Návrh algoritmů I

IB002

Doplňující text k přednášce

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/`
- zápisky z přednášek. . .

Program, algoritmus, funkce

Program je formální popis chování nějakého 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 parciální funkce z množiny vstupů do množiny výstupů, k níž existuje konečný popis v určitém formalismu.

Tímto formalismem je typicky programovací jazyk, ale existují i další matematické formalismy: Turingův stroj, RAM, lambda kalkul, kombinátorový kalkul, vyčíslitelné funkce, Markovův algoritmus . . .

Slovo *algoritmus* je odvozeno ze jména starověkého perského matematika a astronoma al-Chwarezmího (Abū ‘Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī).

Funkce z množiny vstupů do množiny výstupů, které lze vyjádřit algoritmem, se nazývají *rekursivně spočetné*.

Reaktivní systém je program, jehož provádění normálně nemusí končit (např. operační systémy 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 součástí 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* $\to$ *Bool*	vstupní podmínka
ψ : *In* $\times$ *Out* $\to$ *Bool*	výstupní podmínka

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

Výstupní podmínka ψ říká, zda daný výstupní údaj je či není výsledkem výpočtu algoritmu na daném vstupním údaji.

Vstupní podmínka je unární predikát na množině vstupních dat, výstupní podmínka je binární predikát na množinách vstupních a výstupních dat.
Výstupní podmínka ψ říká, zda daný výstupní údaj je či není (pro nedeterministické algoritmy zda může či nemůže být) výsledkem výpočtu algoritmu na daném vstupním údaji.

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

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

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

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

Poznámka: Připustíme i algoritmy, které nenačítají žádný vstup. Pro ně položíme *In* $= \{\emptyset\}$.

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

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

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

Důsledek: Algoritmus A je totálně korektní vzhledem k φ a ψ, právě 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 také říká *specifikace algoritmu*. O totálně korektním algoritmu pak říkáme, že *vyhovuje specifikaci*.

Výpočetní paradigmata

Paradigma (vzor, přístup, pojetí) udává způsob, jakým je algoritmus vyjádřen.

Výpočetní paradigmata:

- imperativní
- funkcionální
- logické

- procedurální
- objektové

- sekvenční
- paralelní

⋮

Imperativní paradigma

Neformální vymezení:

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

Jinak řečeno, příkazy programu jsou *stavové transformátory* — (parciální) funkce z množiny stavů do množiny stavů.

Pro formální definici je nutno zavést abstraktní počítač. V sekvenčním imperativním paradigmatu jím nejčastěji bývá Turingův stroj nebo RAM (Random-Access Machine).

Může jím být také „vyšší“ imperativní jazyk, který kromě elementárních stavových transformátorů (typicky tzv. *přiřazovacího příkazu*) obsahuje složené stavové transformátory realizující sekvenci (begin-end), binární větvení výpočtu (if-then-else), násobné větvení (case), opakování (while-do, for-do), apod.

Výpočetním krokem je pak přechod mezi dvěma konfiguracemi Turingova stroje, resp. provedení jedné instrukce RAMu, resp. provedení jednoho *elementárního* 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

- Spočetná posloupnost $\mathcal{M}$ registrů (paměťových míst) — tzv. *paměť*. Každý registr obsahuje celé číslo, vždy však skoro všechny obsahují nulu.
- *Stav* $\sigma \in S = \mathbb{N} \times \mathbb{Z}^*$
- Konečná množina instrukcí $\mathcal{I}$. Každá instrukce $\iota \in \mathcal{I}$ má danou sémantiku — funkci $\iota : S \dashrightarrow S$.
- Konečná posloupnost $P \in \mathcal{I}^*$ instrukcí, tzv. *program*

Příklad: Výpočet faktoriálu (funkcionální verze – Pascal)

Faktoriál nezáporného celého čísla n je číslo $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$.

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

Výpočet faktoriálu (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 parciálně 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 spočívá v nalezení číselné hodnoty (závislé na obsahu programových proměnných), která v žádném okamžiku výpočtu nemůže být záporná a přitom při každém průchodu tělem cyklu klesne nejméně o jedničku. V tomto příkladě je touto hodnotou číslo $n - i$ a důkaz nezápornosti i klesání je snadný.

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

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

Invariant cyklu

je mezilehlá podmínka, která je splněna v daném bodě výpočtu podle algoritmu Invariant cykluv každém průchodu cyklem.

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

1. Platí-li na začátku $\varphi(n)$, pak po provedení úvodních přiřazení bude splněn invariant cyklu.
2. Je-li v nějakém 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 mezilehlá podmínka za cyklem, tedy $\psi(n, k)$.

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

Příklad: Algoritmus umocňování čísla na přirozený exponent (funkcionální 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 parciálně 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 umocňování čísla 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. Klesání je tentokrát rychlejší než v předcházejícím příkladě, ale pro $\mathtt{k} > 0$ je vždy ostré.

Věta: Funkce `power` je parciálně 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. Klesání je tentokrát rychlejší než v předcházejícím příkladě, ale pro $\mathtt{k} > 0$ je vždy ostré.

Věta: Funkce `power` je parciálně 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 testování 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 této konjunkce vyplývá $\mathtt{r} = z^n$, přičemž `r` je výsledná hodnota funkce `power`. Platí tedy i výstupní podmínka $\psi(z, n, \mathtt{power(}z\mathtt{,}n\mathtt{)})$.

Příklad: Algoritmus řazení vkládáním (imperativní verze – Pascal)

```
const MAX = 999;
type Elem = Integer;
     Posl = array [1..MAX] of Elem;
procedure iInsSort (n:Integer; var A:Posl);
  var i, j : Integer;  x : Elem;
  begin
      for i := 2 to n do
      begin
          x := A[i];  j := i-1;
          while (j>0) && (A[j]>x) do
          begin
              A[j+1] := A[j];
              j := j-1
          end;
          A[j+1] := x
       end
   end
```

Věta: Algoritmus `iInsSort` je totálně korektní vzhledem k podmínkám

$\varphi(n, A) \quad \equiv \quad$ posloupnost A je neprázdná a $n \geq 1$ je její délka

$\psi(n, A, A') \quad \equiv \quad$ 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 $\varphi(n, A) \equiv n \geq 1 \wedge |A| = n$, $\psi(n, A, A') \equiv A' \in \mathit{Permut}(A) \wedge \forall i, j, 1 \leq i < j \leq n. A'_i \leq A'_j$,
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 $\mathtt{j} > 0$. Ale při každém průchodu tělem tohoto cyklu hodnota proměnné `j` klesne. To znamená, že cyklus while se provede konečněkrát (nejvýše $(\mathtt{i} - 1)$-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 daném stavu výpočtu) označme $(A_1,\ldots,A_n)$.
Invariantem vnějšího cyklu for (splněným vždy na začátku tohoto cyklu) je podmínka $A_1 \le A_2 \le \cdots \le A_{i-1} \ \wedge\ (A_1,\ldots,A_{i-1}) \in \mathit{Permut}(a_1,\ldots,a_{i-1}) \ \wedge\ A_i = a_i,\ldots,A_n = a_n$, takže po posledním průchodu cyklem for platí $A_1 \le A_2 \le \ldots \le A_n$.
Invariantem vnitřního cyklu while (opět splněným vždy na začátku cyklu) je podmínka $A_1 \le \cdots \le A_{j-1} \le A_j = A_{j+1} \le A_{j+2} \le \cdots \le A_i \ \wedge\ (A_1,\ldots,A_j,\ a_i,\ A_{j+2},\ldots,A_i) \in \mathit{Permut}(a_1,\ldots,a_i)$

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

Funkcionální paradigma

Neformální definice:

Výpočet je posloupností tzv. *redukčních kroků*, z nichž každý je elementární úpravou programu (výrazu). Výsledkem výpočtu je hodnota, kterou už nelze dále zjednodušit.

Délka výpočtu je pak délkou této posloupnosti, tj. počet redukčních kroků vedoucích k výsledku.

Pro formální definici je nutno zavést formální kalkul. Ve funkcionálním paradigmatu jím nejčastěji bývá jazyk vycházející z lambda kalkulu — *funkcionální jazyk*.

Příklad: Funkce `maxim` najde největší číslo z konečného neprázdného seznamu čísel

```
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 vkládáním (funkcionální 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 totálně korektní vzhledem k φ, ψ.

Poznámka: V Haskellu lze definici zapsat i kratčeji:

`fInsSort =  foldr ins [].`

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

Důkaz: Nechť s je konečný seznam délky n. Konvergence obou funkcí se ukáže indukcí přes délku seznamu.

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

$\vdots$

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 rozumíme časovou 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 problém: seřadit vzestupně n-prvkovou posloupnost celých čísel.
- K řešení použijeme algoritmus řazení vkládáním.
- Naše konkrétní posloupnosti, které budou vstupem, jsou $3, 6, 9, \ldots, 3n$ (tj. jsou už seřazené).

Rozlišujeme tři pojmy:

- časovou složitost problému
- časovou složitost algoritmu
- délku výpočtu

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

Časová složitost algoritmu `iInsSort` je vyjádřena kvadratickým polynomem $an^2 + bn + c$, jak později uvidíme. Je tedy větší než délka výpočtu na našich (příznivých) datech. Na jiných posloupnostech délky n stejný algoritmus stráví času více. V nejhorším případě až kvadraticky mnoho, proto je časová složitost algoritmu řazení vkládáním kvadratická.

Časová složitost problému vzestupného seřazení posloupnosti je však lepší než kvadratická: existují algoritmy, které n-prvkovou posloupnost seřadí v čase $c\,n \log n$ (c je konstanta), a to i v nejhorším případě (tj. i pro „nejzlomyslněji zadané“ vstupní posloupnosti). Proto je časová složitost problému řazení vyjádřena funkcí $c\,n \log n$ (v proměnné n).

Délka výpočtu výrazů

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

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

Vyhodnocení elementárních aritmetických, logických, relačních operací nad skalárními operandy je konstantní. Proto i délku výpočtu výrazu, který obsahuje pouze tyto jednoduché operace, považujeme za konstantní.

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

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

$\tau(\textit{op}\ a) = 1 + \tau(a)$, 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, které vyhodnocují oba svoje operandy, například $+, -, *, /, \mathtt{div}, \mathtt{mod}, <, \leq, >, \geq, \ldots$.

Pozor, některé operátory a jazykové konstrukce nemusí vždy vyhodnocovat všechny svoje operandy. Podle toho se pak určuje délka výpočtu výrazů:

$\tau(a \;\texttt{\&\&}\; b) = 1 + \tau(a) + \tau(b)$ pro a pravdivé
$\tau(a \;\texttt{\&\&}\; b) = 1 + \tau(a)$ pro a nepravdivé

$\tau(a \;\texttt{||}\; b) = 1 + \tau(a) + \tau(b)$ pro a nepravdivé
$\tau(a \;\texttt{||}\; b) = 1 + \tau(a)$ pro a pravdivé

$\tau(\texttt{if}\; a \;\texttt{then}\; b \;\texttt{else}\; c) = 1 + \tau(a) + \tau(b)$ pro a pravdivé
$\tau(\texttt{if}\; a \;\texttt{then}\; b \;\texttt{else}\; c) = 1 + \tau(a) + \tau(c)$ pro a nepravdivé

Délka výpočtu volání funkce

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

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

Bude-li nás zajímat pouze tzv. *asymptotické chování* složitosti algoritmů, budeme ztotožňovat složitosti lišící se jen kladnou multiplikativní konstantou. Pro takové případy bude užitečná následující

Úmluva: Je-li e výraz obsahující jen skalární konstanty a proměnné a elementární operace (na skalárních hodnotách), klademe délku jeho výpočtu rovnu jedné.

Výraz e zejména nesmí obsahovat vektorové operace (např. aritmetické operace na „dlouhých“ číslech) ani volání uživatelských funkcí.

Tuto úmluvu zavádíme proto, že pro účely zjišťování asymptotického chování algoritmů je praktičtější například uvažovat délku přiřazení `x:=a*(b+c+d)` rovnu jedné, než ji počítat jako $1 + 1 + 2 + 5$. Stejně víme, že součet je roven konstantě, a to nám stačí.

U tzv. „čistých“ výrazů předpokládáme, že nemají vedlejší efekty. To však nemusí vždy platit. Délka výpočtu výrazů, které obsahují volání funkcí, závisí najen 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
```

Globální přepisovatelná proměnná `a` má na začátku hodnotu 2, pak ve výrazu `f(4)+f(4)` trvá každé vyhodnocení stejného podvýrazu `f(4)` různou dobu. (Jakou?)

