  Q h (x:t)
```

```
headq :: Queue → Int
headq (Q (x:_) _)  =  x
headq q            =  head h
                      where  Q h _ = revq q

dequeue :: Queue → Queue
dequeue (Q (_:h) t)  =  Q h t
dequeue q            =  Q u []
                        where  Q (_:u) [] = revq q

revq (Q [] t)  =  Q (reverse t) []
```

Cvičení: Funkce `headq` a `dequeue` z předešlé strany zůstávají nedefinovány pro aplikaci na prázdnou frontu. Doplňte jejich definice tak, aby jejich aplikace na prázdnou frontu způsobila chybové hlášení. Využijte haskellovské funkce `error :: String -> a`.

Dá se lehce spočítat, že časové složitosti operací `isemptyq` a `enqueue` z předchozího příkladu jsou konstantní, zatímco obě operace `headq` a `dequeue` jsou lineární vzhledem k velikosti fronty. V nepříznivém případě je totiž nutné volat pomocnou operaci `revq`, která je sama lineární.

Přesto i na tyto „dražší“ operace lze v kontextu programu, v němž jsou použity, pohlížet v jistém smyslu jako na operace v průměrném (očekávaném) případě konstantní.

AmortizovanÆ časovÆ složitost

Def: Nechť f je operace na danØ datovØ struktuře D velikosti (nejvýše) n. Uvažujme všechny výpočty $C_1, \ldots, C_m$ podle algoritmů pracujících s datovou strukturou D. Z každØho takovØho výpočtu vybereme všechna volÆní operace f (tj. všechny aplikace f na D), čímž dostaneme m posloupností volÆní operace f. Průměrnou dØlku výpočtu operace f v i-tØm výpočtu označíme τ_i^{av}. Potom *amortizovanÆ složitost* operace f na datovØ struktuře D je funkce $T^{\text{amort}} : \mathbb{N} \to \mathbb{N}$ definovanÆ pro každou velikost dat n takto

$$T^{\text{amort}}(n) = \max\{\tau_i^{\text{av}} | 1 \leq i \leq m\}$$

Příklad: Amortizovaná složitost operací `headq` a `dequeue` z příkladu implementace fronty dvěma seznamy jsou konstantní.

Každá „drahá“ volání operace `dequeue` nebo `headq` se totiž „rozpustí“ v následujících aspoň n „levných“ voláních těchto operací.

Poznámka: Je-li časová složitost (tj. složitost v nejhorším případě) nějaká operace v $O(f)$, pak také její amortizovaná složitost je v $O(f)$.

Poznámka: Jestliže má nějaká operace amortizovanou složitost $\Theta(g)$, může to být méně příznivé, než když má složitost $\Theta(g)$, protože drahá volání se *může* vyskytnout. Na druhé straně je to příznivější než *průměrná* složitost $\Theta(g)$, protože amortizovaná složitost dává horní ohraničení pro průměrnou složitost v rámci jediného výpočtu, a to pro *každý* výpočet.

Stromy

Obecně *strom* je souvislý graf bez kružnic. Nejčastěji pracujeme s tzv. *kořenovými stromy*, tj. stromy, v nichž je jeden vyznačený uzel, *kořen*, a hrany jsou implicitně orientovÆny směrem od kořene k listům.

Je-li u uzel stromu a do uzlů $u_1, \ldots, u_k$ vedou z uzlu u hrany, pak uzly $u_1, \ldots, u_k$ nazývÆme *(bezprostředními) nÆsledníky* uzlu u.

Na bezprostředních nÆslednících každØho uzlu často bývÆ zavedeno œplnØ uspoøÆdÆní. Pak se nÆsledníci uzlu znÆzorňují zleva doprava; u binÆrních stromů se tedy rozlišuje *levý* a *pravý* nÆsledník.

Uzly bez nÆsledníků se nazývají *listy*, ostatní uzly stromu jsou *vnitřní*.

Stromy pevnØ arity (s omezeným větvením)

Def: Je dÆno přirozenØ číslo n a tzv. bÆzovÆ množina (typ) B. Definujeme *n-Ærní strom* nad B takto:

- PrÆzdný strom $\emptyset$ je n-Ærní strom.
- Jsou-li $T_1, \ldots, T_n$ n-Ærní stromy a $b \in B$, pak $(n+1)$-tice $(b, T_1, \ldots, T_n)$ je n-Ærní strom.

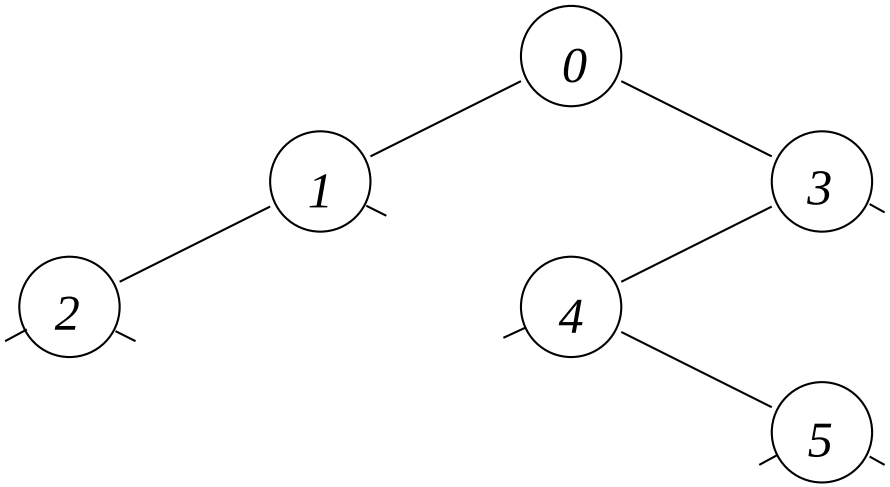
Listy n-Ærního stromu jsou uzly tvaru $(x, \emptyset, \ldots, \emptyset)$

Def: Cestu z kořene stromu do uzlu, jehož aspoň jeden bezprostřední nÆsledník je prÆzdný strom, se nazývÆ *Ævětev* stromu.

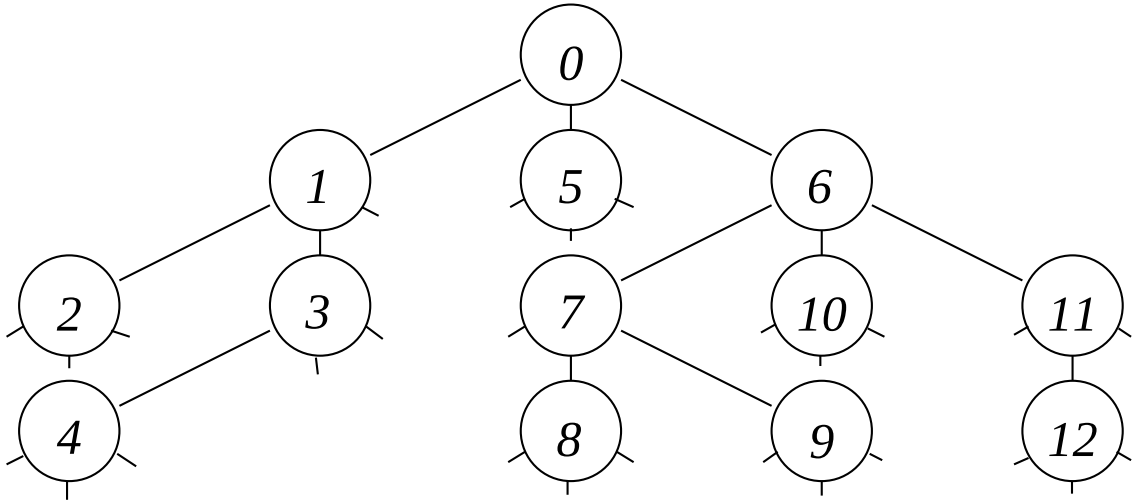
Opovídající datový typ zapsaný v Haskellu např. pro $n = 3$ je

```
data Tree3 b  =  Empty  |  Node b (Tree3 b) (Tree3 b) (Tree3 b)
```

BinÆrní strom (s pevným počadím nÆsledníků)



TernÆrní strom (s pevným počadím nÆsledníků)



BinÆrní stromy

Def: *BinÆrní strom* je strom arity 2.

ZÆkladní operace nad binÆrním stromem

$$\texttt{empty} : \texttt{T}$$

$$\texttt{node} : \texttt{B} \times \texttt{T} \times \texttt{T} \rightarrow \texttt{T}$$

$$\texttt{rootval} : \texttt{T} \dashrightarrow \texttt{B}$$

$$\texttt{left} : \texttt{T} \dashrightarrow \texttt{T}$$

$$\texttt{right} : \texttt{T} \dashrightarrow \texttt{T}$$

$$\texttt{isempty} : \texttt{T} \rightarrow \texttt{Bool}$$

PoznÆmka: ZÆkladní operace mají konstantní složitost, slouží k popisu datovØ struktury a k její implementaci a k implementaci dalších operací, jako například vyhledÆní / pidÆní / odebrÆní uzlu, vyvÆžení stromu a podobn.

Pro zÆkladní operace, každou hodnotu x typu B a každØ binÆrní stromy l, r typu T musí platit

Axiomy binÆrního stromu

$$\texttt{isempty(empty)} = \texttt{True}$$
$$\texttt{isempty(node(}x\texttt{,}l\texttt{,}r\texttt{))} = \texttt{False}$$
$$\texttt{rootval(node(}x\texttt{,}l\texttt{,}r\texttt{))} = x$$
$$\texttt{left(node(}x\texttt{,}l\texttt{,}r\texttt{))} = l$$
$$\texttt{right(node(}x\texttt{,}l\texttt{,}r\texttt{))} = r$$

Implementace binÆrních stromů v Haskellu

