VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Strategie řešení transformuj a vyřeš

doc. Mgr. Jiří Dvorský, Ph.D.

Stav prezentace ke dni 8. října 2024

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Osnova přednášky

Strategie řešení transformuj a vyřeš

Předtřídění dat

- Jedinečnost prvků v poli
- Výpočet modu
- Vyhledávání

Gaussova eliminační metoda

- ***LU***-rozklad matice
- Inverzní matice
- Determinant matice

Vyvážené vyhledávací stromy

- AVL stromy

Osnova přednášky (pokrač.)

- 2-3 stromy

Halda a třídění haldou

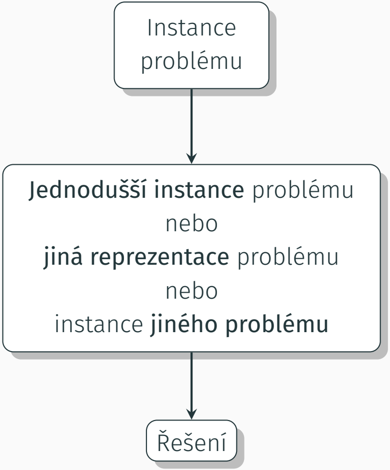
Hornerovo schéma

Redukce problému

Strategie řešení transformuj a vyřeš

Dvoufázová strategie

1. transformace
2. řešení



Strategie řešení transformuj a vyřeš

Předtřídění dat

Předtřídění dat

- Poměrně stará myšlenka, která mimo jiné motivovala výzkum třídících algoritmů.
- Setříděná data vedou na výrazně jednodušší algoritmy, „pořádek musí být“.
- Předpoklady:
  1. data jsou uložena v poli – třídění pole je snazší než třídění seznamu
  2. pro třídění použijeme algoritmus se složitostí $\Theta(n \log n)$ – typicky QuickSort, MergeSort.
- Využití: geometrické algoritmy, grafové algoritmy, žravé algoritmy.

Jedinečnost prvků v poli

Zadání

Máme dáno pole A s n prvky. Máme určit, zda se v poli A vyskytuje každý prvek právě jednou.

Řešení hrubou silou – porovnáváme všechny dvojice prvků dokud:

1. nenajdeme dvojici stejných prvků nebo
2. jsme otestovali všechny dvojice prvků.

Časová složitost je v nejhorším případě $\Theta(n^2)$.

Jedinečnost prvků v poli

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
  if $A[i] = A[i+1]$ **return false**
return true

Časová složitost algoritmu

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Výpočet modu

Zadání

Máme dáno pole A s n prvky. Máme určit, který prvek se v poli vyskytuje nejčastěji. Tento prvek se nazývá **modus**.

Pro jednoduchost budeme předpokládat, že v poli A existuje jen jeden modus.

Řešení hrubou silou

Pro každý prvek $a_i \in A$ prohledáme pomocný seznam L:

1. pokud nalezneme shodu, inkrementujeme příslušnou četnost,
2. v opačném případě vložíme prvek a_i na konec seznamu s četností 1.

Výpočet modu – časová složitost řešení hrubou silou

- Nejhorší případ – všechny prvky v poli A jsou různé.
- Pro a_i musíme provést $i - 1$ porovnání s prvky v seznamu L, než přidáme nový prvek na jeho konec.
- Počet porovnání je tedy roven

$$C(n) = \sum_{i=1}^{n}(i - 1) = 0 + 1 + \cdots + (n - 1) = \frac{1}{2}n(n - 1) \in \Theta(n^2)$$

- Nalezení maxima vyžaduje $n - 1$ porovnání, což neovlivní kvadratickou složitost algoritmu.

Výpočet modu – předtřídění dat

- Pokud pole A setřídíme, budou shodné prvky v poli A vedle sebe.
- Pro výpočet modu stačí nalézt nejdelší úsek (angl. run) shodných prvků v A.
- Časová složitost

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Výpočet modu

ALGORITHM *PresortMode*($A[0..n-1]$)

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \le n-1$ **do**
    $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
    while $i + runlength \le n-1$ **and** $A[i + runlength] = runvalue$
        $runlength \leftarrow runlength + 1$
    if $runlength > modefrequency$
        $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
    $i \leftarrow i + runlength$
return $modevalue$

Vyhledávání prvku x v poli A délky n

- Řešení hrubou silou vede na algoritmus vyžadující n porovnání v nejhorším případě.
- Po setřídění pole, lze použít algoritmus půlení intervalu, který vyžaduje $\lfloor \log_2 n \rfloor + 1$ porovnání v nejhorším případě.
- Časová složitost algoritmu potom bude

  $$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

  což je **více** než složitost sekvenčního vyhledávání!!!
- Ale pro **opakované** vyhledávání se již vyplatí pole A setřídit.

Zdroje pro samostatné studium

- Kniha [1], kapitola 6.1, strany 202 – 205

Strategie řešení transformuj a vyřeš

Gaussova eliminační metoda

Gaussova eliminační metoda – motivace

Soustavu dvou rovnic o dvou neznámých

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

lze řešit poměrně snadno – například proměnnou x vyjádříme jako funkci y, dosadíme do druhé rovnice a rovnici vyřešíme.

Problém

Jak řešit soustavu n rovnic o n neznámých? Stejným způsobem?