Délka výpočtu příkazů

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

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

Prázdný příkaz

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

Přiřazovací příkaz

$\tau(\texttt{v:=}e, \sigma) = 1 + \tau(e, \sigma)$, je-li v jednoduchá přepisovatelná proměnná

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 $\sigma' = \nu(e)(\sigma)$.

$$\tau(\mathtt{if}\ e\ \mathtt{then}\ p\ \mathtt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma') \quad \text{pro pravdivé } e,$$
$$\tau(\mathtt{if}\ e\ \mathtt{then}\ p\ \mathtt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma') \quad \text{pro nepravdivé } e.$$

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

$$\tau(\mathtt{if}\ e\ \mathtt{then}\ p\ \mathtt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma) \quad \text{pro pravdivé } e,$$
$$\tau(\mathtt{if}\ e\ \mathtt{then}\ p\ \mathtt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma) \quad \text{pro nepravdivé } e.$$

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 své tělo s.

Pro $0 \le i \le n$ označme $\sigma_i' = \nu(e)(\sigma_i)$,
pro $0 \le i < n$ označme $\sigma_{i+1} = \nu(s)(\sigma_i)$,
a nechť pro $0 \le i < n$ je $\mu(e)(\sigma_i) =$ pravda, $\mu(e)(\sigma_n) =$ nepravda. Pak

$$\tau(t, \sigma_0) = \sum_{0 \le i < n} \big(\tau(e, \sigma_i) + \tau(s, \sigma_i')\big) + \tau(e, \sigma_n)$$

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

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

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

Stejné rozšíření definice délky výpočtu se vztahuje i na výrazy s vedlejšími efekty.

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

Nechť `f` je funkce definovaná

$\texttt{f (}x_1\texttt{, ... ,}x_n\texttt{) = } e$

a σ_0 je stav, ve kterém začne výpočet.

Pak buďto, v případě volání jménem,

$\tau(\texttt{f(}a_1, \ldots, a_n\texttt{)}, \sigma_0) = 1 + \tau(e', \sigma_0)$, kde výraz e' vznikne z výrazu e (těla funkce `f`)

nahrazením každého výskytu x_i *výrazem* a_i

anebo, v případě volání hodnotou,

$\tau(\texttt{f(}a_1, \ldots, a_n\texttt{)}, \sigma_0) = 1 + \tau(a_1, \sigma_0) + \tau(a_2, \sigma_1) + \cdots + \tau(a_n, \sigma_{n-1}) + \tau(e'', \sigma_n)$,

kde $\sigma_i = \nu(a_i)(\sigma_{i-1})$ (pro $0 < i \leq n$) a výraz e'' vznikne z těla e takto:

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

Délku výpočtu příkazu cyklu for můžeme odvodit tak, že si cyklus for vyjádříme 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 právě n-krát své tělo 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

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

Většinou nás víc než délka jednoho konkrétního výpočtu zajímá, jak se daný algoritmus chová obecně, tj. jak vypadá množina délek všech možných výpočtů — pro množinu všech přípustných vstupních dat (vyhovujících vstupní podmínce).

Časová složitost algoritmu

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

Počáteční stav výpočtu, který odpovídá umístění hodnoty x na vstupu, označíme σ_x.

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

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

Časová složitost algoritmu je tedy funkce, která pro každou velikost vstupních dat je rovna délce *nejdelšího* výpočtu na všech možných datech této velikosti.

Velikost vstupních dat odpovídá počtu bitů, kterými je vstupní údaj vyjádřen. V praxi má často tyto významy:

číslo n	délka jeho bitového zápisu, tj. $\lceil \log_2 n \rceil$
posloupnost čísel	počet jejích prvků
graf	počet uzlů $+$ počet hran

Protože množina v definici složitosti je většinou nekonečná, musíme hledané 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 vkládáním (imperativní verze – Pascal)

```
const MAX = 999;
type Elem = Integer;
     Posl = array [1..MAX] of Elem;
procedure iInsSort (n:Integer; var A:Posl);
  var i, j : Integer;  x : Elem;
  begin
      for i := 2 to n do
      begin
          x := A[i];  j := i-1;
          while (j>0) && (A[j]>x) do
          begin
              A[j+1] := A[j];
              j := j-1
          end;
          A[j+1] := x
       end
   end
```

Příklad: Analýza nejhoršího případu

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

Označme c_i délku výpočtu příkazu nebo testu na i-tém řádku algoritmu `iInsSort`, přičemž na sedmém řádku (v záhlaví cyklu for) jsou tři elementární akce, $c_{7,\text{init}}$, $c_{7,\text{test}}$, $c_{7,\text{inc}}$.

Pak délka výpočtu je

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

kde ξ je hodnota, při které se vnitřní cyklus algoritmu zopakuje nejvícekrát, tj. nejmenší možná hodnota `j`, při níž je ještě splněna podmínka za while (na 10. řádku algoritmu).

Takové minimální možné ξ je zřejmě rovno 1, a to v každém průchodu vnějším cyklem for. Situace, kdy v každém 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 porovnávání složitostí algoritmů abstrahovat od rychlosti konkrétního počítače, budeme chtít srovnávat funkce podle toho, „jak rychle rostou“, přičemž za významný rozdíl v rychlosti růstu nepovažujeme například případ, kdy jedna funkce má c-krát větší hodnoty než druhá (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 :

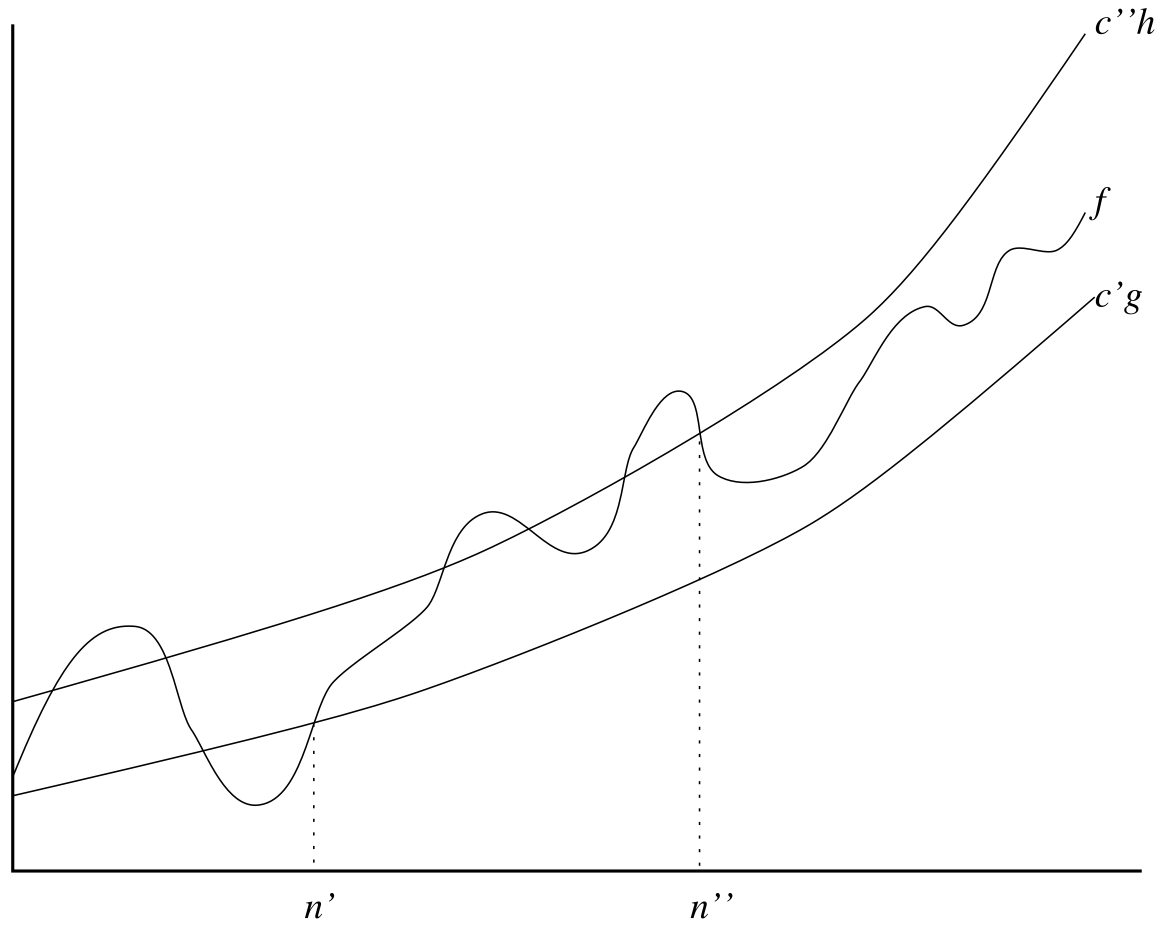
$$\Theta(g) = O(g) \cap \Omega(g)$$

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 (která je vždy nezáporná), omezíme se dále na nezáporné 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 obrátit nerovnost a uvážit kladnou konstantu $\dfrac{1}{c}$.

Třetí tvrzení je triviální.

Čtvrté tvrzení říká, že jistá vlastnost platí pro všechna kladná c. Ale kladná čísla existují, například $c = 1$. Tedy platí také slabší vlastnost pro *nějaké* $c > 0$.

Důkaz pátého tvrzení je analogický.

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

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

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

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

Rovnost $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty$ je ekvivalentní s podmínkou $\forall c > 0 \exists n_0 \forall n \geq n_0 . \left| \dfrac{f(n)}{g(n)} \right| > c$. Obě funkce jsou kladné, tedy absolutní hodnotu nemusíme uvažovat a nerovnost lze vynásobit 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 kladné funkce, pak platí

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

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

Obrácené implikace však obecně neplatí, protože uvedené 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 kladné funkce a nechť funkce $q : \mathbb{N} \to \mathbb{N}$ je definována $q(n) = f(n) + g(n)$.

Pak platí:

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

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

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

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

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

Poznámka: Platí řada podobných odvozených vlastností, například

$$\begin{aligned} f \in \Omega(h) \wedge g \in O(h) &\Rightarrow q \in \Omega(h) \\ f \in \Theta(h) \wedge g \in O(h) &\Rightarrow q \in \Theta(h) \\ f \in \Theta(h) \wedge g \in o(h) &\Rightarrow q \in \Theta(h) \end{aligned}$$

apod.

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

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

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 kladné konstanty v definicích tříd O, Ω, Θ, o, ω. Pro obecné $d \neq 0$ jde o speciální 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.

Poznámka: *Předuspořádání* (též *polouspořádání*) je binární relace, která je reflexivní a transitivní. *Uspořádání* je binární relace, která je reflexivní, antisymetrická a transitivní. *Ekvivalence* je binární relace, která je reflexivní, symetrická a transitivní.

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

Cvičení: Dokažte tvrzení:
Je-li kladná funkce f skoro všude majorizována 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 kladné polynomy stejného stupně rostou stejně rychle.

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

Věta: Exponenciální funkce se základem $a > 1$ roste rychleji než libovolný polynom. Logaritmické funkce rostou pomaleji než funkce lineární.

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

$$\mathsf{log}_a \in \Theta(\mathsf{log}_b)$$

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

Poznámka: Roste-li funkce logaritmicky, pak na základu logaritmu nezáleží. (Jen musí být větší než 1, což je však nutná podmínka k tomu, aby funkce vůbec rostla.)