```
data Tree b  =  Empty  |  Node b (Tree b) (Tree b)

isempty :: Tree b → Bool
isempty Empty          =  True
isempty (Node _ _ _)   =  False

rootval :: Tree b → b
rootval (Node x _ _)   =  x

left :: Tree b → Tree b
left (Node _ l _)   =  l

right :: Tree b → Tree b
right (Node _ _ r)   =  r
```

Stromy s neomezeným větvením

Def: Je dÆna bÆzovÆ množina (typů) B. Nechť $b \in B$. Pak pro každØ přirozenØ číslo $k \geq 0$ a každou k-prvkovou posloupnost neprÆzdných stromů nad B je dvojice $(b, [T_1, \ldots, T_k])$ neprÆzdným *stromem s neomezeným větvením* nad B.

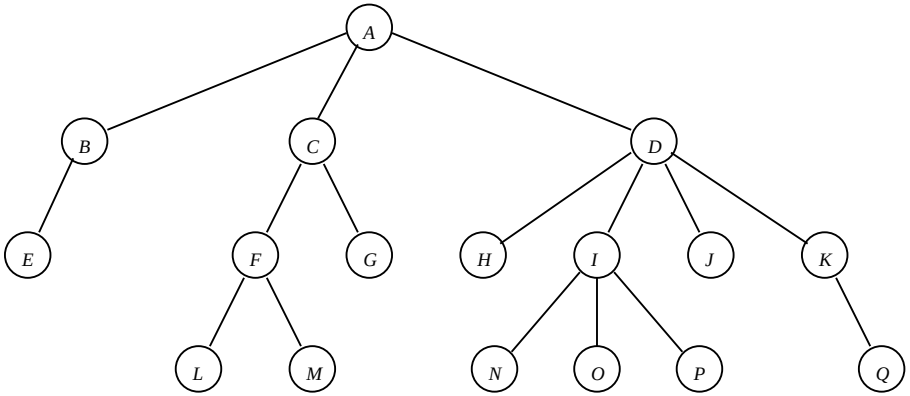
PoznÆmka: Druhou složkou uspořÆdanØ dvojice $(b, [T_1, \ldots, T_k])$ je k-prvkovÆ posloupnost stromů, tj. seznam dØlky k. Je-li $k = 0$, je seznam prÆzdný a dvojice reprezentuje jednouzlový strom (list) s ohodnocením b.

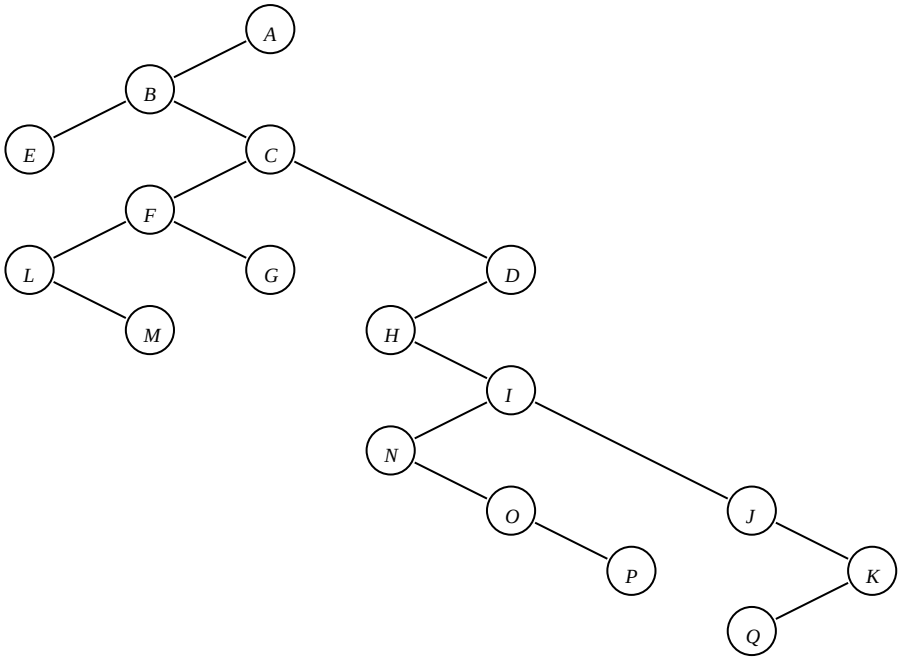
Stromy s neomezeným větvením se od stromů pevnØ arity liší tím, že číslo k není předem pevně danØ, ale může být v jednom stromu pro různØ uzly různØ.

PoznÆmka: Stromy s neomezeným větvením lze reprezentovat binÆrními stromy.

Stromy s neomezeným větvením a binÆrní stromy

```
data NTree a   =  NNode a [ NTree a ]
data BTree a   =  BEmpty  |  BNode a (BTree a) (BTree a)

fb                    ::  [ NTree a ] → BTree a
fb []                  =  BEmpty
fb (NNode v fr : frb)  =  BNode v (fb fr) (fb frb)

nb :: NTree a → BTree a
nb t  =  fb [t]

bf                ::  BTree a → [ NTree a ]
bf BEmpty          =  []
bf (BNode v ec ys)  =  NNode v (bf ec) : bf ys

bn ::  BTree a → NTree a
bn t  =  head (bf t)
```


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q


A
B
E
C
F
L
G
D
M
H
I
N
O
J
P
K
Q

VyhledÆvÆní

Def: Nechť $(K, \leq)$ je œplně uspořÆdanÆ množina tzv. *klíčů*, V je libovolnÆ množina tzv. doplňujících œdajů. Nechť $U = K \times V$ je množina dvojic (k, v), v níž se každý klíč k vyskytuje nejvýše jednou (tj. každý zÆznam z množiny U je určen jednoznačně svým klíčem).

ProblØmu nalØzt k danØmu klíči k zÆznam $(k, v) \in U$ se říkÆ *vyhledÆvací problØm*

PoznÆmka: Například zÆznamy mohou být osobní data a klíči jsou rodnÆ čísla, anebo zÆznamy jsou œdaje o knihÆch v knihovnØ a klíče jsou knihovní signatury apod.

PoznÆmka: V praxi mÆ vÄtšinou smysl pouze případ, kdy množina V je netriviÆlní, aby bylo co hledat.

V ukÆzkových algoritmech se však doplňující œdaje často neuvažují, tj. vyhledÆvají se jen klíče. PříslušnØ rozšíření těchto algoritmů je totiž přímočarØ.

Algoritmus binÆrního vyhledÆvÆní

```
type Elem = Integer;
     Pole = array [1..999] of Elem;

function bSearch (k: Elem; var D: Pole; n: Integer): Integer;
  { Posl. D je rostoucí. Když D[i]=k, tak bSearch(k)=i, jinak bSearch(k)=-1 }

  function bs (l, r: Integer) : Integer;
    var m : Integer;
    begin
      if l > r then bs := -1  {nenalezeno}
               else begin m := (l+r) div 2;
                          if        k < D[m] then bs := bs(l,m-1)
                            else if k > D[m] then bs := bs(m+1,r)
                            else   {k = D[m]}     bs := m
                    end
    end {bs};