Gaussova eliminační metoda

Soustavu $\boldsymbol{n}$ lineárních rovnic o $\boldsymbol{n}$ neznámých

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

transformujeme na ekvivaletní soustavu rovnic, kde všechny koeficienty pod hlavní diagonálou jsou nulové

$$\begin{aligned}
a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\
a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\
&\vdots \\
a'_{nn}x_n &= b'_n
\end{aligned}$$

Gaussova eliminační metoda – maticový zápis

$$\mathbf{A}\vec{x} = \vec{b} \quad \Longrightarrow \quad \mathbf{A}'\vec{x} = \vec{b}'$$

kde

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix}$$

$$\mathbf{A}' = \begin{pmatrix} a'_{11} & a_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \qquad \vec{b}' = \begin{pmatrix} b'_{11} \\ b'_{21} \\ \vdots \\ b'_{n1} \end{pmatrix}$$

$\mathbf{A}'$ se nazývá **horní trojúhelníková matice**.

Gaussova eliminační metoda – výhody změny reprezentace

Soustavu danou horní trojúhelníkovou maticí lze snadno řešit pomocí **zpětné substituce**:

1. Z rovnice
$$a'_{nn}x_n = b'_n$$
vypočteme neznámou x_n.
2. Hodnotu neznámé x_n dosadíme do rovnice
$$a'_{n-1\,n-1}x_{n-1} + a'_{n-1\,n}x_n = b'_{n-1}$$
a vypočteme neznámou x_{n-1}.
3. Takto postupujeme dále až k výpočtu neznámé x_1.

Složitost tohoto algoritmu je $\Theta(n^2)$.

Gaussova eliminační metoda – elementární operace

Matici soustavy $\mathbf{A}$ převedeme na horní trojúhelníkovou matici $\mathbf{A}'$ pomocí **elementárních operací**:

- záměna dvou rovnic v soustavě,
- vynásobení rovnice nenulovým koeficientem a
- přičtení či odečtení násobku jiné rovnice k dané rovnici, tj. lineární kombinace s jinou rovnicí.

Elementární operace nemění řešení soustavy rovnic – transformovaná soustava má stejné řešení jako původní soustava.

Gaussova eliminační metoda – transformace matice

1. Zvolíme a_{11} jako **pivot** a „vynulujeme“ všechny koeficienty v prvním sloupcí, kromě a_{11}.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

   „Vynulování“ – od druhé rovnice odečteme $\frac{a_{21}}{a_{11}}$ násobek první rovnice, od třetí rovnice odečteme $\frac{a_{31}}{a_{11}}$ násobek první rovnice,…
2. Zvolíme a_{22} jako pivot a opakujeme stejný postup.

Poznámka

Změny provádíme pochopitelně i pro vektor pravých stran $\vec{b}$.

Gaussova eliminační metoda – příklad

Mějme soustavu rovnic

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

Rozšířená matice soustavy

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right)$$

Gaussova eliminační metoda – příklad (pokrač.)

Dopředná eliminace

Od druhého řádku odečteme $\frac{4}{2}$ násobek prvního řádku, od třetího řádku odečteme $\frac{1}{2}$ násobek prvního řádku

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{3}{2} \end{array}\right)$$

Od třetího řádku odečtem $\frac{\frac{3}{2}}{3} = \frac{1}{2}$ násobek druhého řádku

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right)$$

Gaussova eliminační metoda – příklad (pokrač.)

Zpětná substituce

$$
\begin{aligned}
x_3 &= \frac{-2}{2} = -1 \\
x_2 &= \frac{3-(-3)x_3}{3} = \frac{3-(-3)(-1)}{3} = 0 \\
x_1 &= \frac{1-x_3-(-1)x_2}{2} = \frac{1-(-1)}{2} = 1
\end{aligned}
$$

Gaussova eliminační metoda – dopředná eliminace

Vstup : Matice $\mathbf{A}$ typu $n \times n$ a sloupcový vektor $\vec{b}$ dimenze n

Výstup: Ekvivalentní trojúhelníková matice $\mathbf{A}$ a vektor $\vec{b}$

```
for i ← 1 to n - 1 do
    for j ← i + 1 to n do
        temp ← A[j, i]/A[i, i];
        for k ← i to n do
            A[j, k] ← A[j, k] - A[i, k] * temp;
        end
        b[j] ← b[j] - b[i] * temp;
    end
end
```

Gaussova eliminační metoda – dopředná eliminace

Částečné pivotování

- V algoritmu dopředné eliminace je chyba. Pokud $a_{ii} = 0$, tak dojde k dělení nulou.
- Problém lze řešit výměnou rovnic (elementární operace) tak, aby $a_{ii} \neq 0$.
- Lze současně řešit i případné zaokrouhlovací chyby – pivot volíme tak, aby byl ze všech prvků a_{ii} až a_{ni} v absolutní hodnotě největší.

ALGORITHM *BetterForwardElimination*($A[1..n, 1..n], b[1..n]$)

```
//Implements Gaussian elimination with partial pivoting
//Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the (n + 1)st column
for i ← 1 to n do A[i, n + 1] ← b[i] //appends b to A as the last column
for i ← 1 to n − 1 do
    pivotrow ← i
    for j ← i + 1 to n do
        if |A[j, i]| > |A[pivotrow, i]|  pivotrow ← j
    for k ← i to n + 1 do
        swap(A[i, k], A[pivotrow, k])
    for j ← i + 1 to n do
        temp ← A[j, i] / A[i, i]
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] − A[i, k] ∗ temp
```

Gaussova eliminační metoda – časová složitost

- Velikost vstupu – počet rovnic v soustavě, tj. rozměr matice n.
- Základní operace – aritmetické operace, z historických důvodů násobení. V nejvnitřnějším cyklu počet násobení odpovídá počtu odčítání, jde jen o násobek konstantou 2.
- Bude nás zajímat počet násobení $C(n)$ v závislosti na čísle n.