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

Důkaz plyne z definice faktoriálu.

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

Poznámka: 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 faktoriálu:

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

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 $f \in \Omega\left(a^{a^n}\right)$.

Poznámka: Existují funkce, které rostou rychleji než všechny k-násobně exponenciální funkce, a to dokonce pro libovolné k.
Naopak existují kladné funkce, které rostou (tedy rychleji než konstantní), ale rostou pomaleji než funkce $\log \circ \cdots \circ \log$, kde logaritmus můžeme složit sama se sebou libovolněkrát.

Def: *Fibonacciho funkce* $F : \mathbb{N} \to \mathbb{N}$ je definovaná rekursivně:

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

pro všechna $n \in \mathbb{N}$.

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

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

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

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

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

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

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

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

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

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

Dále 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 délky výpočtů podle nějakého algoritmu A jsou v množině $\Theta(g)$, právě když časová složitost algoritmu A je v $O(g)$ a délky nejkratších výpočtů patří do $\Omega(g)$.

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

Algoritmus řazení slučováním (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 totální korektnosti: Konvergence i parciální korektnost se dokáže indukcí podle délky seznamu s, parciální korektnost pomocné funkce `merge` se dokáže indukcí podle součtu délek obou slučovaných seznamů.

Lineární složitost funkce `merge` je zřejmá: konstantní délka porovnání a připojení menšího prvku na začátek posloupnosti se dohromady provede tolikrát, kolik je součet délek obou vstupních posloupností (argumentů funkce `merge`).

Odvození složitosti funkce `mergeSort` je na následujících stranách.

Algoritmus řazení slučováním (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 }
```

Poznámka: 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 ;
```

Časová složitost algoritmu řazení slučováním

Analýza nejhoršího případu

Pro danou velikost n je délka výpočtu vždy stejná.

Časová 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 triviální: pro každou vstupní posloupnost délky n algoritmus dělá stejnou práci. Pro složitost $T : \mathbb{N} \to \mathbb{N}$ tedy platí, že $T(n)$ je rovno délce výpočtu na *libovolné* vstupní posloupnosti délky 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 volání), tedy zejména pomocné procedury `merge`.

Procedura `merge` je tvořena cyklem s pevným počtem opakování (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 upravená 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 dostáváme

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

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 speciálním 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 kladná funkce. Například pro algoritmus řazení slučováním je $a = 2, b = 2, f \in \Theta(n)$, tedy případ „2“ následující věty.

Věta: Nechť $a \geq 1, b > 1$, f je kladná 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 konečné posloupnosti celých čísel, a takový, že jeho výsledkem je neklesající permutace vstupní posloupnosti.

Poznámka: Asociativní řadicí algoritmus tedy nemůže předpokládat nic o velikosti řazených prvků a jedinou možností testování je srovnávat prvky mezi sebou.

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

Poznámka: Věta říká, že *dolní odhad* (časové) složitosti problému asociativního řazení je v množině $\Omega(n \log n)$. Protože však známe konkrétní řadicí algoritmy s touto složitostí, vidíme, že tento odhad je dokonce *těsný*: Časová složitost problému asociativního řazení je v množině $\Theta(n \log n)$.

Např. složitost řazení vkládáním, výběrem, bublánním 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 řazené 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 téhož 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 $\log(n \cdot \log n) \in \Theta(\log n)$.

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

Algoritmus řazení rozdělováním

(funkcionální 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 totálně korektní vzhledem k podmínkám φ, ψ.

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

Důkaz korektnosti: Konvergence i parciální korektnost se ukáže indukcí vzhledem k délce seznamu s.

Řazení rozdělováním (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 rozdělováním (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 rozdělováním má složitost v množině $\Theta(n^2)$.

Poznámka: *Průměrná* délka výpočtu podle algoritmu řazení rozdělováním (tzv. *průměrná* nebo *očekávaná časová složitost*) je v množině $\Theta(n \log n)$.

Důkaz věty o kvadratické složitosti: Není těžké zjistit, že délka výpočtu cyklu (repeat) je $c \cdot n$ (celé pole se projde jedním průchodem a výměna prvků trvá konstantní dobu).

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

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

Při nevyváženém dělení posloupnosti na podposloupnosti délek $n-1, 0$ je délka výpočtu kvadratická. 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) – druhá derivace $T(n)$ vzhledem ke q pro kvadratický odhad $T(n) \leq cn^2$ je kladná.
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, $0 \le r \le 1$. 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 $\Theta(n \log n)$.

Prostorová složitost řadicích algoritmů

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

Def: Prostorová složitost algoritmu A je funkce, která pro každou velikost vstupních dat je rovna velikosti obsazené paměti při výpočtu, jenž této paměti spotřebuje nejvíc.

Poznámka: Označíme-li $\rho(A, \sigma_x)$ množství paměti spotřebované výpočtem podle algoritmu A z počátečního stavu σ_x (odpovídajícího umístění dat x na vstupu), potom prostorová 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 sama řazená posloupnost, a ta má délku n. Ostatní data (mimo tuto posloupnost, tj. „extrasekvenční“) mají v uváděných algoritmech velikost nejvýše $O(n)$.

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

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

Řazení rozdělováním není *in situ*.

Věta: První varianta algoritmu `quickSort` má extrasekvenční prostorovou složitost $\Theta(n)$, druhá (modifikovaná) varianta tohoto algoritmu má extrasekvenční prostorovou složitost $\Theta(\log n)$.

Poznámka: Hovořit o řadicích algoritmech, zda jsou in situ, a zavádět jejich extrasekvenční prostorovou složitost má praktický význam jen tehdy, je-li posloupnost reprezentována polem a uložena v souvislém úseku paměti. Při jiných datových reprezentacích posloupnosti (např. seznamem) se extrasekvenční prostorová složitost nezavádí.

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:

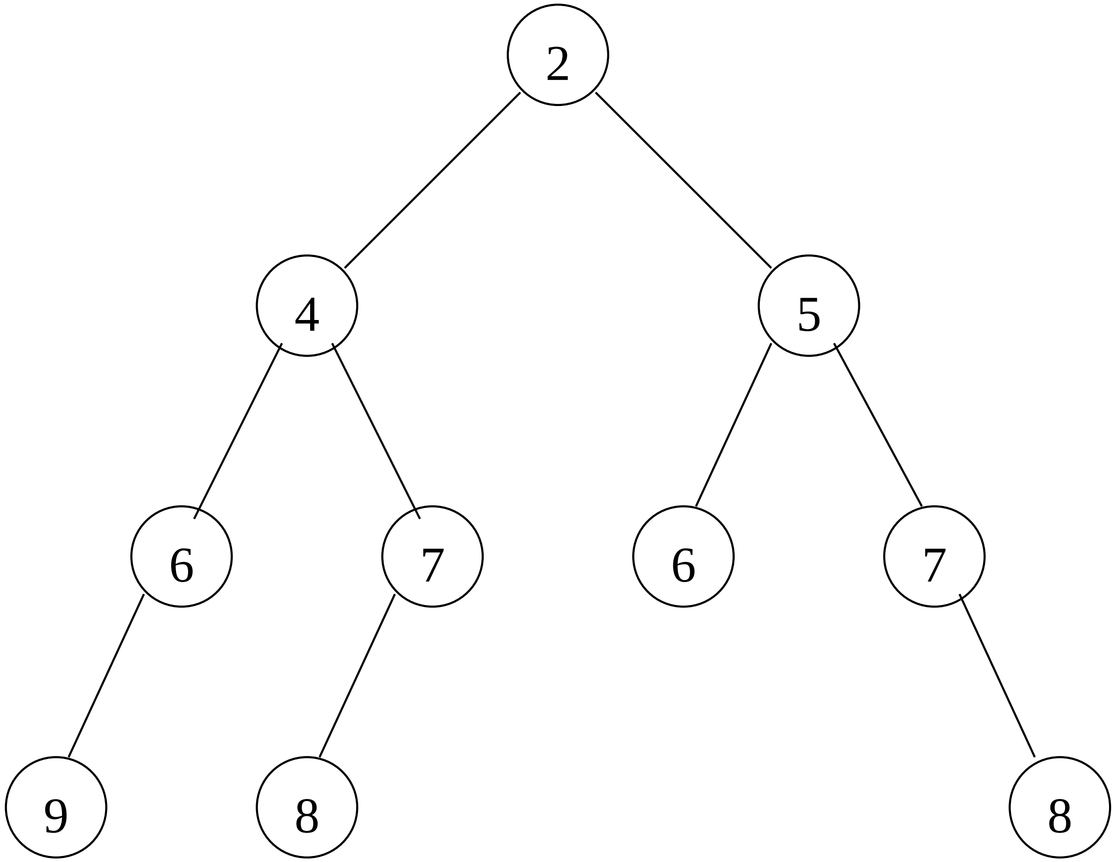
1. Délky všech větví se liší nejvýše o 1: mají délku k, případně $k-1$. Číslu k pak říkáme *hloubka* haldy.

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 funkcionální implementaci algoritmu řazení haldou, kde se pracuje s explicitní haldou (datovou strukturou binární 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 binární haldou

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

Základní 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 práci s binární haldou

Vyhledání minimálního prvku (v minimové haldě)

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

$\mathtt{minH} = \mathtt{rootVal}$

(složitost $\Theta(1)$)

Odstranění minimálního prvku (z minimové haldy)

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

(složitost $\Theta(\log n)$)

Odstranění maximálního prvku (z maximové haldy)

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

(složitost $\Theta(\log n)$)

Přidání prvku

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

(složitost $\Theta(\log n)$)

Odstranění prvku

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

(složitost $\Theta(\log n)$)

Věta: Neprázdná binární halda na n uzlech má hloubku $\lfloor \log_2 n \rfloor$.

Důkaz indukcí podle n.

Def: Binární strom, v jehož každém podstromu se počet uzlů levého a pravého podstromu liší nejvýše o jednu, se nazývá *uzlově vyvážený* binární strom.

Věta: Každý uzlově vyvážený binární strom je haldou (až na ohodnocení uzlů).

Důkaz: Je třeba dokázat, že délky všech větví uzlově vyváženého stromu se liší nejvýše o jednu.
Předpokládejme sporem, že tvrzení neplatí, a T je nejmenší strom, který tvrzení porušuje. Tedy v každém podstromu stromu T se počet uzlů vlevo a vpravo liší nejvýše o jednu, ale existují zde dvě větve délek m a k, přičemž $m \geq k + 2$. Z minimality stromu T plyne, že jedna z těchto dvou větví, řekněme ta delší, délky m, leží v levém podstromu T_l stromu T, a druhá, kratší, délky k, v pravém podstromu T_r stromu T. Navíc, rovněž z minimality, všechny větve jdoucí do T_l mají délku m nebo $m - 1$, všechny větve jdoucí do T_r mají délku k nebo $k + 1$. To znamená, že T_l má aspoň o dva uzly víc než T_r. Ale podle předpokladu věty má nejvýše o jeden uzel více, což je spor.

Algoritmus řazení binární haldou (funkcionání 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 totálně korektní řadicí algoritmus.

Konvergence algoritmu je zřejmá. Důkaz parciální korektnosti vyplývá z následujících lemmat.

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

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

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

Důkaz prvního lemmatu indukcí podle velikosti haldy.

Důkaz druhého lemmatu: indukcí podle počtu prvků v h se ukáže, že binární strom `insHeap` u h bude opět uzlově vyvážený.
Podle věty o binárních stromech vyvážených vzhledem k počtu uzlů bude výsledek opět halda.
Uspořádání hodnot podél větví vyplývá z definice funkce `insHeap`.

Třetí lemma indukcí podle délky seznamu a z definice kombinátoru `foldr`.