  begin  bSearch := bs (1,n)  end
```

Věta: Algoritmus binÆrního vyhledÆvÆní mÆ logaritmickou časovou složitost, tj. jeho složitost je v $\Theta(\log n)$, kde n je dØlka prohledÆvanØho pole.

PoznÆmka: Extrasekvenční paměťovÆ složitost algoritmu je konstantní, protože rekursivní volÆní v něm jsou *prostÆ*

Cvičení: Implementujte algoritmus binÆrního vyhledÆvÆní bez použitírekurse.

Cvičení: Dokažte totÆlní korektnost algoritmu binÆrního vyhledÆvÆní.

HašovÆní

O množině K všech možných klíčů obecně předpoklÆdÆme jen to, že na ní existuje œplnØ uspořÆdÆní. Někdy však tato množina může mít další speciÆlní vlastnosti, jichž lze využít pro efektivní vyhledÆvÆní. Například je-li $K = \{1, \ldots, m\}$ a číslo m je malØ, můžeme data z množiny V uložit do jednorozměrnØho pole a klíče využít jako indexy. VyhledÆvÆní v takovØto datovØ struktuře je efektivní – stejně rychlØ jako indexovÆní pole. Navíc, pokud se počet n skutečně uložených prvků bude blížit počtu m všech možných klíčů, bude efektivní i využití paměti.

Pokud je $|K| = m$, ale klíče nejsou čísla, lze to obejít pomocí bijektivní funkce $h : K \to \{1, \ldots, m\}$ a pole indexovat pomocí fukčních hodnot $h(k)$, kde $k \in K$. Je však důležitØ, aby výpočet hodnot funkce h byl rychlý, v ideÆlním případě aby měl konstantní časovou složitost. Funkci h nazývÆme *hašovací funkcí* a pole indexovanØ jejími hodnotami nazývÆme *hašovací tabulkou*.

Hašovací tabulky

Hašovací tabulka je datovÆ struktura, pomocí níž lze prakticky efektivně realizovat „slovníkovØ“ operace vyhledÆní, přidÆní a zrušení položky.

„Prakticky efektivně“ znamenÆ, že operace mají příznivou *průměrnou* časovou složitost (tedy ne nutně časovou složitost v nejhorším případě).

Hašovací tabulka je jednorozměrnØ pole H indexovanØ čísly $1, \ldots, n$. Převod klíčů na čísla realizuje *hašovací funkce* $h : K \to \{1, \ldots, n\}$. Výpočet hodnot hašovací funkce musí být efektivní, nejlØpe složitosti $\Theta(1)$.

Častým případem však je, že počet n skutečně uložených prvků je podstatně menší než počet m všech možných klíčů. I v tomto případě lze postupovat podobně a data uklÆdat do n-prvkovØho pole, až na to, že funkce $h : K \to \{1, \ldots, n\}$ nebude injektivní. Bude tedy existovat index i a klíče $k_1, \ldots, k_r$, tak, že $h(k_1) = \cdots = h(k_r) = i$. To nemusí vadit, pokud je splněna nÆsledující podmínka. Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, kterÆ budou skutečně uložena (tedy $|K'| \leq n$). Pak požadujeme, aby z každØ množiny klíčů, kterØ hašovací funkce zobrazí na stejný index, byl v množině K' nejvýše jeden klíč. MÆ-li pro pevně danou množinu K' klíčů skutečně uložených dat hašovací funkce tuto vlastnost, nazývÆme ji *dokonalou hašovací funkcí pro klíče z* K'. Výhodou tabulek s dokonalými hašovacími funkcemi je optimÆlní složitost vyhledÆní, vložení i zrušení prvku; je stejnÆ jako pro pole.

Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, kterÆ budou skuteènì uložena. Klíče z množiny K' nazveme *použitØ klíče*.

Je-li zoženíh$|_{K'} : K' \to \{1, \ldots, n\}$ injektivní, říkÆme, že hašovací funkce je *dokonalÆ* vzhledem k množině použitých klíčů K'.

Věta: MÆ-li výpoèet hašovací funkce konstantní složitost a hašovÆní je dokonalØ pro množinu použitých klíčů K', pak operace vyhledÆní, vložení, resp. zrušení prvku v hašovací tabulce mají stejnou časovou složitost, jako vyhledÆní, vložení, resp. zrušení prvku v poli.

Nevýhodou dokonalých hašovacích funkcí je, že závisí na množině K'. Tato množina musí být známa předem, abychom mohli dokonalou hašovací funkci sestrojit. V praxi však ukládané data předem neznáme a tato data se mění. Proto v praxi používané hašovací funkce většinou nebývají dokonalé a musí se počítat s takzvanými *kolizemi*.

Kolize je případ, kdy má být více dat uloženo na jedné pozici hašovací tabulky, tj. chceme uložit data s různými klíči $k_1, \ldots, k_r$ a $h(k_1) = \cdots = h(k_r)$.

Kolize

Tzv. kolize vznikají při nedokonalém hašování: hašovací funkce zobrazuje více použitých klíčů na stejnou pozici pole.

Nejjednodušší řešení kolizí je ukládat do každé pozice pole *seznam* prvků. V něm se pak hledá sekvenčně.

Složitost přidání prvku zůstává stejná jako složitost indexování, tedy složitost vyhledání i složitost zrušení prvku je $\Theta(n)$. To je sice velmi špatný výsledek, ale v praxi nevadí, protože *průměrná* časová složitost dopadne pro vhodně sestrojenou hašovací funkci mnohem lépe.

Věta: Nechť hašovací funkce zobrazuje použité klíče na indexy $1, \ldots, n$ rovnoměrně, tj. pravděpodobnost, že $h(k) = i$, je stejná pro všechny indexy i, $1 \leq i \leq n$.

Pak průměrná časová složitost vyhledání, přidání a zrušení prvku pak je $O\left(\left|K'\right|/n\right)$, za předpokladu, že složitosti výpočtu hašovací funkce i indexování pole jsou konstantní.

Pravděpodobnostní rozložení výskytů klíčů v množině K' nemusí být rovnoměrné, ale dobře navržená hašovací funkce transformuje toto rozložení na rovnoměrné v množině indexů $\{1, \ldots, n\}$. To znamená, že pro náhodně zvolený klíč $k \in K'$ a index i, $1 \leq i \leq n$, je pravděpodobnost, že hodnota $h(k) = i$, stejná nezávisle na k a i a rovná $\frac{1}{n}$.

Nyní pro jednoduchost předpokládejme, že $K = \{1, \ldots, m\}$ a $m \geq n$, kde n je velikost hašovací tabulky. Nechť p je prvočíslo, $p \geq m$, a nechť a, b jsou celá čísla, $0 < a < p$, $0 \leq b < p$. Hašovací funkci $h_{a,b}$ zavedeme takto:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod n$$

Pak při volbě parametrů a, b nezávisle na K (což je v praxi splněno) mají hodnoty $h(k)$ rovnoměrné pravděpodobnostní rozložení na množině indexů $\{0, \ldots, n - 1\}$.

BinÆrní vyhledÆvací stromy

Def: *BinÆrní vyhledÆvací strom* je binÆrní strom nad œepr uspoØÆdanou množinou (tzv. *klíčů*) $(K, \leq)$ takový, že pro každý jeho podstrom t platí:
hodnoty uzlů v podstromu $\texttt{left}(t)$ jsou menší než $\texttt{rootval}(t)$ a hodnoty uzlů v podstromu $\texttt{right}(t)$ jsou větší než $\texttt{rootval}(t)$.

Operace nad binÆrními vyhledÆvacími stromy

Datový typ

```
data  STree a  =  Empty  |  Node a (STree a) (STree a)
```

ZjišťovÆní příslušnosti

```
member                :: a → STree a → Bool
member _ Empty         = False
member k (Node v l r) =    k == v
                        || k < v && member k l
                        || k > v && member k r
```

VyhledÆvÆní

```
search ::  a → STree a → STree a
search _ Empty           = Empty
search k t@(Node v l r)
       | k == v          = t
       | k < v           = search k l
       | otherwise       = search k r
```

VklÆdÆní uzlu

```
insert                    :: a → STree a → STree a
insert k Empty             = Node k Empty Empty
insert k t@(Node v l r)
       | k < v             = Node v (insert k l) r
       | k > v             = Node v l (insert k r)
       | otherwise         = t
```

Rušení uzlu

```
delete :: a → STree a → STree a
delete _ Empty          = Empty
delete k (Node v l r)
       | k < v          = Node v (delete k l) r
       | k > v          = Node v l (delete k r)
       | otherwise      = join l r

join :: STree a → STree a → STree a
join l     Empty  =  l
join Empty r      =  r
join l     r      =  Node u (delete u l) r
                     where u = rightmostkey l

rightmostkey (Node v _ Empty) = v
rightmostkey (Node _ _ r)     = rightmostkey r
```

Cvičení: Vybudujte vyhledÆvací strom z posloupnosti klíčů
$9, 12, 10, 7, 12, 1, 8, 5, 11, 4, 0, 14, 13, 3, 6, 2$.

Cvičení: Zrušte v něm uzel s klíčem 5.

Cvičení: Definujte v Haskellu funkci `leftmostkey`.

Cvičení: Dokažte, že binÆrní strom vzniklý vložením resp. zrušením uzlu z vyhledÆvacího stromu pomocí funkcí `insert` resp. `delete` bude opět vyhledÆvací.

Cvičení: Modifikujte funkce `member`, `search`, `insert`, `delete` tak, aby pracovaly s realističtějším vyhledávacím stromem, v němž jsou kromě klíčů uložena i vlastní data:

```
data STree a b = Empty | Node (a,b) (STree a b) (STree a b)

member :: a → STree a b → Bool

search :: a → STree a b → Maybe b

insert :: (a,b) → STree a b → STree a b

delete :: a → STree a b → STree a b
```

Složitost vyhledÆvÆní ve vyhledÆvacích stromech

Označíme-li $T : \mathbb{N} \to \mathbb{N}$ složitost funkce `search`, pak

$$\begin{aligned} T(0) &= c \\ T(h) &= c' + \max\{T(h'), T(h'')\} \end{aligned}$$

kde h je hloubka vyhledÆvacího stromu,h' je hloubka jeho levØho podstromu,h'' je hloubka jeho pravØho podstromu ac, c' jsou konstanty.

Zřejmě $\max\{T(h'), T(h'')\} = h - 1$ a řešením uvedenØ rekursivní soustavy rovnic je lineÆrní funkce.

Složitost vyhledÁvÁní ve vyhledÁvacích stromech

Složitosti operací vyhledÁní, pidÁní a zrušení položky ve vyhledÁvacím stromjsou přímo œmČrnØ hloubce stromu. Ta je v nejhorším případě lineÁrnČ zÁvislÁ na velikosti stromu (počtu jeho uzlů).

Složitost vyhledÁvÁní, vklÁdÁní a rušení v obecnØm vyhledÁvacím stromě je tedy lineÁrní.

AVL stromy

NazvanØ podle G. M. Adeľsona-VeľskØho a E. M. Landise.

Def: VyhledÆvací binÆrní strom je *AVL*, když hloubka levØho a pravØho podstromu libovolnØho uzlu se liší nejvýše o jednu.

Věta: Hloubka AVL stromu v zÆvislosti na počtu jeho uzlů je vždy v $\Theta(\log)$.

AVL stromy tedy mají logaritmickou hloubku — použijeme-li je jako vyhledÆvací stromy, pak mÆ operace vyhledÆní položky logaritmickou složitost.

Def: *Fibonacciho strom* řÆdu k definujeme takto:

$$\begin{aligned} FT_0 &= \texttt{empty} \\ FT_1 &= \texttt{node}(\texttt{empty}, \texttt{empty}) \\ \text{pro } k \in \mathbb{N} \text{ je } FT_{k+2} &= \texttt{node}(FT_k, FT_{k+1}) \end{aligned}$$

Lemma: Pro $k \geq 1$ je Fibonacciho strom FT_k *minimÆlní* (vzhledem k počtu uzlů) AVL strom hloubky $k-1$.

Důkaz: Indukcí přes k se snadno ukÆže, že hloubka neprÆzdnØho stromu FT_k je $k-1$.

Odtud a z definice Fibonacciho stromů vyplývÆ, že Fibonacciho stromy jsou AVL.

Minimalita je důsledkem předchozích dvou bodů: odebrÆním nějakØho uzlu z jinØ než nejpravější větve by někde vznikly sousední podstromy s hloubkami lišícími se aspoň o dvě; odebrÆním uzlu z nejpravější větve by se snížila hloubka celØho stromu.

Věta: Hloubka AVL stromu v závislosti na počtu jeho uzlů je vždy v $\Theta(\log)$.

Důkaz: Označme $N(k)$ velikost stromu FT_k. Tedy $N : \mathbb{N} \to \mathbb{N}$ a

$$\begin{aligned} N(0) &= 0 \\ N(1) &= 1 \\ \text{pro } k \in \mathbb{N} \text{ je } N(k+2) &= 1 + N(k) + N(k+1) \end{aligned}$$

Protože funkce N majorizuje Fibonacciho funkci, je $N \in \Omega(\varphi^k)$, kde $\varphi = \dfrac{1+\sqrt{5}}{2}$ a za k bereme řád stromu. Ale víme, že řád stromu a hloubka je (skoro) totéž, takže dostáváme, že počet uzlů Fibonacciho stromu hloubky h roste aspoň tak rychle jako φ^h.

Protože Fibonacciho strom je minimální AVL strom, máme, že počet uzlů *každého* AVL stromu hloubky h roste aspoň tak rychle jako φ^h. Obráceně, každý AVL strom velikosti n má hloubku $O(\log_\varphi n)$.

Víme, že hloubka každØho binÆrního stromu velikosti n je $\Omega(\log_2 n)$.

Dohromady mÆme hloubku v $O(\log_\varphi n) \cap \Omega(\log_2 n) = \Theta(\log)$.

Operace na AVL stromech

DatovÆ struktura:

```
data AVL a = Empty | Node Int a (AVL a) (AVL a)
```

Každý uzel nese informaci o hloubce (jakožto parametr konstruktorovØ funkce `Node`).

VyhledÆvÆní v AVL stromu se neliší od vyhledÆvÆní v obecnØm vyhledÆvacím stromu.

Po přidÆní nebo odebrÆní položky ovšem může nastat situace, že vyhledÆvací strom přestane splňovat podmínku AVL. Pak je nutnØ pozměnit strukturu stromu.

PřidÆvÆní položky (novØho uzlu) do AVL stromu

StejnØ jako u bžnØho vyhledÆvacího stromu, ale s kontrolou vyvÆženosti.

PřidÆvaný uzel oznaíme x.

Pokud se poruší vyvÆženost stromu, nalezne se nejmenší podstrom F, který je nevyvÆžený. Oznaíme-li h jeho hloubku před přidÆním uzlu x, bude po přidÆní jeho hloubka $h + 1$. Kořen stromu F označíme f.

Bez œejmy na obecnosti lze předpoklÆdat, že uzel x je přidÆvÆn do *levØho* podstromu stromu F. Nechť B je levý podstrom stromu F, G je pravý podstrom stromu F.

Strom B je neprÆzdný (jinak by přidÆvÆní uzlu x nemohlo porušit vyvÆženost stromu F). Označíme A resp. D jeho levý resp. pravý podstrom, b bude kořen stromu B.

Rozlišíme dva případy:

1. Uzel x je přidÆn do stromu A. Pak strom A mÆ hloubku $h-1$, stromy D, G mají hloubku $h-2$.

   Vytvoříme strom $T = \texttt{Node}\ h\ b\ A\ (\texttt{Node}\ (h-1)\ f\ D\ G)$.

   Pak T je AVL strom hloubky h se stejnými uzly jako v F.

2. Uzel x je přidÆn do stromu D. Pak stromy A, G mají hloubku $h-2$, strom D mÆ hloubku $h-1$. Označíme d resp. C resp. E kořen resp. levý podstrom resp. pravý podstrom stromu D.

   Vytvoříme strom

$$T = \texttt{Node}\ h\ d\ \ (\texttt{Node}\ (h-1)\ b\ A\ C)\ \ (\texttt{Node}\ (h-1)\ f\ E\ G)$$

Dva dílčí podpřípady jsou:

(a) Uzel x byl přidÆn do stromu C. Pak C mÆ hloubku $h-2$, E mÆ hloubku $h-3$.

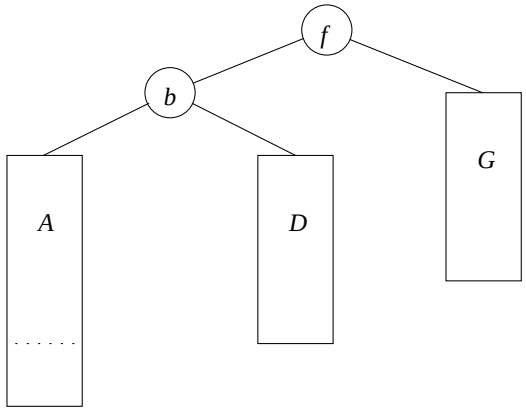
(b) Uzel x byl přidÆn do stromu E. Pak C mÆ hloubku $h-3$, E mÆ hloubku $h-2$.

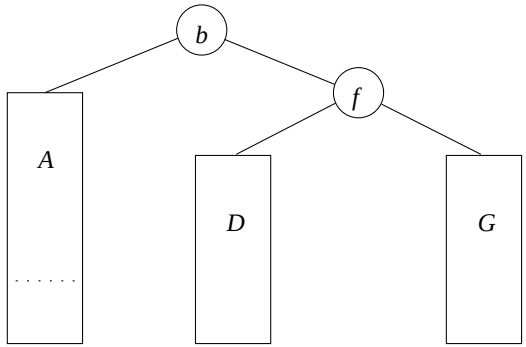
V obou případech však strom T mÆ hloubku h, je AVL a mÆ stejnØ uzly jako strom F.

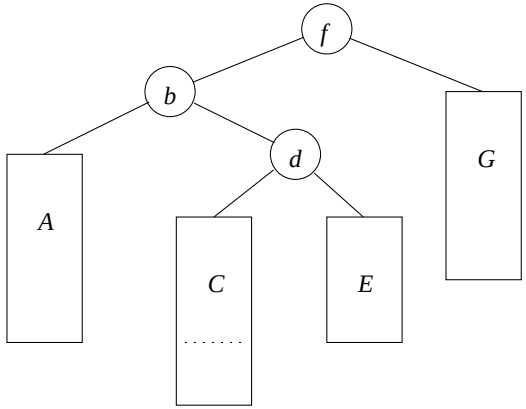
V původním AVL stromu nahradíme podstrom F stromem T. Jelikož T mÆ stejnou hloubku jako měl původní podstrom bez uzlu x, zůstane celý strom vyvÆžený (AVL).

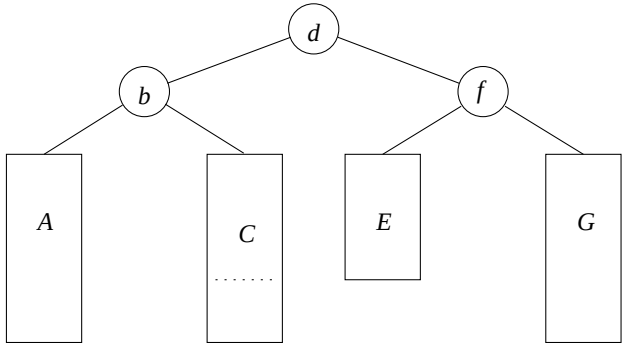
PřepočítÆme hloubky na cestě od kořene stromu T k uzlu x.

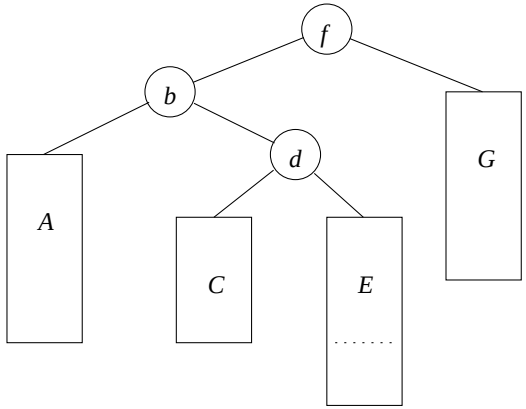
Byl-li uzel x přidÆvÆn do *pravØho* podstromu stromu F, je postup analogický (stranově převrÆcený).

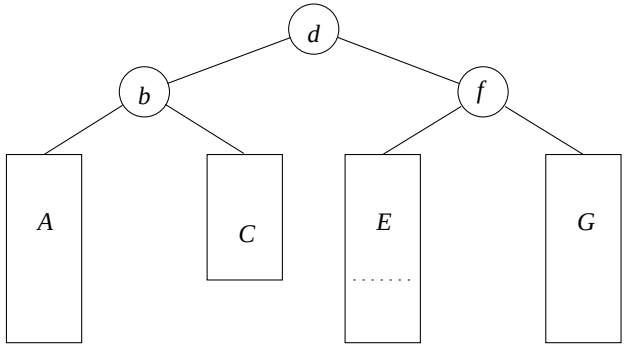

f
b
A
D
G


b
f
A
D
G


f
b
d
G
A
C
E


d
b
f
A
C
E
G


f
b
d
G
A
C
E


d
b
f
A
C
E
G

Složitost operace přidÆní položky do AVL

Hloubka AVL stromu je logaritmickÆ vzhledem k počtu jeho uzlů.

PřidÆní uzlu x: $\Theta(\log n)$.

Nalezení nejmenšího nevyvÆženØho podstromu:
Protože hloubky podstromů jsou spočteny a uloženy v datovØ struktuře, stačí po cestě k uzlu x testovat, zda rozdíl spočtených hloubek levØho a pravØho podstromu každØho uzlu je v množině $\{-1, 0, 1\}$. Nejnižší uzel, který tuto podmínku nesplňuje, je kořen podstromu F. Jeho nalezení trvÆ $\Theta(\log n)$.

Hloubky podstromů se přepočítÆvají jen po cestě od kořene k x, tedy složitost tØto œpravy je $\Theta(\log n)$.

CelkovÆ složitost přidÆní uzlu je tedy $\Theta(\log n)$.

Rušení položky (uzlu) z AVL stromu

Rušení vnitřního uzlu převedeme na rušení listu (podobně jako u vyhledÆvacího stromu).

Odebíraný list označíme x.

Pokud se poruší vyvÆženost stromu, nalezne se nejmenší podstrom B, který je nevyvÆžený. Označíme h jeho hloubku (před i po zrušení uzlu x je stejnÆ).

Bez œejmy na obecnosti lze předpoklÆdat, že uzel x byl odebrÆn z *levØho* podstromu stromu B. Nechť A je levý podstrom stromu B, F je pravý podstrom stromu B.

Strom F je neprÆzdný (jinak by rušení uzlu x nemohlo porušit vyvÆženost stromu B). Označíme D resp. G jeho levý resp. pravý podstrom, f bude kořen stromu F.

Rozlišíme dva případy:

1. Hloubka stromu G je větší nebo rovna hloubce stromu D.

   Vytvoříme strom $T = \texttt{Node}\ h'\ f\ (\texttt{Node}\ (h'-1)\ b\ A\ D)\ G$.

   Pak T je AVL strom hloubky h' se stejnými uzly jako v B, $h' = h$ nebo $h' = h - 1$.

2. Hloubka stromu G je menší než hloubka stromu D. Označíme d resp. C resp. E kořen resp. levý podstrom resp. pravý podstrom stromu D.

   Vytvoříme strom

   $$T = \texttt{Node}\ h'\ d\ \ (\texttt{Node}\ (h'-1)\ b\ A\ C)\ \ (\texttt{Node}\ (h'-1)\ f\ E\ G)$$

   Pak strom T mÆ hloubku $h' = h - 1$, je AVL a mÆ stejnØ uzly jako ně kterÝ strom B.

V původním AVL stromu nahradíme podstrom B stromem T.

PřepočítÆme hloubky na cestě od kořene stromu T k uzlu x.

V případě rušení uzlu se může stÆt, že hloubka podstromu T bude menší než hloubka původního podstromu, který byl na jeho místě.

Proto je nutnØ proces vyvažovÆní opakovat: nalezne se nejmenší nadstrom T_2 stromu $T = T_1$, který není vyvÆžený, vyvÆží se ho, nalezne další nevyvÆžený nadstrom T_3 . . . atd., až je vyvÆžený celý strom, T_k.

Hloubky však stačí přepočítÆvat vždy od kořene stromu T ke kořenu nejbližšího vyvažovanØho nadstromu.

Složitost operace rušení položky z AVL

Zrušení listu: $\Theta(\log n)$.

Nechť hloubka nejmenšího nevyvÆženØho podstromu T_1 je h_1. Nalezení tohoto podstromu trvÆ $\Theta(h_1)$, jeho vyvÆžení (rotace uzlů) trvÆ konstantní dobu, přepočítÆní hloubek trvÆ $\Theta(h_1)$.

Nechť pro $1 < i \leq k$ je h_i dØlka cesty z kořene podstromu T_{i-1} do kořene stromu T_i. Pak každØ nalezení dalšího nevyvÆženØho podstromu T_i trvÆ $\Theta(h_i)$, jeho vyvÆžení trvÆ $\Theta(1)$, přepočítÆní hloubek trvÆ $\Theta(h_i)$.

To znamenÆ, že celkovÆ složitost rušení uzlu je $\Theta\left(\sum_{i=1}^{k} h_i\right) = \Theta(\log n)$.

Cvičení: Na vstupu jsou čísla $8, 3, 5, 0, 2, 4, 1, 6, 9, 7$ a při jejich načítání se postupně vytváří AVL strom s uzly ohodnocenými těmito čísly. Nakreslete tento AVL strom v každém kroku vytváření (tj. po přidání každého uzlu).

Cvičení: Do AVL stromu, který je zpočátku prázdný, se postupně přidávají položky $4, 1, 2, 7, 5, 6$, pak se odebere položka 7, přidá položka 3, odeberou se postupně $5, 6, 1$. Určete stav AVL stromu v každém kroku.

Cvičení: AVL strom má uzly ohodnocené čísly 1 až 20 a má strukturu Fibonacciho stromu šestého řádu. Popište a nakreslete proces rušení listu obsahujícího klíč 2.

Cvičení: Fibonacciho strom čtvrtého řádu na sedmi uzlech je ohodnocen jako AVL strom čísly $1, 3, 5, 7, 9, 11, 13$. Do tohoto stromu přidáme jako další položku náhodně zvolené sudé číslo k, $0 \leq k \leq 14$. Jaká je pravděpodobnost, že budeme muset rotovat uzly, abychom zachovali AVL vyváženost?

ČernobílØ stromy

ČernobílØ stromy jsou binÆrní vyhledÆvací stromy, jejichuzly nesou kromě klíče další atribut — *barvu* — černou nebo bílou.

Def: *Černobílý strom* je binÆrní vyhledÆvací strom, jehož každý uzel je obarven černou nebo bílou barvou. Musí splňovat tyto podmínky:

1. Kořen stromu je černý.
2. Je-li vnitřní uzel bílý, jeho nÆsledníci (pokud existují) jsoučerní.
3. Všechny větve obsahují stejný počet černých uzlů.

PoznÆmka: Vedle nÆzvučernobílØ *stromy* se lze setkat i s doslovnými překlady anglickØho nÆzvu*red-black tree* (drzewo czerwono-czarne, rot-schwarzer Baum, ruĝnigra arbo, . . .).

Def: *ČernÆ hloubka* černobílØho stromu t je počet černých uzlů na libovolnØ větvi. Značíme ji $\mathrm{bh}(t)$.

Lemma: Hloubka černobílØho stromu t na n uzlech je nejvýše $2\log_2(n+1)$.

Důkaz: Indukcí podle hloubky stromu se ukÆže, že každý černobílý strom t' mÆ aspoň $2^{\mathrm{bh}(t')} - 1$ uzlů. Odtud vyplývÆ $\mathrm{bh}(t') \leq \log_2(n+1)$. Ale podle definice černobílØho stromu jeho hloubka nepřevyšuje dvojnÆsobek jeho černØ hloubky. Odtud plyne tvrzení.

Důsledek: VyhledÆvÆní (operace `member` a `search`) v černobílØm stromě mají složitost $\Theta(\log n)$.

ČernobílØ stromy v Haskellu