Gaussova eliminační metoda – časová složitost (pokrač.)

$$
\begin{aligned}
C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n - i + 1) \\
&= \sum_{i=1}^{n-1} (n - i + 1) \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n - i + 1)(n - i)
\end{aligned}
$$

Poslední sumu rozepíšeme pro jednotlivá i

$$
\begin{array}{ccccc}
i = 1 & (n - 1 + 1)(n - 1) & = & n(n-1) \\
i = 2 & (n - 2 + 1)(n - 2) & = & (n-1)(n-2) \\
\vdots & \vdots & \vdots & \vdots \\
i = n - 2 & (n - n + 2 + 1)(n - n + 2) & = & 3 \cdot 2 \\
i = n - 1 & (n - n + 1 + 1)(n - n + 1) & = & 2 \cdot 1
\end{array}
$$

Z posledního sloupce je vidět, že jde o součet řady

$$1 \cdot 2 + 2 \cdot 3 + \cdots + (n-2)(n-1) + (n-1)n = \sum_{l=1}^{n-1} l(l+1)$$

$$\begin{aligned}
\sum_{l=1}^{n-1} l(l+1) &= \sum_{l=1}^{n-1} l^2 + \sum_{l=1}^{n-1} l \\
&= \frac{1}{6}n(n-1)(2n-1) + \frac{1}{2}n(n-1) \\
&= \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n + \frac{1}{2}n^2 - \frac{1}{2}n \\
&= \frac{1}{3}n^3 - \frac{1}{3}n
\end{aligned}$$

Gaussova eliminační metoda – časová složitost (pokrač.)

A tedy

$$C(n) = \frac{1}{3}n^3 - \frac{1}{3}n \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Protože složitost zpětné substituce je $\Theta(n^2)$, je složitost celé Gaussovy eliminační metody $\Theta(n^3)$.

LU-rozklad matice

Mějme matici $\mathbf{A}$ soustavy lineárních rovnic z předchozího příkladu

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

Dále uvažujme dvě matice:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}$$

Koeficienty z Gaussovy eliminace

$$\mathbf{U} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Výsledek Gaussovy eliminace

Definice

Mějme $\mathbf{A}$ regulární čtvercovou matici s prvky z $\mathbb{R}$, u které není třeba při Gaussově eliminaci prohazovat řádky. Pak existují regulární matice $\mathbf{L}$ a $\mathbf{U}$, které jsou určeny jednoznačně a platí pro ně následující tvrzení

$$\mathbf{A} = \mathbf{LU},$$

kde $\mathbf{L}$ je dolní trojúhelníková matice s jedničkami na celé hlavní diagonále a $\mathbf{U}$ horní trojúhelníková matice s nenulovými prvky na hlavní diagonále.

Řešení soustavy rovnic *LU*-rozkladem

Mějme soustavu lineárních rovnic

$$\mathbf{A}\vec{x} = \vec{b}$$

Matici $\mathbf{A}$ nahradíme jejím *LU* rozkladem

$$\mathbf{L}\mathbf{U}\vec{x} = \vec{b}$$

Dále označme součin $\mathbf{U}\vec{x} = \vec{y}$. Po dosazení dostáváme soustavu rovnic

$$\mathbf{L}\vec{y} = \vec{b}$$

Tuto soustavu můžeme snadno vyřešit, protože $\mathbf{L}$ je dolní trojúhelníková matice. A nakonec můžeme lehce vyřešit i soustavu

$$\mathbf{U}\vec{x} = \vec{y},$$

protože $\mathbf{U}$ je horní trojúhelníková matice.

Řešení soustavy rovnic *LU*-rozkladem, příklad

Mějme soustavu rovnic

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

Provedeme *LU*-rozklad matice soustavy $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Řešení soustavy rovnic LU-rozkladem, příklad (pokrač.)

Nejprve budeme řešit soustavu $\mathbf{L}\vec{y} = \vec{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$$

$$\begin{aligned} y_1 &= 1 \\ y_2 &= 5 - 2y_1 = 3 \\ y_3 &= 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2 \end{aligned}$$

Řešení soustavy rovnic *LU*-rozkladem, příklad (pokrač.)

Následně vyřešíme soustavu $\mathbf{U}\vec{x} = \vec{y}$

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$$

$$\begin{aligned} x_3 &= \frac{-2}{2} = -1 \\ x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\ x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1 \end{aligned}$$

LU-rozklad matice, poznámky

- V praxi se pro řešení soustav lineárních rovnic používá právě ***LU***-rozklad.
- Pomocí ***LU***-rozkladu lze efektivně řešit více soustav rovnic se shodnou maticí soustavy.
- Matice $\mathbf{L}$ a $\mathbf{U}$ lze uložit společně v jedné „matici“ – z matice $\mathbf{L}$ ukládáme jen prvky pod diagonálou. Proč?
- Pokud je nutné v matici $\mathbf{A}$ nutno provést částečné pivotování, tj. prohazovat řádky, pak má rozklad tvar

  $$\mathbf{PA} = \mathbf{LU}$$

  a odtud

  $$\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU},$$

  kde $\mathbf{P}$ je permutační matice.