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

Složitost řazení haldou

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

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

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

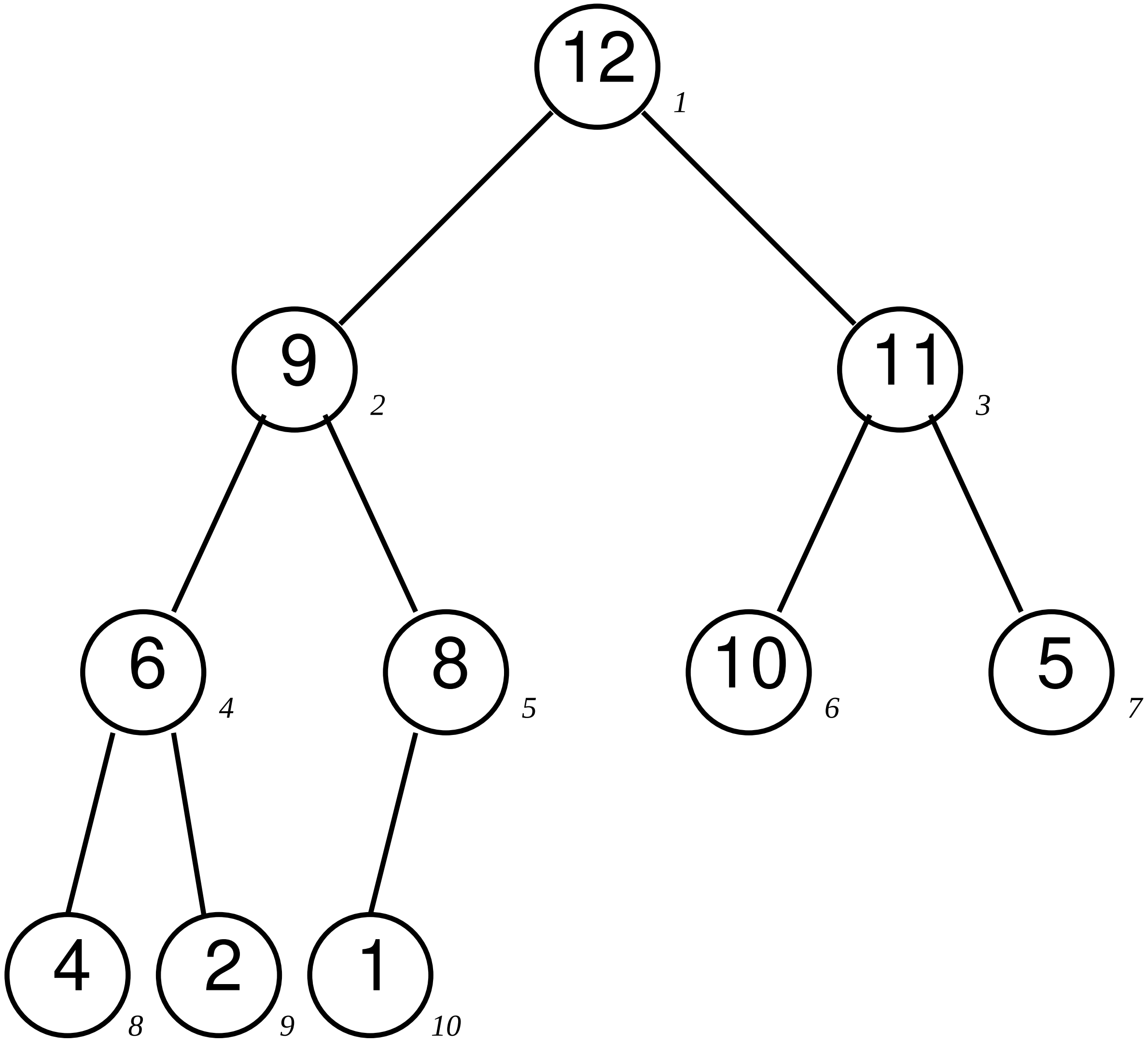
Důkaz: Z předchozího lemmatu vyplývá, že složitost funkce `toHeap` je v $O(n \log n)$. Ve funkci `toList` je hloubka rekursivního zanoření logaritmická, protože seznamy se půlí. Složitost pomocné funkce `merge` je lineární a úhrnná délka výpočtu všech volání funkce `merge` ve stejné hloubce k (tj. se stejně dlouhými seznamy) je $\frac{n}{2^k} \cdot 2^k = n$. Funkce `toList` má tedy složitost také 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 používá maximová halda, tj. binární halda, která má položky ve větvích seřazeny *sestupně*. Tedy největší položka je vždy v kořeni.

Jelikož algoritmus má haldu efektivně reprezentovánu polem, je zde výhodnější uvažovat binární haldu nikoliv uzlově vyváženou, ale naopak s levým podstromem „těžším“.

Def: *Vlevo zarovnaná halda* je binární halda, jejíž všechny větve délky k leží nalevo od větví délky $k - 1$.

Vlevo zarovnaná halda



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

Věta umožňuje pracovat s vlevo zarovnanou binární haldou reprezentovanou v paměti polem. Vlevo zarovnaná binární halda je totéž jako pole rozepsané „po vrstvách“.

Algoritmus řazení binární 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 binární 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 $\texttt{createHeap}(a, l, r)$ splňuje podmínky (vlevo zarovnané) binární haldy.

Důkaz indukcí podle hloubky haldy.

Věta: Provedení procedury $\texttt{heapSort}(A)$ seřadí posloupnost A

Idea důkazu: Jednoprvkové podstromy reprezentované druhou polovinou posloupnosti A jsou triviální podhaldy. První fáze algoritmu vytváří z těchto podhald větší podhaldy. Na konci první fáze je posloupnost A binární haldou.

Druhá fáze algoritmu vytvoří z haldy seřazenou posloupnost. To vyplývá z toho, že největší prvek haldy je vždy v jejím kořenu: největší prvky se skládají na konec posloupnosti.

Věta: Algoritmus řazení haldou konverguje.

Důkaz: Tvrzení věty vyplývá z konečné velikosti a hloubky haldy.

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

Důkaz: První fáze (vytváření haldy) má složitost $O\left(\frac{n}{2} \log n\right)$, druhá fáze má 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.

Datové struktury

Typy rozlišujeme *datové* a *funkcionální*.

Datové typy lze zavést pomocí funkcionálních, případně funkcionální typy lze simulovat pomocí nekonečných datových typů, ale kvůli efektivnosti implementace se rozlišují a uvažují se zvlášť.

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

Hodnoty datových typů jsou obvykle známy 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 (celá nebo desetinná 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, statická pole,...) — tzv. *statické datové struktury*
- Datové struktury proměnné velikosti — tzv. *dynamické datové struktury*

Statické datové struktury mají konstantní velikost a časová složitost zpřístupnění libovolného prvku je konstantní.

Dynamické datové struktury mají velikost danou nějakou proměnnou n, jejíž hodnota se může měnit. Časová složitost zpřístupnění libovolného prvku dynamické datové struktury je neklesající funkcí závislou na n; často lineární, u efektivních datových struktur lepší, typicky logaritmická.

Dynamické datové struktury

Seznam

Seznam nad bázovým typem $\mathbb{B}$ je lineární datová struktura typu $\mathbb{S}$ s následujícími operacemi:

$\mathtt{nil} : \mathbb{S}$

$\mathtt{cons} : \mathbb{B} \times \mathbb{S} \to \mathbb{S}$

$\mathtt{head} : \mathbb{S} \dashrightarrow \mathbb{B}$

$\mathtt{tail} : \mathbb{S} \dashrightarrow \mathbb{S}$

$\mathtt{null} : \mathbb{S} \to \mathtt{Bool}$

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

Axiomy seznamu

$$\begin{aligned}
\texttt{null(nil)} &= \texttt{True} \\
\texttt{null(cons(}x\texttt{,}s\texttt{))} &= \texttt{False} \\
\texttt{head(cons(}x\texttt{,}s\texttt{))} &= x \\
\texttt{tail(cons(}x\texttt{,}s\texttt{))} &= s
\end{aligned}$$

Zásobník

Zásobník nad bázovým typem $\mathtt{B}$ je lineární datová struktura typu $\mathtt{S}$ s následujícími operacemi:

$\mathtt{empty} : \mathtt{S}$

$\mathtt{push} : \mathtt{B} \times \mathtt{S} \rightarrow \mathtt{S}$

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

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

$\mathtt{isempty} : \mathtt{S} \rightarrow \mathtt{Bool}$

Pro tyto operace a každou hodnotu x typu B a každý zásobník s typu S musí platit tzv.

Axiomy zásobníku

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

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

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

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

Jak je vidět, zásobník a seznam je tatáž datová struktura.

Liší se pouze v použití:

O *zásobníku* obvykle mluvíme, když na něm používáme jen základní operace a pracujeme jen s jedním zásobníkem nebo pevně daným malým počtem zásobníků. Zásobníky bývají dány předem: během výpočtu se mění jejich obsah, ale nemění se počet zásobníků, tj. neruší se a nevznikají nové zásobníky. Proto se v imperativních jazycích základní operace často implementují jako procedury; navíc jejich parametr zásobník se vynechává, pracuje-li se jen s jedním zásobníkem.

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

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

Fronta

Fronta nad bázovým typem $\mathtt{B}$ je lineární datová struktura typu $\mathtt{Q}$ s následujícími operacemi:

$\mathtt{empty} : \mathtt{Q}$

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

$\mathtt{enqueue} : \mathtt{B} \times \mathtt{Q} \rightarrow \mathtt{Q}$

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

$\mathtt{isempty} : \mathtt{Q} \rightarrow \mathtt{Bool}$

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

Axiomy fronty

$$\mathtt{isempty(empty)} \;=\; \mathtt{True}$$

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

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

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

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

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

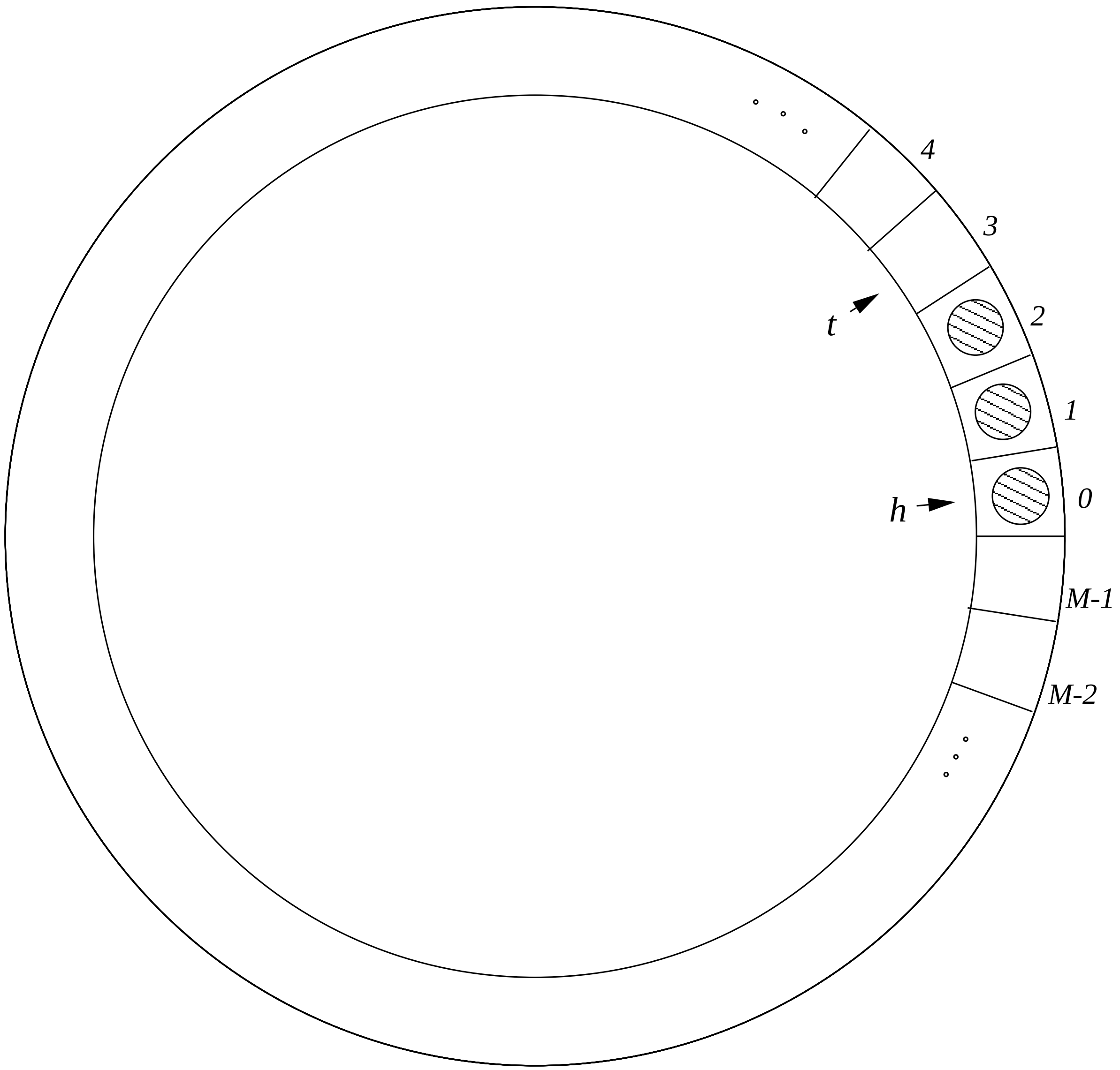
Implementace datových struktur

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.