```
data  Barva  =  Ce  |  Bi
data  CBS a  =  E  |  N Barva a (CBS a) (CBS a)
```

VyhledÆvÆní v černobílØm stromu je stejnØ jako v nevyvÆženØm vyhledÆvacím stromu.

ZjišťovÆní příslušnosti

```
member ::  a → CBS a → Bool
member _ E            = False
member k (N _ v l r) =    k == v
                       || k < v && member k l
                       || k > v && member k r
```

VyhledÆvÆní

```
search :: a → CBS a → CBS a
search _ E                  = E
search k t@(N _ v l r)
       | k == v             = t
       | k < v              = search k l
       | otherwise          = search k r
```

PřidÆvÆní uzlu do černobílØho stromu

PřidÆvaný uzel bude bílý. Tím se nezmění černÆ hloubka podstromů, ale mohou se dostat pod sebe dva bílØ uzly.

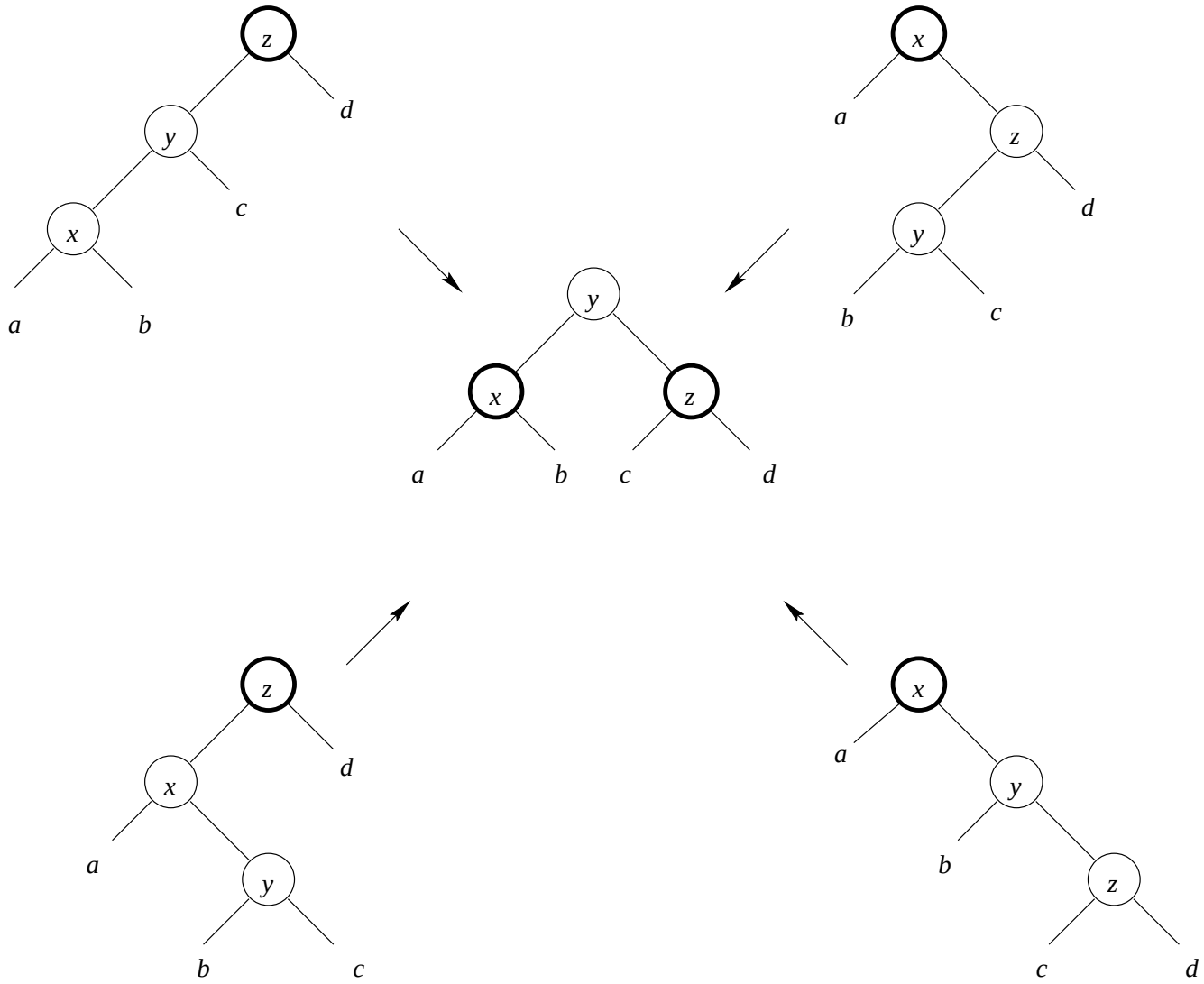
Se dvěma bílými uzly nad sebou a s černým uzlem nad nimi provedeme takovou rotaci, abychom snížili hloubku stromu, ale přebarvíme je tak, aby černÆ hloubka stromu zůstala zachovÆna: kořen bude bílý a jeho dva nÆslednící černí.

Tím se opět mohly dostat pod sebe dva bílØ uzly. Proto celý postup opakujeme tak dlouho, dokud jsou někde pod sebou dva bílØ uzly.

Zůstane-li bílý kořen, přebarvíme ho na černo. Tím se černÆ hloubka celØho stromu zvýší o jedničku.

PřidÆvÆní uzlu