Permutační matice

- Reprezentuje permutaci n prvků jako matici
- Čtvercová binární matice řádu n, v každém řádku a sloupci je jedna 1, zbytek 0
- Pro každou permutační matici $\mathbf{P}$ platí:
  - součin zleva, $\mathbf{PM}$, vede k permutaci řádků matice $\mathbf{M}$, kde $\mathbf{M}$ je matice s n řádky
  - součin zprava, $\mathbf{MP}$, vede k permutaci sloupců matice $\mathbf{M}$, kde $\mathbf{M}$ je matice s n sloupci
  - $\mathbf{P}$ je ortogonální, tj. inverzní matice je rovna matici transponované, $\mathbf{P}^{-1} = \mathbf{P}^T$

Permutační matice, příklad

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \quad \leftrightarrow \quad R_\pi = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\updownarrow \qquad\qquad\qquad\qquad \updownarrow$$

$$C_\pi = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \leftrightarrow \quad \pi^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$$

Definice

Ke každé regulární čtvercové matici $\mathbf{A}$ řádu n existuje právě jedna čtvercová matice $\mathbf{A}^{-1}$ řádu n taková, že

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}.$$

Matice $\mathbf{A}^{-1}$ se nazývá **inverzní matice** k matici $\mathbf{A}$.

Matice bez inverze se nazývají **singulární**.

Inverzní matice – obdoba převráceného čísla u racionálních či reálných čísel; srovnej řešení lineární rovnice a soustavy lineárních rovnic

$$ax = b \qquad\qquad \mathbf{A}\vec{x} = \vec{b}$$

$$x = \frac{1}{a}b = a^{-1}b \qquad\qquad \vec{x} = \mathbf{A}^{-1}\vec{b}$$

Výpočet inverzní matice

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Musíme řešit n soustav lineárních rovnic o n neznámých se stejnou maticí $\mathbf{A}$

$$\mathbf{A}\vec{x}^j = \vec{e}^j,$$

kde $\vec{x}^j$ resp. $\vec{e}^j$ je j-tý sloupcový vektor matice $\mathbf{A}$ resp. $\mathbf{I}$.

Využijeme LU rozklad

$$\mathbf{LU}\vec{x}^j = \vec{e}^j$$

Determinant matice

Definice

Determinant čtvercové matice $\mathbf{A}$ řádu n definujeme předpisem

$$\det \mathbf{A} = \sum_{p \in P_n} \sigma(p) a_{1p_1} a_{2p_2} \cdots a_{np_n},$$

kde

- P_n je množina všech permutací čísel $\{1, \dots, n\}$ a
- $\sigma(p) = (-1)^s$ je znaménko permutace, s označuje počet inverzí v permutaci p.

Determinant $\det \mathbf{A}$ zapisujeme i jako $|\mathbf{A}|$.

Výpočet determinantů matic řádu 1, 2 a 3

$$\begin{vmatrix} a_{11} \end{vmatrix} = a_{11}$$

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{13}a_{21}a_{32} + a_{12}a_{23}a_{31}$$

$$-a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

Poslední vzorec je znám také jako tzv. Sarrusovo pravidlo.

Výpočet determinantů matic vyšších řádů

- Výpočet podle definice vyžaduje sečíst $n!$ součinů prvků matice.
- Opět lze využít, lehce modifikovanou, Gaussovu eliminaci a upravit matici na horní trojúhelníkovou. Platí věta, že determinant horní (dolní) trojúhelníkové matice je roven součinu prvků na její diagonále.
- Determinanty vyšších řádů tak, lze počítat s kubickou časovou složitostí.

Výpočet determinantů matic vyšších řádů – příklad

$$\begin{aligned}\begin{vmatrix}3 & 6 & -9\\ 4 & 8 & 10\\ 2 & 7 & 5\end{vmatrix} &= 3\begin{vmatrix}1 & 2 & -3\\ 4 & 8 & 10\\ 2 & 7 & 5\end{vmatrix} = 3\begin{vmatrix}1 & 2 & -3\\ 0 & 0 & 22\\ 0 & 3 & 11\end{vmatrix} = -3\begin{vmatrix}1 & 2 & -3\\ 0 & 3 & 11\\ 0 & 0 & 22\end{vmatrix}\\ &= -3 \cdot 66 = -198\end{aligned}$$

Poznámka

Tento výpočet slouží pouze pro ukázku, determinant matice řádu 3 lze pochopitelně počítat přímo vzorcem.

Cramerovo pravidlo

Řešení soustavy rovnic $\mathbf{A}\vec{x} = \vec{b}$ lze vyjádřit jako

$$\vec{x} = \left(\frac{\det \mathbf{A}_1}{\det \mathbf{A}}, \dots, \frac{\det \mathbf{A}_i}{\det \mathbf{A}}, \dots, \frac{\det \mathbf{A}_n}{\det \mathbf{A}}, \right)$$

kde $\mathbf{A}_i$ je matice, která vznikne záměnou i-tého sloupce matice $\mathbf{A}$ za vektor pravých stran $\vec{b}$.

Složitost algoritmu

- Výpočet determinantu řádu n lze zvládnout v čase $O(n^3)$.
- Musíme vypočítat n determinantů matic $\mathbf{A}_i$ a jeden determinant matice $\mathbf{A}$, celkem tedy $n + 1$ determinantů.
- Celková složitost výpočtu řešení soustavy rovnic Cramerovým pravidlem má složitost $O(n^4)$.

Zdroje pro samostatné studium

- Kniha [1], kapitola 6.2, strany 208 – 216
- Kniha [2], kapitoly 28.1 a 28.2, strany 819 – 838
- Kniha [3], kapitoly 12 a 13, strany 133 – 163
- Kniha [4], kapitoly 1.3 a 1.4, strany 24 – 36

Strategie řešení transformuj a vyřeš

Vyvážené vyhledávací stromy

Binární vyhledávací stromy – připomenutí

- Fundamentální datová struktura pro implementaci množin, slovníků atd.
- Každý uzel obsahuje jeden klíč; nad klíči musí být definováno uspořádání.
- Pro každý uzel platí, že všechny klíče v levém podstromu jsou menší než klíč v daném uzlu a v pravém podstromu jsou všechny klíče větší.
- Průměrná časová složitost hledání, vkládání a mazání uzlů je $\Theta(\log_2 n)$.
- Nejhorší případ je ale stále $\Theta(n)$ – strom degeneruje na seznam.