Poznámka: Jazyky s malou podporou dynamických datových struktur (Pascal, C,. . .) často poskytují alespoň dynamickou datovou strukturu „nízké úrovně“: *paměť* $\mathtt{M}$ spolu
s typem *ukazatel* (adresa) $\mathtt{P}$,
nulární operací $\mathtt{nil:P}$ prázdného ukazatele
a unární operací zpřístupnění (dereferencování) části paměti $\mathtt{deref:P} \dashrightarrow \mathtt{M}$.

Pak například seznam se běžně implementuje pomocí záznamů v $\mathtt{M}$, jejichž složky jsou ukazatele na další záznamy, apod.



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

Příklad: Implementace ohraničené fronty pomocí cyklického 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 při aplikaci na prázdnou frontu. Doplňte jejich definice tak, aby jejich aplikace na prázdnou frontu způsobila chybové hlášení. Využijte haskellovské funkce `error :: String` $\rightarrow$ `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.

Amortizovaná časová složitost

Def: Nechť f je operace na dané datové struktuře D velikosti (nejvýše) n. Uvažujme všechny výpočty $C_1, \ldots, C_m$ podle algoritmů pracujících s datovou strukturou D. Z každého takového výpočtu vybereme všechna volání operace f (tj. všechny aplikace f na D), čímž dostaneme m posloupností volání operace f. Průměrnou délku výpočtu operace f v i-tém výpočtu označíme τ_i^{av}. Potom *amortizovaná složitost* operace f na datové struktuře D je funkce $T^{\mathsf{amort}} : \mathbb{N} \to \mathbb{N}$ definovaná pro každou velikost dat n takto

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

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

Každé „drahé“ 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) nějaké 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 drahé 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 orientovány 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 $u_1, \ldots, u_k$ nazýváme *(bezprostředními) následníky* uzlu u.

Na bezprostředních následnících každého uzlu často bývá zavedeno úplné uspořádání. Pak se následníci uzlu znázorňují zleva doprava; u binárních stromů se tedy rozlišuje *levý* a *pravý* následník.

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

Stromy pevné arity (s omezeným větvením)

Def: Je dáno přirozené číslo n a tzv. bázová množina (typ) B. Definujeme n*-ární strom* nad B takto:

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

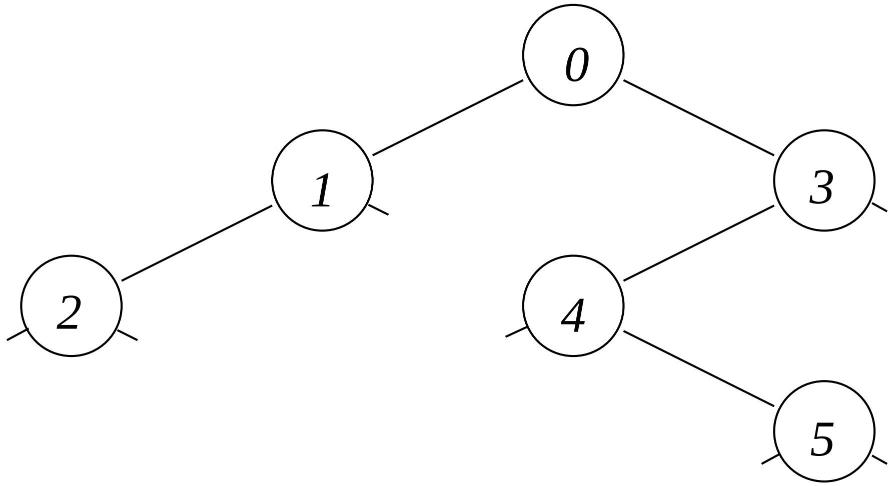
Listy n-árního stromu jsou uzly tvaru $(x, \emptyset, \ldots, \emptyset)$

Def: Cestu z kořene stromu do uzlu, jehož aspoň jeden bezprostřední následník je prázdný strom, se nazývá *větev* stromu.

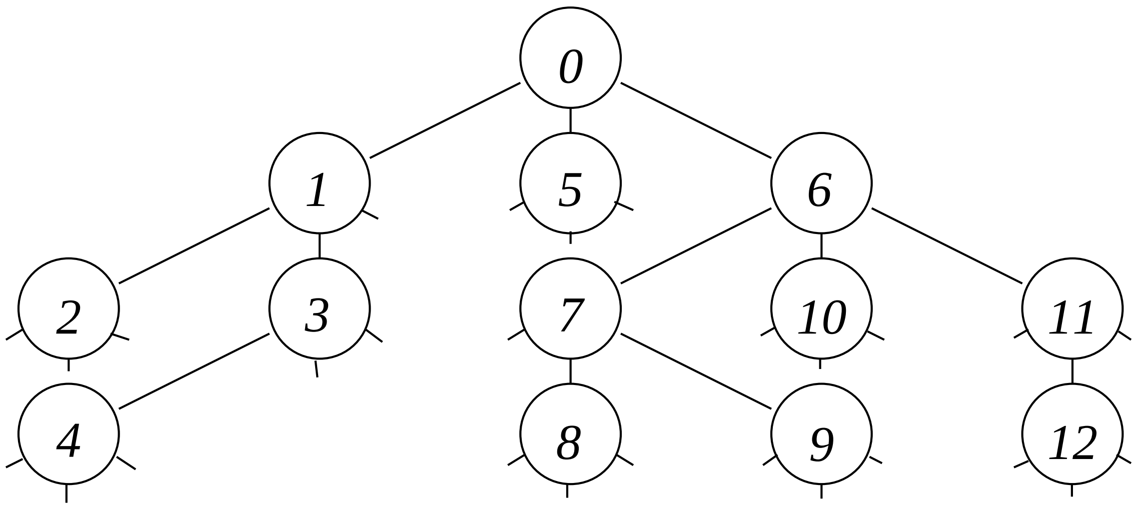
Opovídající datový typ zapsaný v Haskellu např. pro $n = 3$ je

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

Binární strom (s pevným pořadím následníků)



Ternární strom (s pevným pořadím následníků)



Binární stromy

Def: *Binární strom* je strom arity 2.

Základní operace nad binárním stromem

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

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

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

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

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

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

Poznámka: Základní operace mají konstantní složitost, slouží k popisu datové struktury a k její implementaci a k implementaci dalších operací, jako například vyhledání / přidání / odebrání uzlu, vyvážení stromu a podobně.

Pro základní operace, každou hodnotu x typu B a každé binární stromy l, r typu T musí platit

Axiomy binárního stromu

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

Implementace binárních 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 dána bázová množina (typ) B. Nechť $b \in B$. Pak pro každé přirozené číslo $k \geq 0$ a každou k-prvkovou posloupnost neprázdných stromů nad B je dvojice $(b, [T_1, \ldots, T_k])$ neprázdným *stromem s neomezeným větvením* nad B.

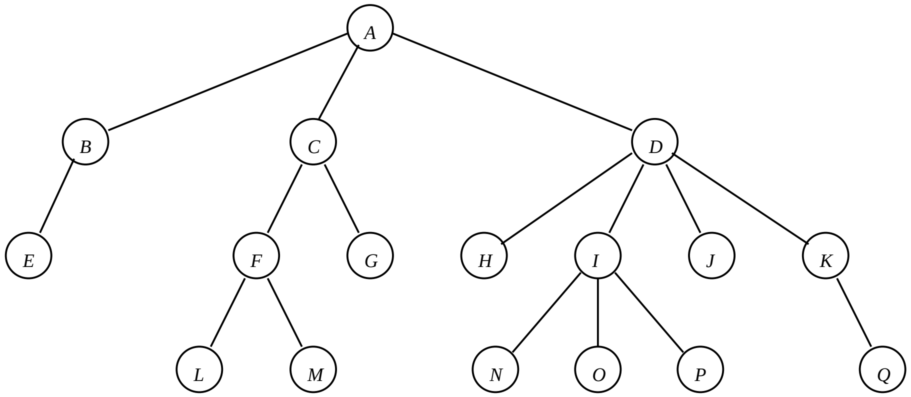
Poznámka: Druhou složkou uspořádané dvojice $(b, [T_1, \ldots, T_k])$ je k-prvková posloupnost stromů, tj. seznam délky k. Je-li $k = 0$, je seznam prázdný a dvojice reprezentuje jednouzlový strom (list) s ohodnocením b.

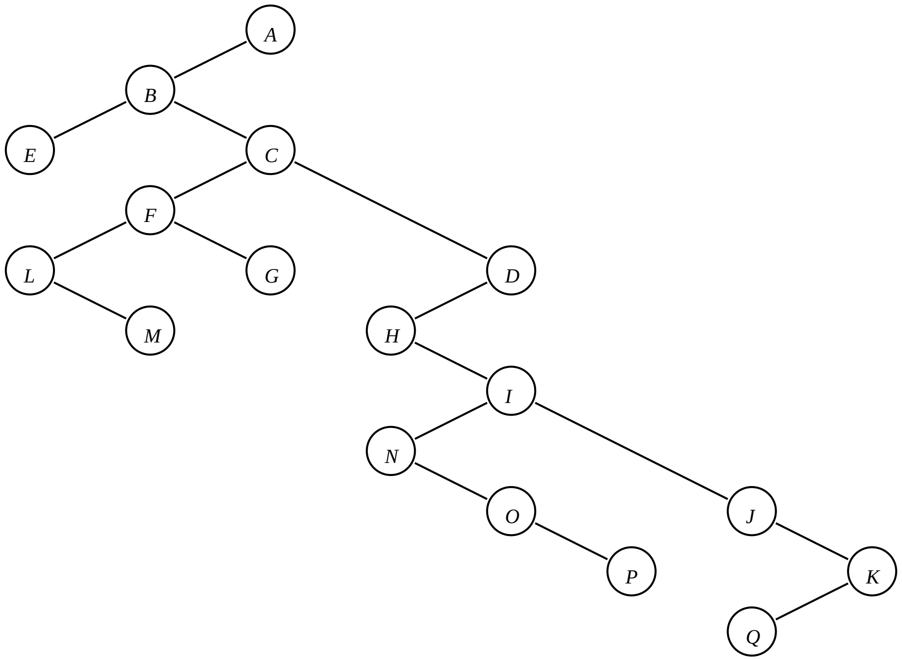
Stromy s neomezeným větvením se od stromů pevné arity liší tím, že číslo k není předem pevně dané, ale může být v jednom stromu pro různé uzly různé.

Poznámka: Stromy s neomezeným větvením lze reprezentovat binárními stromy.