```
insert ::  a → CBS a → CBS a
insert k s = N Ce y tl tr
             where N _ y tl tr       = ins s
                   ins E             = N Bi k E E
                   ins t@(N b y l r)
                       | k < y       = bal (N b y (ins l) r)
                       | k > y       = bal (N b y l (ins r))
                       | otherwise   = t

bal :: CBS a → CBS a
bal (N Ce z (N Bi y (N Bi x a b) c) d)  =  rt
bal (N Ce z (N Bi x a (N Bi y b c)) d)  =  rt
bal (N Ce x a (N Bi z (N Bi y b c) d))  =  rt
bal (N Ce x a (N Bi y b (N Bi z c d)))  =  rt
bal t                                  =  t
    where  rt  =  N Bi y (N Ce x a b) (N Ce z c d)
```


z
y
x
a
b
c
d
x
a
z
y
b
c
d
y
x
z
a
b
c
d
z
x
d
a
y
b
c
x
a
y
b
z
c
d

Složitost operace přidÆní uzlu

Věta: Operace `insert` přidÆní položky dočernobílØho stromu velikosti n mÆ časovou složitost $\Theta(\log n)$.

Důkaz: VyplývÆ z logaritmickØ hloubky černobílØho stromu a z toho, že operace vyvÆženíbal mÆ konstantní složitost.

Cvičení: Proč se při přidávání položky do červenobílého stromu obarvuje kořen celého stromu na černo?

Cvičení: Na vstupu je posloupnost $6, 5, 4, 2, 3, 1, 0$, z jejíchž prvků se postupně vytváří červenobílý strom. Jaké jsou tyto stromy v každém kroku?

Cvičení: Nechť černožlutobílý strom je binární strom splňující podmínky:

- každý uzel je obarven jednou barvou — černou, žlutou, anebo bílou
- počet černých uzlů na každé větvi je stejný
- bezprostřední předchůdce žlutého uzlu nesmí být žlutý
- bezprostřední předchůdce bílého uzlu nesmí být bílý ani žlutý
- kořen stromu je vždy černý

Jaká je minimální a jaká maximální hloubka černožlutobílého stromu na n uzlech? Dokažte.

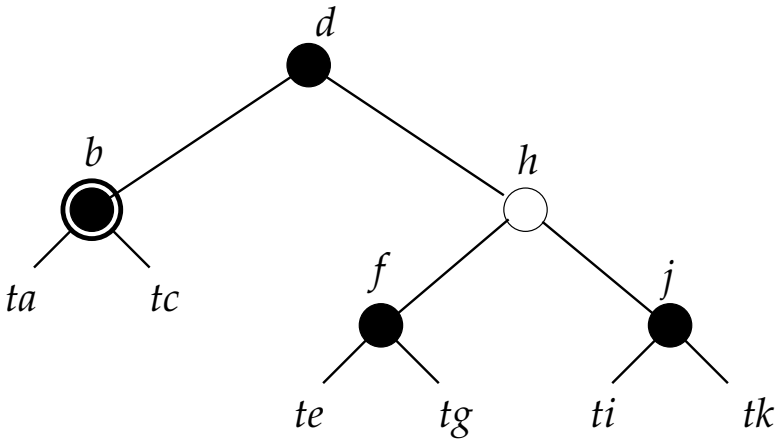
Rušení uzlu z černobílého stromu

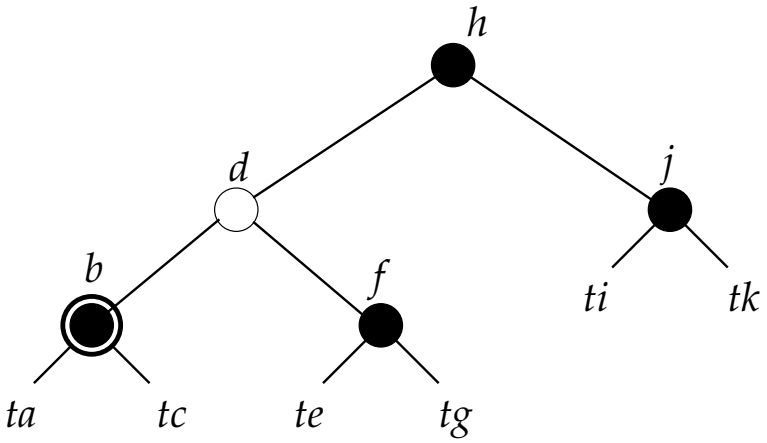
Rušení vnitřního uzlu převedeme známým způsobem na rušení listu. Je-li rušený list bílý, je zrušení triviální — strom i bez listu zůstane černobílý. Je-li rušený list černý, je nutno ho nejdříve „odbarvit“. Odbarvíme-li černý list, stane se tento list bílým a můžeme ho snadno zrušit.

Operace odbarvení spočívá v přesunutí přebytečné černé barvy blíže ke kořenu tak, aby černé délky všech větví zůstaly stejné. Přitom může dojít k „přibarvení“ černého uzlu – přesuneme-li černou barvu na uzel, který byl sám černý, stane se tento uzel „dvojnásobně černý“ a je nutno ho dále odbarvovat (přesouvat čerň stále blíže ke kořenu), aby byl každý uzel „nejvýše jednou černý“.

Stane-li se dvojnásobně černý kořen celého stromu, přebytečnou černou barvu z něho smažeme a necháme ho „jednoučerný“. Tím se sníží černá hloubka celého stromu.

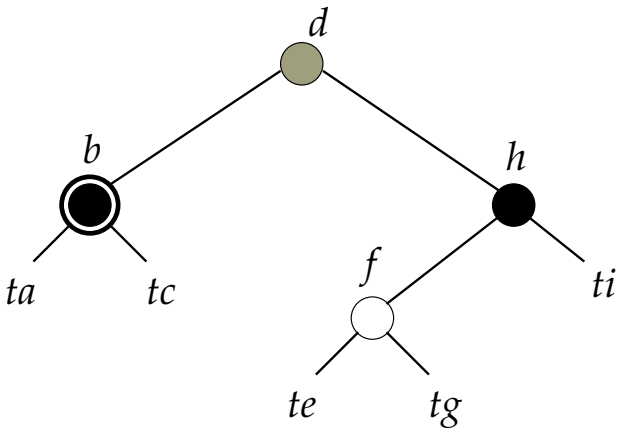
1. Bratr odbarvovanØho uzlu je bílý — převede se na případ 2

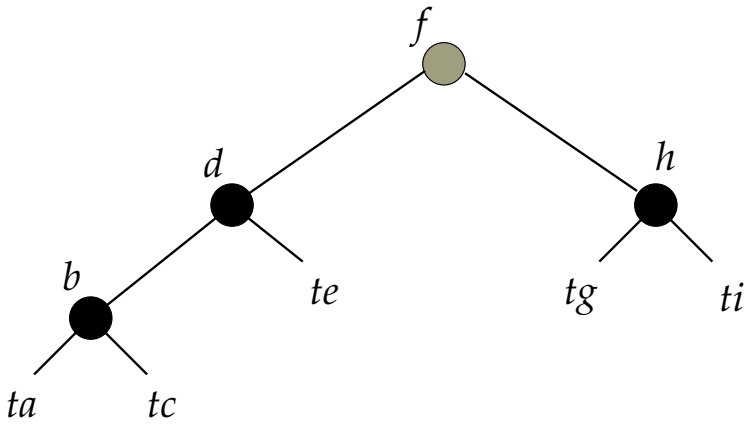




2. Bratr odbarvovanØho uzlu ječerný

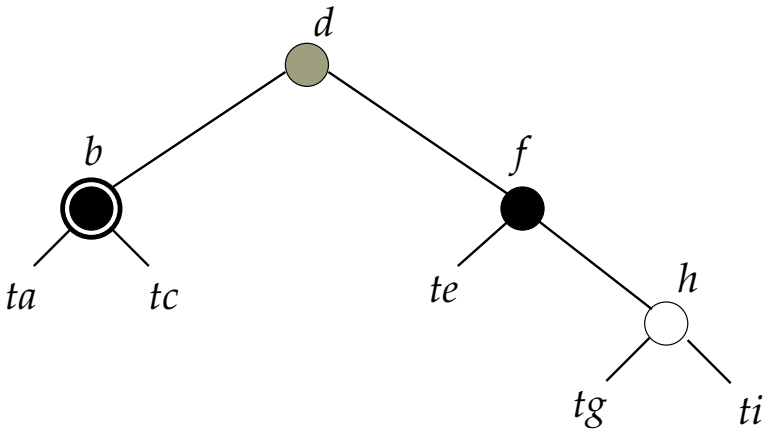
   2a bližší synovec je bílý

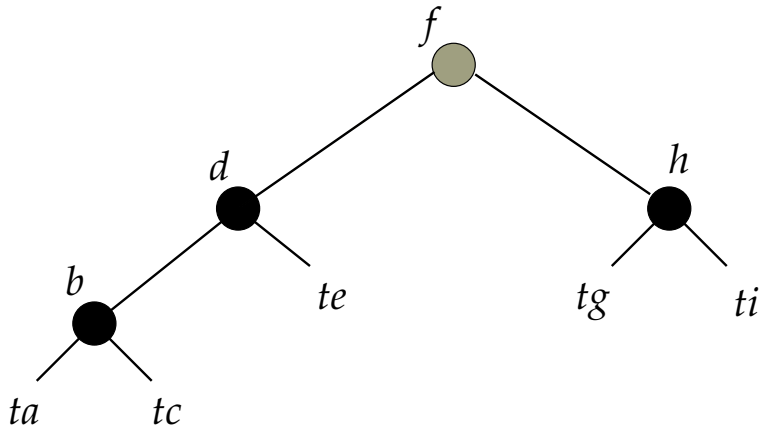




2. Bratr odbarvovanØho uzlu je černý

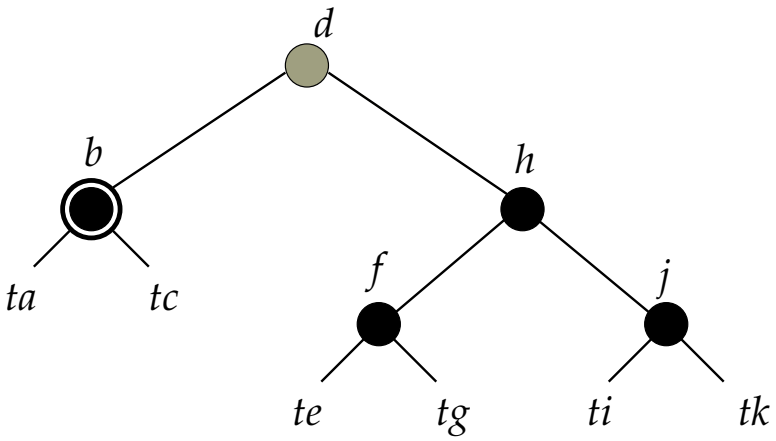
   2b vzdÆlenější synovec je bílý

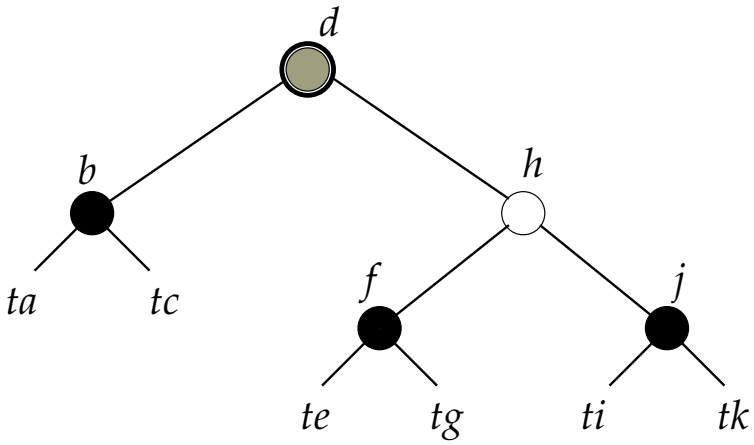




2. Bratr odbarvovanØho uzlu ječerný

   2c žÆdný synovec není bílý





Rušení uzlu