Možná řešení nejhoršího případu:

Aktivní opatření

- transformace na vyvážený binární strom pomocí rotací
- různé definice vyváženosti
- AVL stromy, červeno-černé stromy, splay stromy.

Změna reprezentace

- více klíčů v jednom uzlu,
- 2-3 stromy, 2-3-4 stromy, B-stromy.

AVL stromy

Autoři

- Georgij Maximovič **A**delson-**V**elskij a
- Jevgenij Michajlovič **L**andis

Poprvé publikováno v roce 1962.

Definice

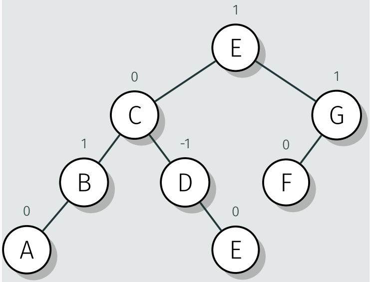
Faktorem vyváženosti uzlu u nazýváme rozdíl výšek jeho levého a pravého podstromu. Výšku prázdného stromu definujeme jako -1.

Definice

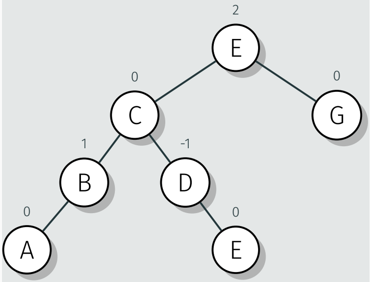
Binární vyhledávací strom nazýváme **AVL stromem** tehdy a jen tehdy, je-li faktor vyváženosti pro každý uzel ve stromu buď -1, 0 nebo +1.

AVL stromy – příklad

AVL strom



Toto není AVL strom

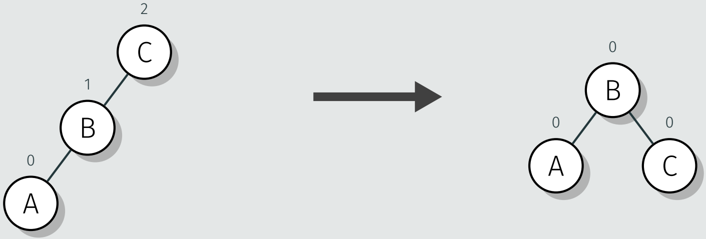


AVL stromy – udržování vyváženosti

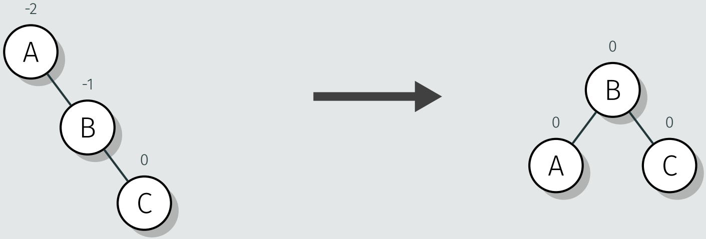
- Vložení nového uzlu, resp. smazání existujícího, může způsobit nevyváženost AVL stromu.
- Vyváženost je nutné po každé takové operaci obnovit.
- Vyváženost se obnovuje pomocí **rotací**.
- Rotace je lokální transformace stromu v těch uzlech, kde faktor vyváženosti dosáhne hodnoty -2 nebo 2.
- Pokud je takových uzlů více, začínáme vždy uzlem na nejnižší úrovni (co nejblíže k listům stromu).
  A postupujeme vzhůru ke kořeni stromu.
- Existují celkem čtyři rotace – dvě dvojice navzájem zrcadlově symetrických rotací.