Stromy s neomezeným větvením a binární 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

Vyhledávání

Def: Nechť $(K, \leq)$ je úplně uspořádaná množina tzv. *klíčů*, V je libovolná množina tzv. doplňujících údajů. Nechť $U = K \times V$ je množina dvojic (k, v), v níž se každý klíč k vyskytuje nejvýše jednou (tj. každý záznam z množiny U je určen jednoznačně svým klíčem).

Problému nalézt k danému klíči k záznam $(k, v) \in U$ se říká *vyhledávací problém*.

Poznámka: Například záznamy mohou být osobní data a klíči jsou rodná čísla, anebo záznamy jsou údaje o knihách v knihovně a klíče jsou knihovní signatury apod.

Poznámka: V praxi má většinou smysl pouze případ, kdy množina V je netriviální, aby bylo co hledat.

V ukázkových algoritmech se však doplňující údaje často neuvažují, tj. vyhledávají se jen klíče. Příslušné rozšíření těchto algoritmů je totiž přímočaré.

Algoritmus binárního vyhledávání

```
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 binárního vyhledávání má logaritmickou časovou složitost, tj. jeho složitost je v $\Theta(\log n)$, kde n je délka prohledávaného pole.

Poznámka: Extrasekvenční paměťová složitost algoritmu je konstantní, protože rekursivní volání v něm jsou *prostá*.

Cvičení: Implementujte algoritmus binárního vyhledávání bez použití rekurse.

Cvičení: Dokažte totální korektnost algoritmu binárního vyhledávání.

Hašování

O množině K všech možných klíčů obecně předpokládáme jen to, že na ní existuje úplné uspořádání. Někdy však tato množina může mít další speciální vlastnosti, jichž lze využít pro efektivní vyhledávání. Například je-li $K = \{1, \ldots, m\}$ a číslo m je malé, můžeme data z množiny V uložit do jednorozměrného pole a klíče využít jako indexy. Vyhledávání v takovéto datové struktuře je efektivní – stejně rychlé jako indexování 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 důležité, aby výpočet hodnot funkce h byl rychlý, v ideálním případě aby měl konstantní časovou složitost. Funkci h nazýváme *hašovací funkcí* a pole indexované jejími hodnotami nazýváme *hašovací tabulkou*.

Hašovací tabulky

Hašovací tabulka je datová struktura, pomocí níž lze prakticky efektivně realizovat „slovníkové“ operace vyhledání, přidání a zrušení položky.

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

Hašovací tabulka je jednorozměrné pole H indexované čísly $1, \dots, n$. Převod klíčů na čísla realizuje *hašovací funkce* $h : K \to \{1, \dots, n\}$. Výpočet hodnot hašovací funkce musí být efektivní, nejlépe 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 ukládat do n-prvkového 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 následující podmínka. Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, která budou skutečně uložena (tedy $|K'| \leq n$). Pak požadujeme, aby z každé množiny klíčů, které hašovací funkce zobrazí na stejný index, byl v množině K' nejvýše jeden klíč. Má-li pro pevně danou množinu K' klíčů skutečně uložených dat hašovací funkce tuto vlastnost, nazýváme ji *dokonalou hašovací funkcí pro klíče z* K'. Výhodou tabulek s dokonalými hašovacími funkcemi je optimální složitost vyhledání, vložení i zrušení prvku; je stejná jako pro pole.

Označme K' podmnožinu klíčů, $K' \subseteq K$, těch dat, která budou skutečně uložena. Klíče z množiny K' nazveme *použité klíče*.

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

Věta: Má-li výpočet hašovací funkce konstantní složitost a hašování je dokonalé pro množinu použitých klíčů K', pak operace vyhledání, vložení, resp. zrušení prvku v hašovací tabulce mají stejnou časovou složitost, jako vyhledání, 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 ukládaná 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 pole, ale 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 $\dfrac{1}{n}$.

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 $\{0, \ldots, n-1\}$.

Binární vyhledávací stromy

Def: *Binární vyhledávací strom* je binární strom nad úplně uspořádanou 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 binárními vyhledávacími stromy

Datový typ

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

Zjišťování 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
```

Vyhledávání

```
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
```

Vkládání 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 vyhledávací 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 binární strom vzniklý vložením resp. zrušením uzlu z vyhledávacího stromu pomocí funkcí `insert` resp. `delete` bude opět vyhledávací.

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 vyhledávání ve vyhledávacích stromech

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

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

kde h je hloubka vyhledávacího stromu, h' je hloubka jeho levého podstromu, h'' je hloubka jeho pravého podstromu a c, c' jsou konstanty.

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

Složitost vyhledávání ve vyhledávacích stromech

Složitosti operací vyhledání, přidání a zrušení položky ve vyhledávacím stromě jsou přímo úměrné hloubce stromu. Ta je v nejhorším případě lineárně závislá na velikosti stromu (počtu jeho uzlů).

Složitost vyhledávání, vkládání a rušení v obecném vyhledávacím stromě je tedy lineární.

AVL stromy

Nazvané podle G. M. Adeľsona-Veľského a E. M. Landise.

Def: Vyhledávací binární strom je *AVL*, když hloubka levého a pravého podstromu libovolného uzlu se liší nejvýše o jednu.

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

AVL stromy tedy mají logaritmickou hloubku — použijeme-li je jako vyhledávací stromy, pak má operace vyhledání položky logaritmickou složitost.

Def: *Fibonacciho strom* řádu 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 *minimální* (vzhledem k počtu uzlů) AVL strom hloubky $k-1$.

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

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

Minimalita je důsledkem předchozích dvou bodů: odebráním nějakého uzlu z jiné než nejpravější větve by někde vznikly sousední podstromy s hloubkami lišícími se aspoň o dvě; odebráním uzlu z nejpravější větve by se snížila hloubka celého stromu.

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

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 každého binárního stromu velikosti n je $\Omega(\log_2 n)$.

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

Operace na AVL stromech

Datová struktura:

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

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

Vyhledávání v AVL stromu se neliší od vyhledávání v běžném vyhledávacím stromu.

Po přidání nebo odebrání položky ovšem může nastat situace, že vyhledávací strom přestane splňovat podmínku AVL. Pak je nutné pozměnit strukturu stromu.

Přidávání položky (nového uzlu) do AVL stromu

Stejné jako u běžného vyhledávacího stromu, ale s kontrolou vyváženosti.

Přidávaný uzel označíme x.

Pokud se poruší vyváženost stromu, nalezne se nejmenší podstrom F, který je nevyvážený. Označíme-li h jeho hloubku před přidáním uzlu x, bude po přidání jeho hloubka $h + 1$. Kořen stromu F označíme f.

Bez újmy na obecnosti lze předpokládat, že uzel x je přidáván do *levého* podstromu stromu F. Nechť B je levý podstrom stromu F, G je pravý podstrom stromu F.

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

Rozlišíme dva případy:

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

   Vytvoříme strom

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

Dva dílčí podpřípady jsou:

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

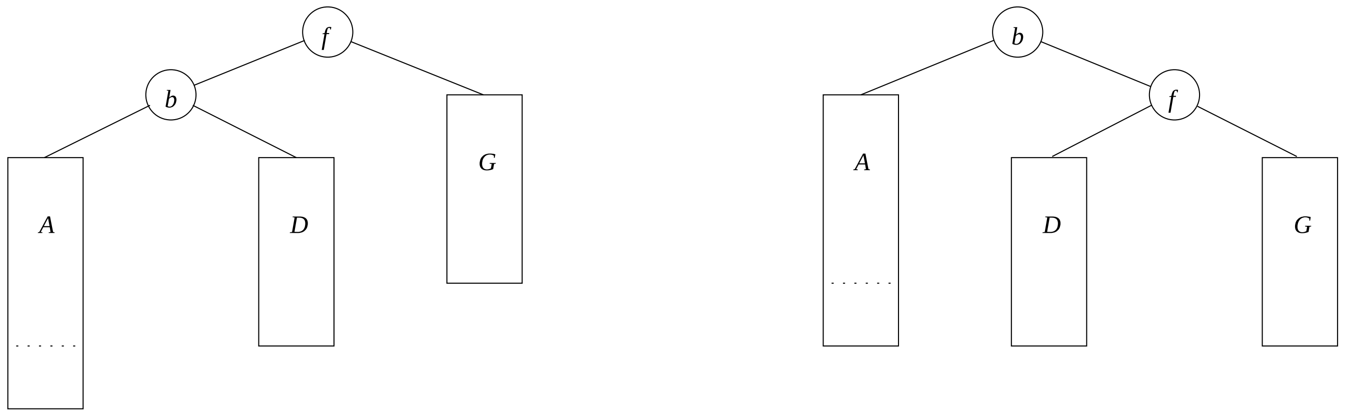
(b) Uzel x byl přidán do stromu E. Pak C má hloubku $h - 3$, E má hloubku $h - 2$.

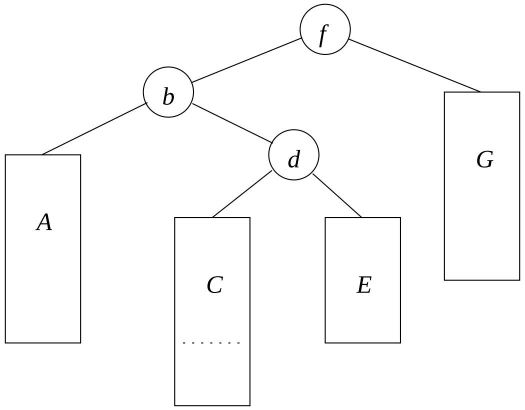
V obou případech však strom T má hloubku h, je AVL a má stejné uzly jako strom F.

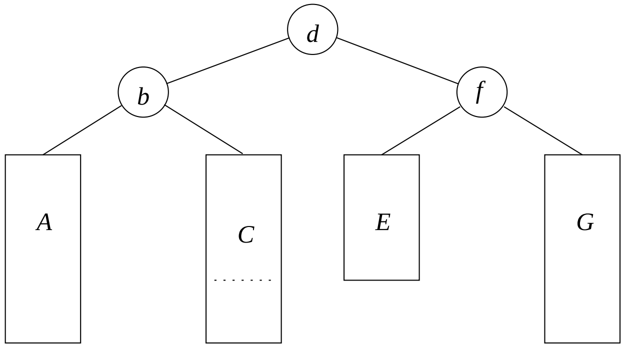
V původním AVL stromu nahradíme podstrom F stromem T. Jelikož T má stejnou hloubku jako měl původní podstrom bez uzlu x, zůstane celý strom vyvážený (AVL).

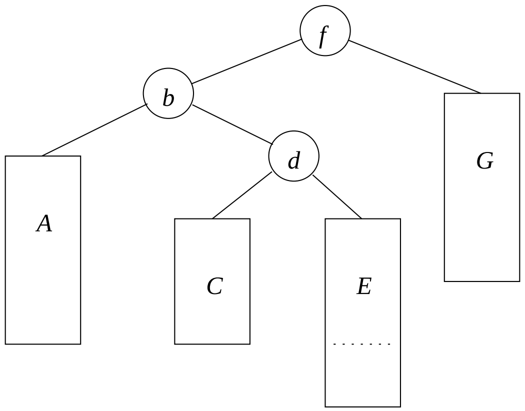
Přepočítáme hloubky na cestě od kořene stromu T k uzlu x.

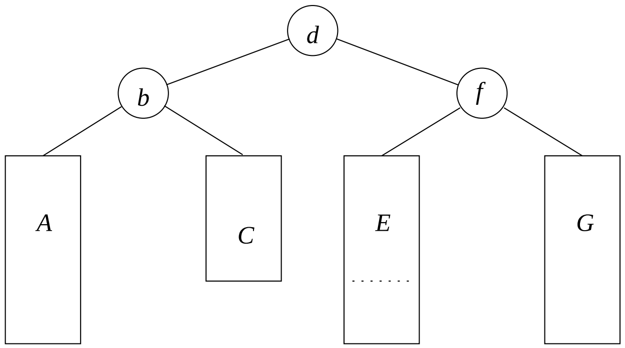
Byl-li uzel x přidáván do *pravého* podstromu stromu F, je postup analogický (stranově převrácený).


f
b
A
D
G
b
f
A
D
G


f
b
d
A
C
E
G


d
b
f
A
C
E
G


f
b
d
A
C
E
G


d
b
f
A
C
E
G

Složitost operace přidání položky do AVL

Hloubka AVL stromu je logaritmická vzhledem k počtu jeho uzlů.