```
delete :: a → CBS a → CBS a
delete k t = cbs (del t)
             where del  E           = E
                   del (N b v l r)
                       | k < v     = lbal b v (del l) r
                       | k > v     = rbal b v l (del r)
                       | otherwise  = join b l r

join :: Barva → CBS a → CBS a → CCBS a
-- definice jako u binárního vyhledávacího stromu
```

```
lbal :: Barva → CCBS a → CBS a → CCBS a
lbal Ce d (CeCe tb) (N Bi h tf tj)                =  -- (1)
        N Ce h (lbal Bi d (CeCe tb) tf) tj
lbal co d (CeCe tb) (N Ce h (N Bi f te tg) ti) =  -- (2a)
        N co f (N Ce d tb te) (N Ce h tg ti)
lbal co d (CeCe tb) (N Ce f te (N Bi h tg ti)) =  -- (2b)
        N co f (N Ce d tb te) (N Ce h tg ti)
lbal co d (CeCe tb) (N Ce h tf tj)                =  -- (2c)
        cern (N co d tb (N Bi h tf tj))
lbal co d tb tf                                   =  -- (3)
        N co d tb tf

rbal :: Barva → CBS a → CCBS a → CCBS a
-- analogicky jako lbal
```

```
data  CCBS a  =  CeCe (CBS a)  |  Cbs (CBS a)

cern :: CBS a → CCBS a
cern E = Cbs E
cern (N Bi v l r) = Cbs (N Ce v l r)
cern (N Ce v l r) = CeCe (N Ce v l r)

cbs :: CCBS a → CBS a
cbs (CeCe t) = t
cbs (Cbs t)  = t
```

Složitost rušení uzlu z černobílého stromu

Elementární přesun černé barvy trvá konstantní dobu, ale v nejhorším případě je nutné odbarvovat až ke kořenu, tj. $\Theta(\log n)$.

Nalezení rušeného uzlu a náhrada rušení vnitřního uzlu za rušení listu trvá také logaritmickou dobu, takže celá operace rušení zabere čas $\Theta(\log n)$.

Cvičení: Implementujte v Haskellu vyvažovací operaci `rbal`.

Cvičení: Napište definici funkce `join`.

Další typy vyhledávacích stromů s logaritmickou složitostí operací

B-stromy Stromy s proměnným počtem následníků, ale omezeným zdola číslem k a shora číslem $2k$ pro pevně zvolenou konstantu k, tzv. řád B-stromu (viz PV062). Složitost operací přidání/zrušení položky je $\Theta\left(\log_k n\right)$.

2-3-4-stromy Speciální případ B-stromů. Každý vnitřní uzel má buď dva, tři, anebo čtyři následníky. Mají opět logaritmickou hloubku.

Hranově a uzlově ohodnocenØ grafy

Def: Nechť V, H_V, H_E jsou množiny, $E \subseteq V^2$, $h_V : V \to H_V$, $h_E : E \to H_E$. Čtveřici (V, E, h_V, h_E) nazývÆme *uzlově a hranově ohodnocený orientovaný graf*.

VynechÆním hranovØho ohodnocenÍ h_E dostaneme uzlově ohodnocený orientovaný graf (V, E, h_V).

VynechÆním uzlovØho ohodnocenÍ h_V dostaneme hranově ohodnocený orientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E dvojice tvaru (x, x), mÆme (ohodnocenØ) orientovanØ grafy bez smyček.

Def: Nechť V, H_V, H_E jsou množiny, E je nějakÆ množina dvouprvkových a jednoprvkových podmnožin množiny V, $h_V : V \to H_V$, $h_E : E \to H_E$. Čtveřici (V, E, h_V, h_E) nazývÆme *uzlově a hranově ohodnocený neorientovaný graf*.

VynechÆním hranovØho ohodnocení h_E dostaneme uzlově ohodnocený neorientovaný graf (V, E, h_V).

VynechÆním uzlovØho ohodnocení h_V dostaneme hranově ohodnocený neorientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E jednoprvkovØ množiny, dostÆvÆme (ohodnocenØ) neorientovanØ grafy bez smyček.

PoznÆmka: Obecnější definice grafu připouští, aby množina hran E nebyla podmnožinou V^2 (resp. množiny všech dvouprvkových a jednoprvkových podmnožin množiny V), ale aby to byla libovolnÆ konečnÆ množina, na níž je dÆno zobrazení $\varepsilon : E \to V^2$, kterØ určuje, mezi kterými uzly danÆ hrana vede. Není-li zobrazení ε injektivní, hovoří se o tzv. *multigrafu*.

Reprezentace

Graf lze reprezentovat *maticí sousednosti*: neohodnocený graf $G = (V, E)$ na n uzlech je reprezentovÆn čtvercovou maticí $\bar{G} = [g_{i,j}]_{1 \leq i \leq n, 1 \leq j \leq n}$ nul a jedniček tak, že $g_{i,j} = 1$, prÆvě když $(i, j) \in E$.

Je-li graf hranově ohodnocený, $G = (V, E, h_E)$, $h_E : E \to H_E$, jsou v matici sousednosti přímo hodnoty hran: pro $(i, j) \in E$ je $g_{i,j} = h(i, j)$, pro $(i, j) \notin E$ je $g_{i,j} = v$, kde $v \notin H_E$.

Pro tzv. *řídkØ grafy*, tj. grafy, v nichž počet hran je v $o(n^2)$, je výhodnÆ reprezentace pomocí *množin sousedů*: neohodnocený graf $G = (V, E)$ na n uzlech je reprezentovÆn posloupností $[s_1, \ldots, s_n]$ množin nÆsledníků každØho uzlu. Jsou-li $v_1, \ldots, v_k$ všichni nÆsledníci uzlu u (tj. uzly, do nichž vede z u hrana), je $s_u = \{v_1, \ldots, v_k\}$.

Nejčastější reprezentace posloupnosti množin sousedů je polem seznamů. Je-li graf uzlově ohodnocený, obsahuje i-tý prvek pole $\bar{G}$ dvojici $(s_i, h_V(i))$. Je-li graf hranově ohodnocený, obsahuje j-tý prvek i-tØho seznamu s_i dvojici $(v_j, h_E(i, v_j))$.

ProchÆzení grafu

ProblØm: Projít systematicky všechny uzly grafu.

ProchÆzení grafu do hloubky

Algoritmus `dfs` (*depth-first search*)

$G = (V, E), \quad V = \{1, \ldots, n\}, \quad W \subseteq V$

Budeme označovat všechny navštívenØ uzly pomocí procedury`mark`.

Na začÆtku jsou všechny uzly nenavštívenØm(`marked`$(u) =$ `False`).

Pro každý uzel u je `Successors`(u) seznam uzlů – nÆsledníků uzlu$u$.

VychÆzíme z uzlů z množiny$W$ a postupujeme stÆle dÆl po hranÆch do dosud nenavštívených uzlů.

Cesty z označených do nově navštívených uzlů tvoří les s kořeny ve W.

ProchÆzení grafu do hloubky – imperativní pseudokód

```
procedure dfs (G:Graph; var W:Nodeset; var P:Nodearray);
  procedure dfs1 (u:Node);
   begin
     if not marked (u)
        then begin mark (u);
                   for v ∈ Successors (u) do
                     begin P[v] := u;
                           dfs1 (v)
                     end
             end
   end;
```