Jednoduché rotace

R-rotace

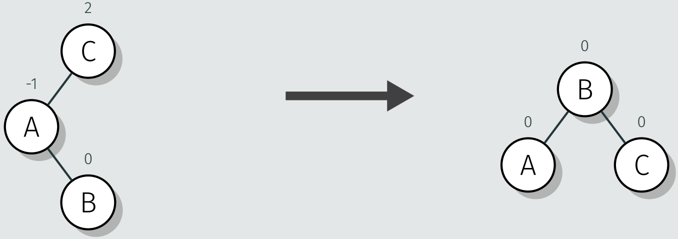


L-rotace

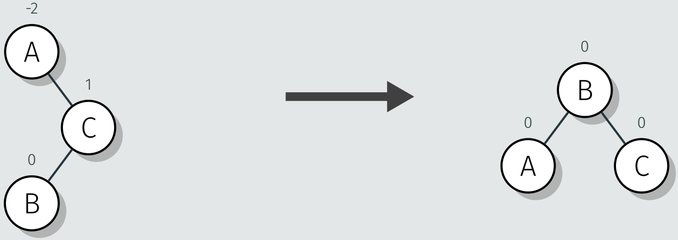


Dvojité rotace

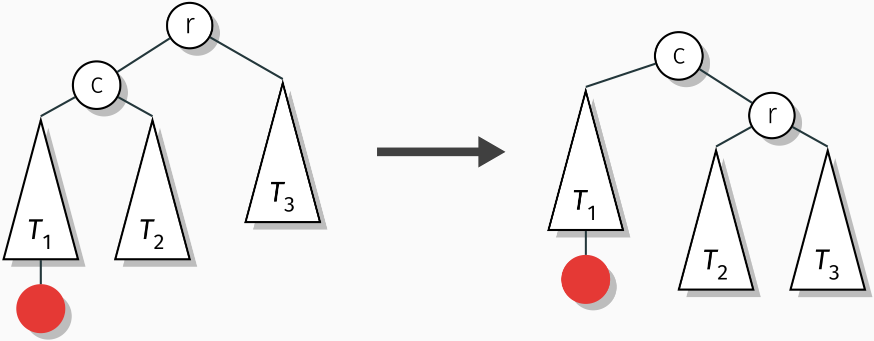
LR-rotace



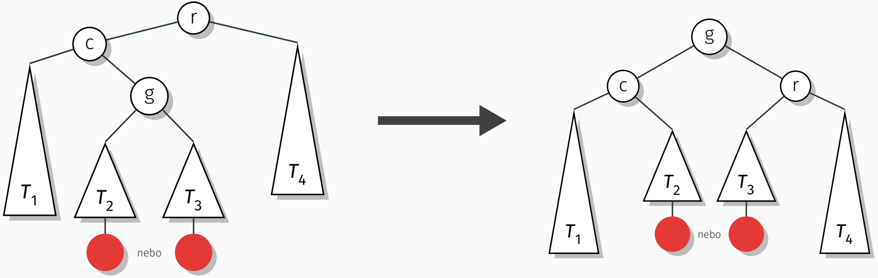
RL-rotace



AVL stromy – obecné schéma R-rotace



AVL stromy – obecné schéma LR-rotace



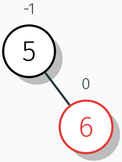
AVL stromy – vlastnosti rotací

- Konstantní časová složitost – přesunují se pouze ukazatele mezi uzly, ne data.
- Rotace zachovávají uspořádání klíčů ve stromu – po dokončení rotace jsou „vlevo“ vždy menší klíče, „vpravo“ vždy větší.

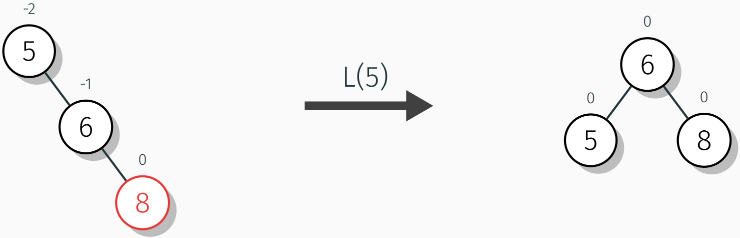
AVL stromy – postupná konstrukce stromu

Vložení 5

Vložení 6

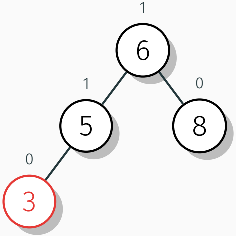


Vložení 8

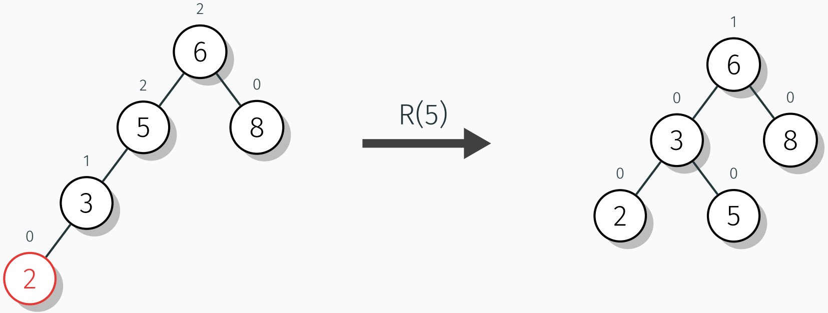


AVL stromy – postupná konstrukce stromu (pokrač.)

Vložení 3



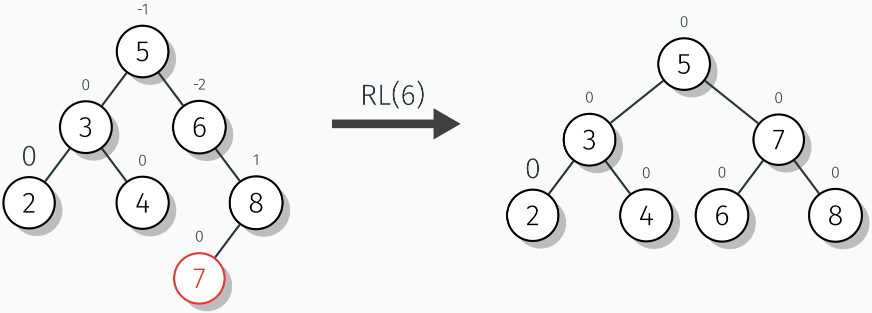
Vložení 2



Vložení 4

AVL stromy – postupná konstrukce stromu (pokrač.)

Vložení 7



AVL stromy – vlastnosti

- Výška AVL stromu s n uzly je ohraničena

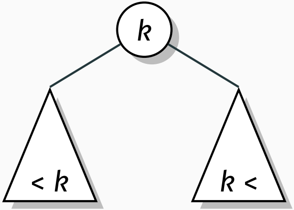
$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$$

- Operace vyhledávání a vkládání tedy probíhají se složitostí $\Theta(\log_2 n)$ i v nejhorším případě.
- Průměrná výška AVL stromu sestrojeného z náhodné posloupnosti n klíčů je $1.01 \log_2 n + 0.1$.
- Smazání uzlu je komplikovanější, ale stále spadá do logaritmické třídy složitosti.
- Nevýhody – velké množství rotací při vyvažování stromu.