Přidání uzlu x: $\Theta(\log n)$.

Nalezení nejmenšího nevyváženého podstromu:

Protože hloubky podstromů jsou spočteny a uloženy v datové struktuře, stačí po cestě k uzlu x testovat, zda rozdíl spočtených hloubek levého a pravého podstromu každého uzlu je v množině $\{-1, 0, 1\}$. Nejnižší uzel, který tuto podmínku nesplňuje, je kořen podstromu F. Jeho nalezení trvá $\Theta(\log n)$.

Hloubky podstromů se přepočítávají jen po cestě od kořene k x, tedy složitost této úpravy je $\Theta(\log n)$.

Celková složitost přidání uzlu je tedy $\Theta(\log n)$.

Rušení položky (uzlu) z AVL stromu

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

Odebíraný list označíme x.

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

Bez újmy na obecnosti lze předpokládat, že uzel x byl odebrán z *levého* podstromu stromu B. Nechť A je levý podstrom stromu B, F je pravý podstrom stromu B.

Strom F je neprázdný (jinak by rušení uzlu x nemohlo porušit vyváženost 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

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

   Pak strom T má hloubku $h' = h - 1$, je AVL a má stejné uzly jako měl strom B.

V původním AVL stromu nahradíme podstrom B stromem T.

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

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

Proto je nutné proces vyvažování opakovat: nalezne se nejmenší nadstrom T_2 stromu $T = T_1$, který není vyvážený, vyvážíme ho, nalezneme další nevyvážený nadstrom $T_3 \ldots$ atd., až je vyvážený celý strom, T_k.

Hloubky však stačí přepočítávat vždy od kořene stromu T ke kořenu nejbližšího vyvažovaného nadstromu.

Složitost operace rušení položky z AVL

Zrušení listu: $\Theta(\log n)$.

Nechť hloubka nejmenšího nevyváženého podstromu T_1 je h_1. Nalezení tohoto podstromu trvá $\Theta(h_1)$, jeho vyvážení (rotace uzlů) trvá konstantní dobu, přepočítání hloubek trvá $\Theta(h_1)$.

Nechť pro $1 < i \leq k$ je h_i délka cesty z kořene podstromu T_{i-1} do kořene stromu T_i. Pak každé nalezení dalšího nevyváženého podstromu T_i trvá $\Theta(h_i)$, jeho vyvážení trvá $\Theta(1)$, přepočítání hloubek trvá $\Theta(h_i)$.

To znamená, že celková 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?

Černobílé stromy

Černobílé stromy jsou binární vyhledávací stromy, jejichž uzly nesou kromě klíče další atribut — *barvu* — černou nebo bílou.

Def: *Černobílý strom* je binární vyhledávací 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 následníci (pokud existují) jsou černí.
3. Všechny větve obsahují stejný počet černých uzlů.

Poznámka: Vedle názvu *černobílé stromy* se lze setkat i s doslovnými překlady anglického názvu *red-black tree* (drzewo czerwono-czarne, rot-schwarzer Baum, ruĝnigra arbo, . . .).

Def: *Černá hloubka* černobílého stromu t je počet černých uzlů na libovolné větvi. Značíme ji $\mathsf{bh}(t)$.

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

Důkaz: Indukcí podle hloubky stromu se ukáže, že každý černobílý strom t' má aspoň $2^{\mathsf{bh}(t')} - 1$ uzlů. Odtud vyplývá $\mathsf{bh}(t') \leq \log_2(n+1)$. Ale podle definice černobílého stromu jeho hloubka nepřevyšuje dvojnásobek jeho černé hloubky. Odtud plyne tvrzení.

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

Černobílé stromy v Haskellu

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

Vyhledávání v černobílém stromu je stejné jako v nevyváženém vyhledávacím stromu.

Zjišťování 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
```

Vyhledávání