```
begin {dfs}
 for u ∈ V do begin unmark (u);
                    P[u] := empty
                end;
 for u ∈ W do
   if not marked (u) then dfs1 (u)
end;
```

Chceme-li projít do hloubky *všechny* uzly grafu $G = (V, E)$, položíme $W = V$.

Věta: Algoritmus `dfs` navštíví všechny uzly grafu dosažitelnØ z uzlů z množiny W.

Důkaz je zřejmý a plyne z toho, že se opakovaně označují všechny uzly, kterØ dosud označeny nebyly. Množina neoznačených uzlů se zmenšuje, takže algoritmus konverguje.

PoznÆmka: Algoritmus funguje stejně pro orientovanØ i neorientovanØ grafy.

PoznÆmka: Algoritmus obecně není deterministický, protože nemusí být (a nebývÆ) specifikovÆno pořadí uzlů v množinÆch `Successors(u)` ani v množině W. ProchÆzení grafu do hloubky tedy neurčuje pořadí navštívených uzlů jednoznačně.

Cvičení: Nechť $G = (V, E)$ je neorientovaný graf, $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{\{1, 2\}, \{2, 3\}, \{4, 5\}, \{5, 6\}, \{1, 4\}, \{2, 5\}, \{3, 6\}\}$, $W = \{1\}$. JakÆ jsou všechna možnÆ pořadí navštívených uzlů při volÆní `dfs(`G`,`W`)`?

PoznÆmka: Významnou praktickou aplikací algoritmu `dfs` je nalezení tzv. *silně souvislých komponent* orientovanØho grafu.

ProchÁzení grafu do šířky

Řešíme problÉm Projít systematicky všechny uzly grafu $G = (V, E)$ dosažitelnÉ (orientovanou) cestou z některÉho z počÁteČních uzlů zadaných množinou $W \subseteq V$.

Algoritmus `bfs` (*breadth-first search*) prochÁzí uzly v jinÉm poŘadí než při prohledÁvÁní do hloubky: uzly, kterÉ leží blíže počÁteČním uzlům (uzlům, v nichž prochÁzení začÍnÁ), jsou navštíveny dříve než uzly ležící dÁle od počÁteČních uzlů.

PomocnÉ datovÉ struktury: fronta `Q` uzlů, les nejkratších cest (uložený v poli `P`)
NavštívenÉ uzly se opět označují pomocí procedury `mark`,
uzel u bude označen, prÁvě když $\texttt{marked}(u) = \texttt{True}$.

ProchÆzení grafu do šíky – imperativní pseudokód

```
procedure bfs (G:Graph; var W:Nodeset; var P:Nodearray);
  var Q:Queue;
  procedure bfs1 (q:Queue);
    begin while not (isempty(q)) do
            begin u := headq(q);
                  dequeue(q);
                  for v ∈ Successors (u) do
                    if not (marked(v))
                       then begin mark (v);
                                  enqueue (v,q);
                                  P[v] := u
                            end
            end
    end{bfs1};
```

```
begin {bfs}
   for u ∈ V do begin unmark (u);
                      P[u] := empty
                end;
   Q := emptyq;
   for u ∈ W do
     begin  mark (u);
            enqueue (u,Q)
     end;
   bfs1 (Q)
end {bfs}
```

Korektnost prochÆzení grafu do šíky

Def: *DØlkou sledu* (v hranově neohodnocenØm grafu) rozumíme počet hran na tomto sledu.

VzdÆlenost z uzlu u *do uzlu* v je dØlka minimÆlního sledu vedoucího z uzlu u do uzlu v, pokud takový sled existuje; pokud z uzlu u do uzlu v žÆdný sled neexistuje, pak vzdÆlenost položíme $+\infty$.

V neorientovanØm grafu mluvíme stručně o *vzdÆlenosti mezi uzly* u *a* v.

Věta: Nechť $G = (V, E)$, $s, u, v \in V$ a nechť vzdálenost z s do u je menší než vzdálenost z s do v. Pak uzel u bude při výpočtu `bfs`$(G, \{s\})$ označen dříve než uzel v.

Důkaz: Stačí ukázat, že do fronty Q jsou vždy přidávány uzly s aspoň tak velkou vzdáleností od s, než jakou měl poslední přidaný uzel. To však lehce plyne indukcí podle vzdálenosti od uzlu s.

Věta: Nechť $G = (V, E)$, $s \in V$. Pak po provedení výpočtu `bfs`$(G, \{s\})$ jsou navštíveny právě všechny uzly, do nichž vede z uzlu s (orientovaná) cesta.

Důkaz: Indukcí podle vzdálenosti od uzlu s.

Složitost procházení grafu do šířky

Věta: Nechť $G = (V, E)$ je graf a $W \subseteq V$ je množina počátečních uzlů (vstupního stupně 0). Pak délka výpočtu `bfs`(G, W) v nejhorším případě je v $\Theta(|W| + |E|)$.

Důkaz věty: Každý uzel bude označen a zařazen do fronty a u všech uzlů z fronty se ptÆme na označenost jejich nÆsledníků. Tedy dØlka výpočtu bude jistě v $\Omega(|E|)$. Jelikož zařazujeme do fronty všechny uzly z W, je dØlka výpočtu takØ v $\Omega(|W|)$. Z těchto dvou faktů vyplývÆ, že je v $\Omega(|E|) \cap \Omega(|W|) = \Omega(|W| + |E|)$.

Do fronty zařazujeme jen uzly, kterØ až do tØ doby nebyly označenØ, a přitom je hned označíme. Odtud plyne, že každý uzel je do fronty zařazen jen jednou.
Tedy vnější cyklus while proběhne $O(|W| + |E|)$-krÆt. Test ve vnitřním cyklu for proběhne vždy tolikrÆt, kolik hran vychÆzí z uzlu v, dohromady tolikrÆt, kolik mÆ celý graf hran. Odtud opět dostÆvÆme, že dØlka výpočtu je v $O(|W| + |E| + |E|) = O(|W| + |E|)$.

Dohromady dostÆvÆme $\Theta(|W| + |E|)$, čímž je tvrzení dokÆzÆno.

MinimÆlní kostra grafu

Def: Nechť $G = (V, E, h_E)$, $h_E : E \to \mathbb{R}$, je hranově ohodnocený neorientovaný graf. *Faktor* grafu G je podgraf $F = (V, E', h_E|_{E'})$ grafu G.

Def: Takový faktor grafu G, který je strom, se nazývÆ *kostra* grafu G.

Def: Nechť G je souvislý neorientovaný graf s číselně ohodnocenými hranami. Kostra grafu G, kterÆ mÆ ze všech jeho koster nejmenší součet hranových ohodnocení, se nazývÆ *minimÆlní kostra* grafu G.

PoznÆmka: Kostry mají jen souvislØ grafy, protože kostra musí podle definice být strom. Tento požadavek však není zÆsadní a většinu úvah o kostrÆch lze přenØst i na nesouvislØ grafy: stačí uvažovat zvlÆšť kostru každØ komponenty.

Cvičení: MÆme neorientovaný graf $G = (V, E)$, $V = \{1, 2, 3, 4\}$,
$E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}\}$.
Kolik mÆ tento graf podgrafů, faktorů a koster? V každØm z tìchto tří případů spočítejte, kolik je *všech* podgrafů (faktorů, koster), a kolik z nich je *navzÆjem neisomorfních*.

Generický algoritmus pro nalezení minimÆlní kostry

ProblØm: Je dÆn souvislý hranově ohodnocený neorientovaný graf $G = (V, E, h_E)$.

Úkolem je nalØzt jeho minimÆlní kostru.

```
type SpT = Set of Edges;
procedure genericMST (G:Graph; var A:SpT);
  begin
    A := ∅;
    while A ještě netvoří kostru do
      begin
        najdi hranu {u, v} bezpečnou vzhledem k A;
        A := A ∪ {{u, v}}
      end
  end
```

Def: Nechť G je souvislý hranově ohodnocený graf a A takový jeho podgraf, že A je podgrafem nějaké minimální kostry T grafu G. Pak každá hrana z $G - A$, která leží v minimální kostře T, se nazývá *bezpečná* vzhledem k A.

Def: Nechť $G = (V, E)$ je graf, $V_1 \subset V$, $V_2 \subset V$, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$, $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$. Pak množině

$$C = \{\{x, y\} \in E \mid x \in V_1, y \in V_2\}$$

říkáme *řez* grafu G.

Je-li navíc $G' = (V', E', h')$ podgraf grafu G takový, že $V' \subseteq V_1$ nebo $V' \subseteq V_2$, říkáme, že řez C *respektuje* podgraf G'.

Věta: Nechť $G = (V, E, h_E)$ je souvislý hranově ohodnocený neorientovaný graf, $h_E : E \to \mathbb{R}$. Nechť množina hran $A \subseteq E$ je obsažena v nějaké minimální kostře grafu G, nechť C je řez respektující podgraf indukovaný hranami z A a nechť $\{u, v\} \in C$ je hrana s minimálním ohodnocením v C. Pak hrana $\{u, v\}$ je bezpečná vzhledem k A.

Důsledek: Nechť $G = (V, E, h_E)$ je souvislý hranově ohodnocený neorientovaný graf, množina hran $A \subseteq E$ je obsažena v nějaké minimální kostře grafu G a nechť T je nějaká souvislá komponenta v lese (V, A). Pak každá hrana spojující T s nějakou jinou komponentou v lese (V, A) a mající minimální ohodnocení je bezpečná vzhledem k A.

Kruskalův (Borůvkův-Kruskalův) algoritmus

Je dán hranově ohodnocený neorientovaný graf $G = (V, E, h_E)$, hledáme množinu hran $A \subseteq E$ tvořící minimální kostru $T = (V, A, h_E|_A)$.

```
type SpT = Set of Edges;
     Component = Set of Nodes; {„vhodně reprezentovaná“ množina}
procedure kruskal (G:Graph; var A:SpT);
  var W: Set of Component;
  begin A := ∅;
        W := ∅;
        for v ∈ V do W := W ∪ {{v}};
        seřadíme množinu E podle ohodnocení;
        for (u, v) ∈ E {v pořadí od nejníže ohodnocené} do
            if W_u ≠ W_v {tj. u, v leží v různých komponentách}
                then begin A := A ∪ {(u, v)};
                           sjednotíme obě tyto komponenty
                     end
   end
```

Korektnost Kruskalova algoritmu

Věta: Po skončení výpočtu `kruskal`(G, A) je v A množina hran grafu G tvořících jeho minimÁlní kostru.

Důkaz vyplývÁ z předchozího Důsledku.

Složitost Kruskalova algoritmu

Složitost algoritmu zÁvisí na datovÉ struktuře reprezentující množinu komponent W. Pro reprezentaci komponent lesa můžeme použít tzv. binomiÁlní haldy. Dotaz $W_u \neq W_v$ vyjÁdříme dotazem, zda halda, v níž se nachÁzí uzel u, je shodný s haldou, v níž se nachÁzí uzel v. Tento dotaz mÁ logaritmickou složitost.

Složitost takto implementovanÉho Kruskalova algoritmu je pak $\Theta(m \log m)$, kde $m = |E|$.