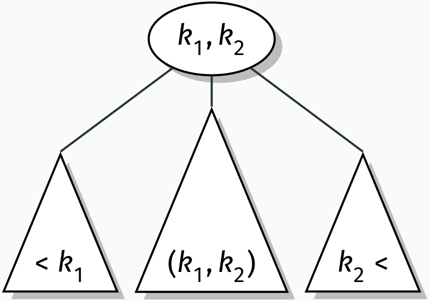
2-3 stromy

2-3 stromy – druhy uzlů

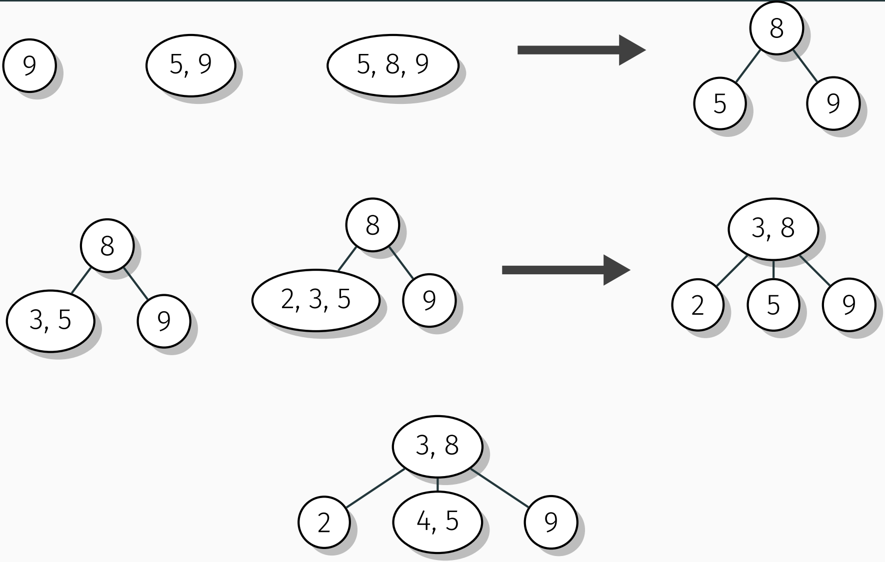
2-uzel



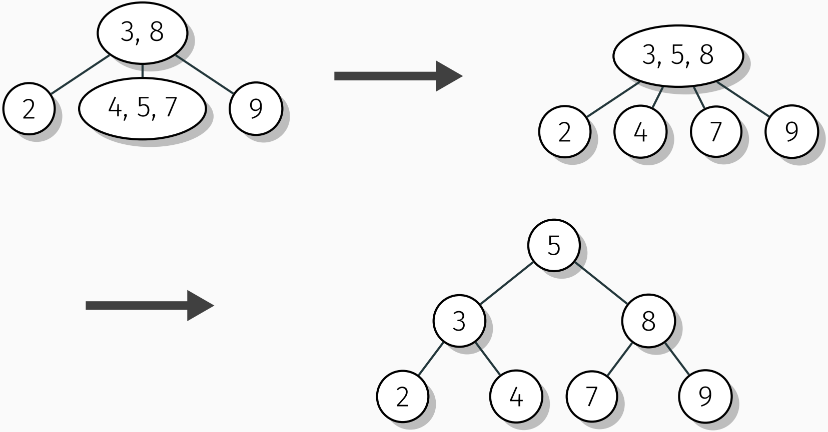
3-uzel



Konstrukce 2-3 stromu z posloupnosti 9, 5, 8, 3, 2, 4, 7



Konstrukce 2-3 stromu z posloupnosti 9, 5, 8, 3, 2, 4, 7 (pokrač.)



Zdroje pro samostatné studium

- Kniha [1], kapitola 6.3, strany 218 – 225
- Kniha [5], kapitoly 4.4.6, 4.4.7 a 4.4.8, strany 296 – 310

Strategie řešení transformuj a vyřeš

Halda a třídění haldou

Halda (Heap)

Halda – částečně setříděná datová struktura, zvláště vhodná pro implementaci prioritní fronty.

Prioritní fronta – datová struktura chápaná jako multimnožina, kde prvky jsou řazeny podle **priority** a podporující operace:

- nalezení prvku s nejvyšší prioritou,
- odebrání prvku s nejvyšší prioritou a
- vložení nového prvku do fronty.

Využití prioritní fronty :

- plánování úloh v OS
- grafových algoritmech např. Primův, Dijkstrův atd.
- třídění haldou – **HeapSort**
- a dalších...

Halda – rozlišení terminologie

Pojem **halda** se v informatice používá pro označení:

- datové struktury a
- části operační paměti za běhu programu.

V dalším výkladu se budeme zabývat haldou výhradně jako **datovou strukturou**.

Halda

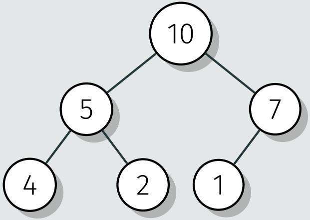
Definice

Haldu definujeme jako binární strom s jedním klíčem v každém uzlu, který splňuje následující dvě vlastnosti:

1. **kompletnost**, tj. všechna patra stromu jsou zaplněna, s výjimkou posledního. V posledním patře může zprava chybět několik listů a
2. **rodičovská dominance**, tj. klíč v každém uzlu je vždy větší nebo roven než klíče ve všech jeho potomcích. V listech je libovolný klíč vždy brán jako větší než klíče v neexistujících potomcích.