```
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
```

Přidávání uzlu do černobílého stromu

Přidávaný uzel bude bílý. Tím se nezmění černá hloubka podstromů, ale mohou se dostat pod sebe dva bílé uzly.

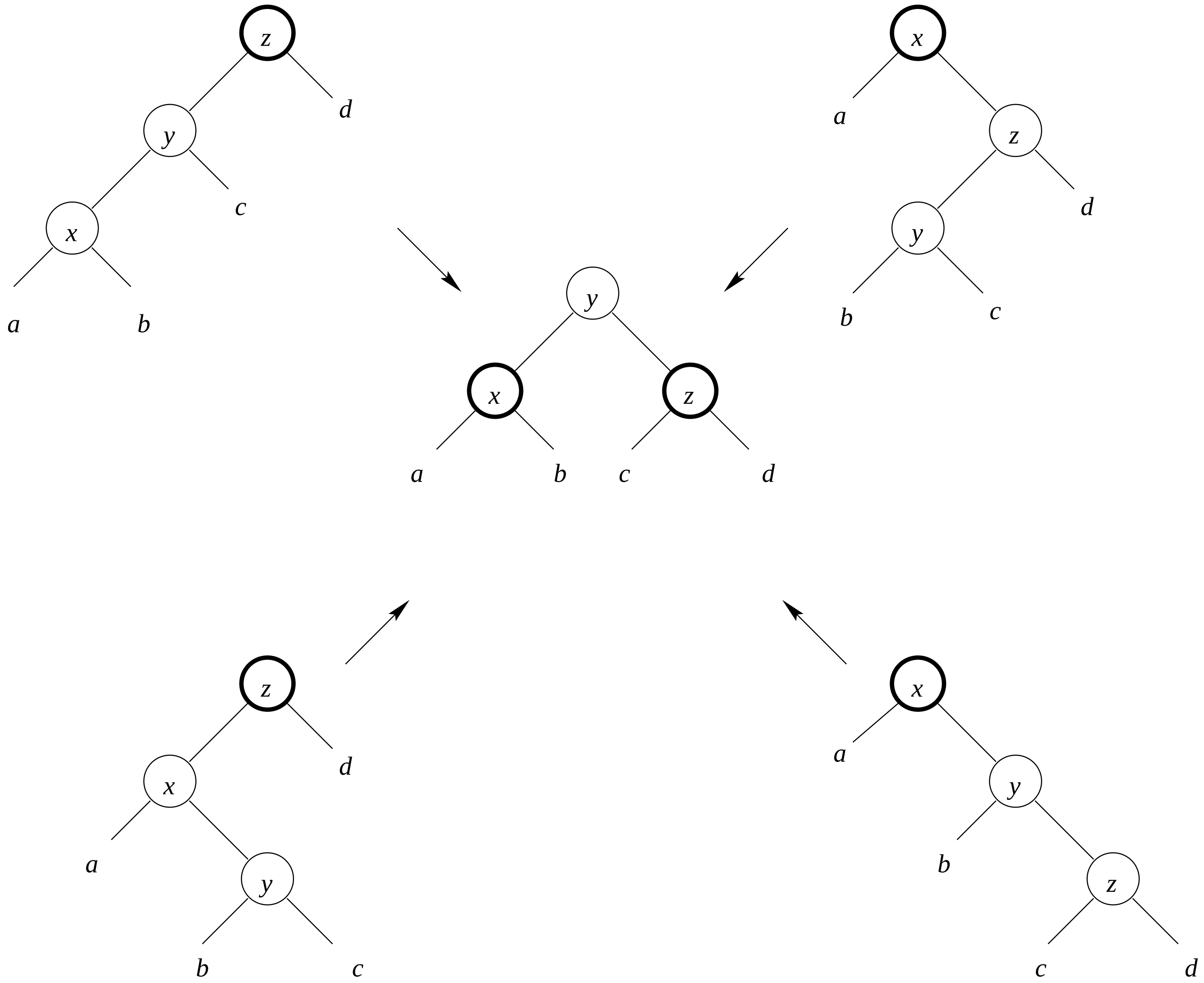
Se dvěma bílými uzly nad sebou a s černým uzlem nad nimi provedeme takovou rotaci, abychom snížili hloubku stromu, ale přebarvíme je tak, aby černá hloubka stromu zůstala zachována: kořen bude bílý a jeho dva následníci černí.

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

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

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

Složitost operace přidání uzlu

Věta: Operace `insert` přidání položky do černobílého stromu velikosti n má časovou složitost $\Theta(\log n)$.

Důkaz: Vyplývá z logaritmické hloubky černobílého stromu a z toho, že operace vyvážení `bal` má konstantní složitost.

Cvičení: Proč se při přidávání položky do černobílého 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 černobílý 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.

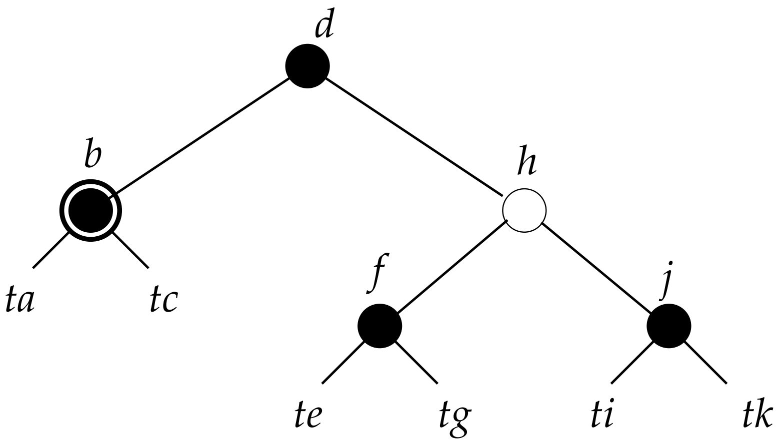
Rušení uzlu z černobílého stromu

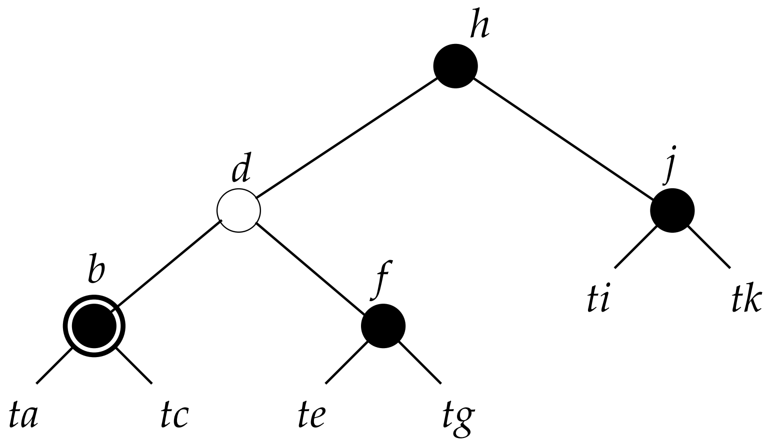
Rušení vnitřního uzlu převedeme známým způsobem na rušení listu. Je-li rušený list bílý, je zrušení triviální — strom i bez listu zůstane černobílý. Je-li rušený list černý, je nutno ho nejdříve „odbarvit“. Odbarvíme-li černý list, stane se tento list bílým a můžeme ho snadno zrušit.

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 „jednou černý“. Tím se sníží černá hloubka celého stromu.

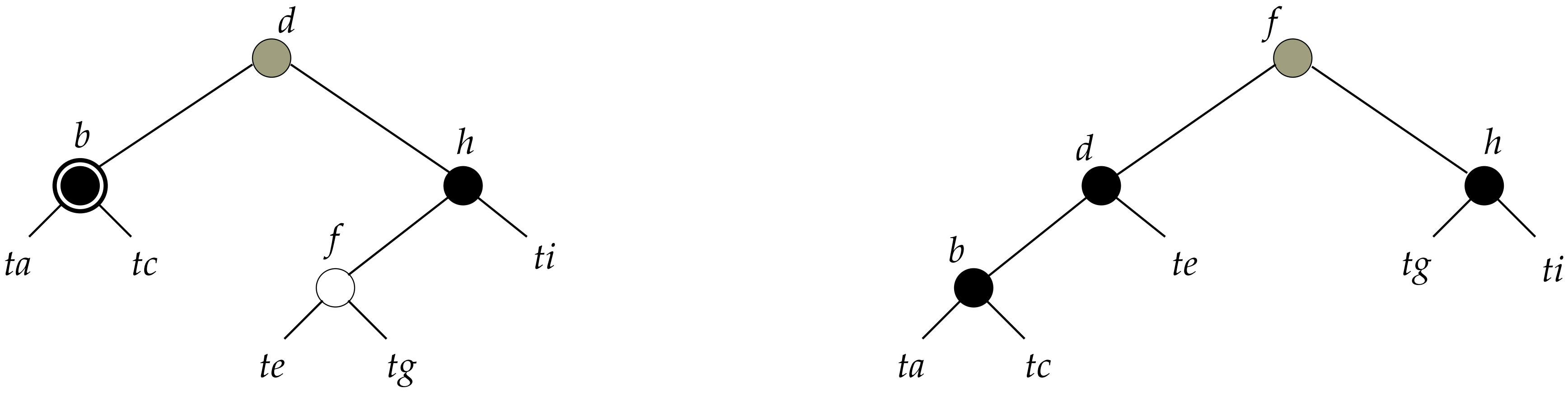
1. Bratr odbarvovaného uzlu je bílý — převede se na případ 2





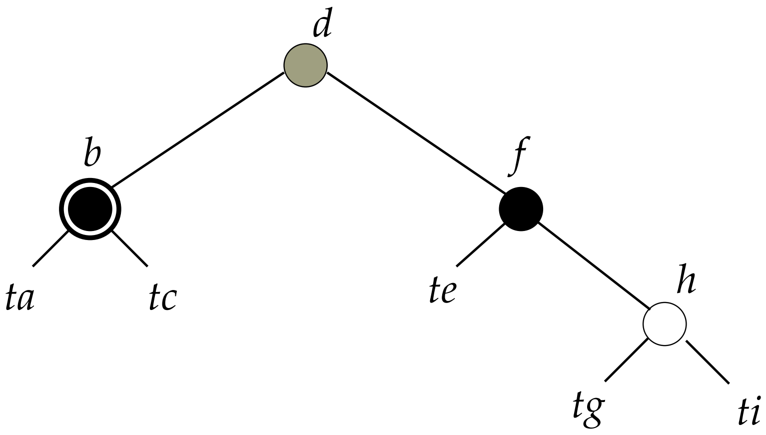
2. Bratr odbarvovaného uzlu je černý

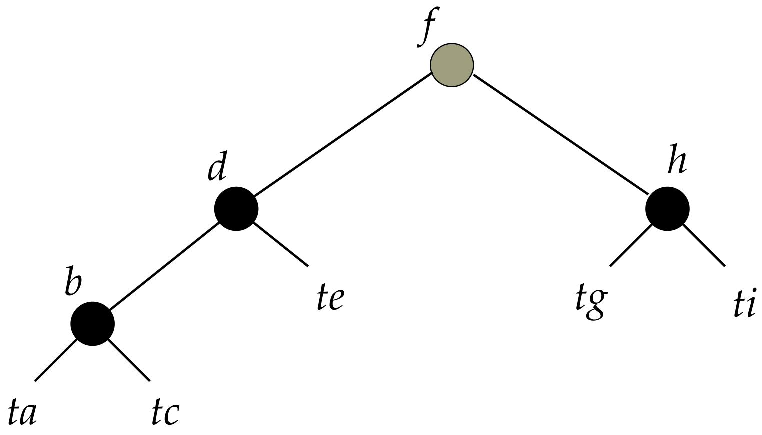
2a bližší synovec je bílý



2. Bratr odbarvovaného uzlu je černý

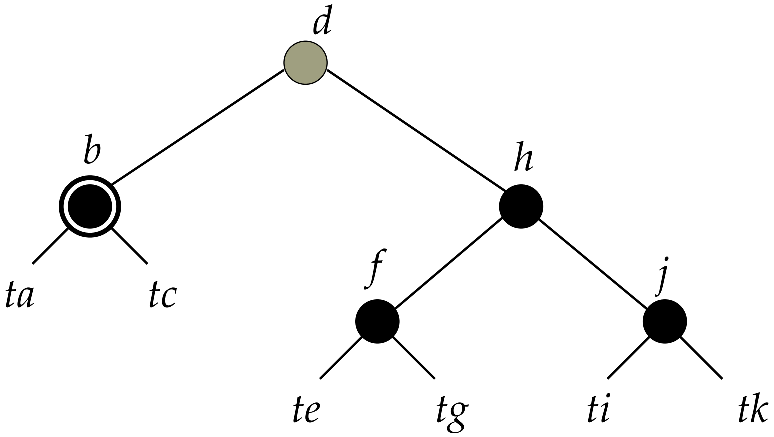
   2b vzdálenější synovec je bílý

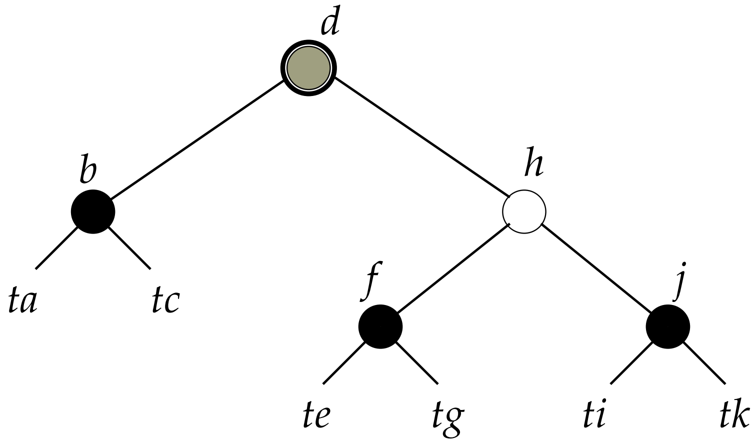




2. Bratr odbarvovaného uzlu je černý

2c žádný 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 $\Theta(\log_k n)$.

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 ohodnocené 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) nazýváme *uzlově a hranově ohodnocený orientovaný graf*.

Vynecháním hranového ohodnocení h_E dostaneme uzlově ohodnocený orientovaný graf (V, E, h_V).

Vynecháním uzlového ohodnocení h_V dostaneme hranově ohodnocený orientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E dvojice tvaru (x, x), máme (ohodnocené) orientované grafy bez smyček.

Def: Nechť V, H_V, H_E jsou množiny, E je nějaká 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) nazýváme *uzlově a hranově ohodnocený neorientovaný graf*.

Vynecháním hranového ohodnocení h_E dostaneme uzlově ohodnocený neorientovaný graf (V, E, h_V).

Vynecháním uzlového ohodnocení h_V dostaneme hranově ohodnocený neorientovaný graf (V, E, h_E).

Nepřipustíme-li v množině E jednoprvkové množiny, dostáváme (ohodnocené) neorientované grafy bez smyček.

Poznámka: 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 libovolná konečná množina, na níž je dáno zobrazení $e : E \to V^2$, které určuje, mezi kterými uzly daná hrana vede. Není-li zobrazení e injektivní, hovoří se o tzv. *multigrafu*.

Reprezentace

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

Pro tzv. *řídké grafy*, tj. grafy, v nichž počet hran je v $o(n^2)$, je výhodná reprezentace pomocí *množin sousedů*: neohodnocený graf $G = (V, E)$ na n uzlech je reprezentován posloupností $[s_1, \ldots, s_n]$ množin následníků každého uzlu. Jsou-li $v_1, \ldots, v_k$ všichni následníci 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-tého seznamu s_i dvojici $(v_j, h_E(i, v_j))$.

Procházení grafu

Problém: Projít systematicky všechny uzly grafu.

Procházení grafu do hloubky

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

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

Budeme označovat všechny navštívené uzly pomocí procedury `mark`.

Na začátku jsou všechny uzly nenavštívené ($\mathtt{marked}(u) = \mathtt{False}$).

Pro každý uzel u je $\mathtt{Successors}(u)$ seznam uzlů – následníků uzlu u.

Vycházíme z uzlů z množiny W a postupujeme stále dál po hranách do dosud nenavštívených uzlů.

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

Procházení 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 dosažitelné z uzlů z množiny W.

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

Poznámka: Algoritmus funguje stejně pro orientované i neorientované grafy.

Poznámka: Algoritmus obecně není deterministický, protože nemusí být (a nebývá) specifikováno pořadí uzlů v množinách `Successors(u)` ani v množině W. Procházení grafu do hloubky tedy neurčuje pořadí navštívených uzlů jednoznačně.

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

Poznámka: Významnou praktickou aplikací algoritmu `dfs` je nalezení tzv. *silně souvislých komponent* orientovaného 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 $\mathtt{marked}(u) = \mathtt{True}$.

Procházení grafu do šířky – 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 procházení grafu do šířky

Def: *Délkou sledu* (v hranově neohodnoceném grafu) rozumíme počet hran na tomto sledu.

Vzdálenost z uzlu u do uzlu v je délka minimálního sledu vedoucího z uzlu u do uzlu v, pokud takový sled existuje; pokud z uzlu u do uzlu v žádný sled neexistuje, pak vzdálenost položíme $+\infty$.

V neorientovaném grafu mluvíme stručně o *vzdálenosti 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 vžy 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 $\texttt{bfs}(G, W)$ 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 ptáme na označenost jejich následníků. Tedy délka výpočtu bude jistě v $\Omega(|E|)$. Jelikož zařazujeme do fronty všechny uzly z W, je délka výpočtu také v $\Omega(|W|)$. Z těchto dvou faktů vyplývá, že je v $\Omega(|E|) \cap \Omega(|W|) = \Omega(|W| + |E|)$.

Do fronty zařazujeme jen uzly, které až do té doby nebyly označené, a přitom je hned označíme. Odtud plyne, že každý uzel je do fronty zařazen jen jednou.
Tedy vnější cyklus while proběhne $O(|W| + |E|)$-krát. Test ve vnitřním cyklu for proběhne vždy tolikrát, kolik hran vychází z uzlu v, dohromady tolikrát, kolik má celý graf hran. Odtud opět dostáváme, že délka výpočtu je v $O(|W| + |E| + |E|) = O(|W| + |E|)$.

Dohromady dostáváme $\Theta(|W| + |E|)$, čímž je tvrzení dokázáno.

Minimální 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 nazývá *kostra* grafu G.

Def: Nechť G je souvislý neorientovaný graf s číselně ohodnocenými hranami. Kostra grafu G, která má ze všech jeho koster nejmenší součet hranových ohodnocení, se nazývá *minimální kostra* grafu G.

Poznámka: Kostry mají jen souvislé grafy, protože kostra musí podle definice být strom. Tento požadavek však není zásadní a většinu úvah o kostrách lze přenést i na nesouvislé grafy: stačí uvažovat zvlášť kostru každé komponenty.

Cvičení: Máme neorientovaný graf $G = (V, E), V = \{1, 2, 3, 4\}$,
$E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 3\}\}$.
Kolik má tento graf podgrafů, faktorů a koster? V každém z těchto tří případů spočítejte, kolik je *všech* podgrafů (faktorů, koster), a kolik z nich je *navzájem neisomorfních*.

Generický algoritmus pro nalezení minimální kostry

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

Úkolem je nalézt jeho minimální 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 $\texttt{kruskal}(G, A)$ 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 $m = |E|$.