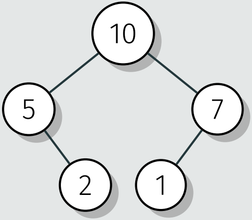
Halda – příklad

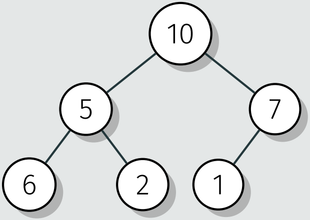
Halda



Ne každý binární strom je haldou!

Toto nejsou haldy – proč?





Halda – další vlastnosti

Pro všechny haldy lze dokázat, že:

1. Klíče na každé cestě z kořene do listu tvoří **nerostoucí** posloupnost. Jinak mezi klíči nejsou žádné vztahy, např. menší klíče v levém podstromu než v pravém atd.
2. Pro n klíčů existuje pouze jeden úplný binární strom. Jeho výška je $\lfloor \log_2 n \rfloor$.
3. Největší klíč je vždy v kořeni haldy.
4. Každý uzel v haldě je vždy kořenem haldy tvořené tímto uzlem a jeho potomky.

V poli haldu ukládáme směrem od kořene k listům a zleva doprava: Potom:

1. vnitřní uzly – prvních $\left\lfloor \frac{n}{2} \right\rfloor$, listy zbývajících $\left\lceil \frac{n}{2} \right\rceil$,
2. potomci uzlu na pozici i, kde $1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$, se nachází na pozicích $2i$ a $2i + 1$. A naopak rodič uzlu na pozici j, pro $2 \leq j \leq n$, se nachází na pozici $\left\lfloor \frac{j}{2} \right\rfloor$.

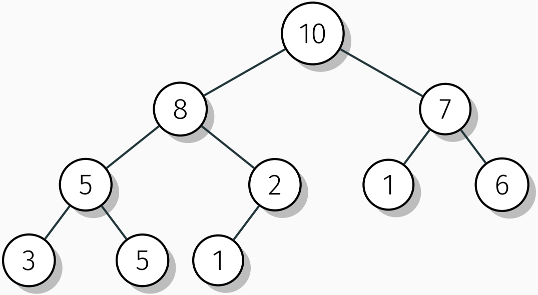
Poznámka

Haldu lze definovat jako pole $H[1 \dots n]$ ve kterém pro každý prvek na indexu i platí

$$H[i] \geq max\{H[2i], H[2i + 1]\}$$

pro všechna $i = 1, \dots, \left\lfloor \frac{n}{2} \right\rfloor$.

Halda – reprezentace v poli, příklad



index	1	2	3	4	5	6	7	8	9	10
klíč	10	8	7	5	2	1	6	3	5	1
	vnitřní uzly					listy				

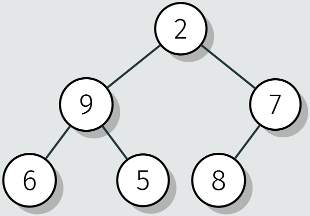
Konstrukce haldy

Haldu lze kostruovat dvěma způsoby:

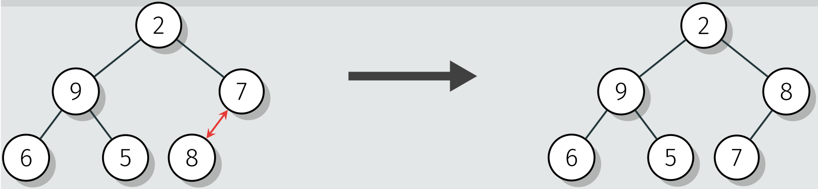
1. **zdola nahoru** a
2. shora dolů.

Konstrukce haldy zdola nahoru – příklad

Výchozí stav haldy

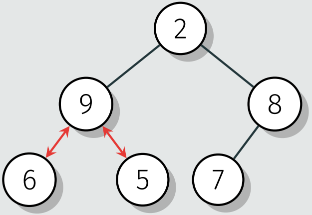


Krok 1

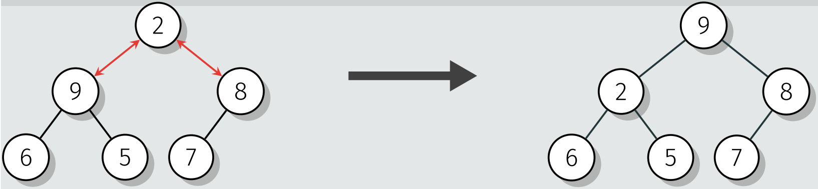


Konstrukce haldy zdola nahoru – příklad (pokrač.)

Krok 2

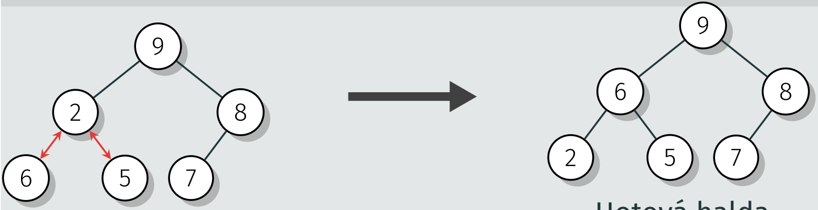


Krok 3a



Konstrukce haldy zdola nahoru – příklad (pokrač.)

Krok 3b



Hotová halda

Konstrukce haldy zdola nahoru

Vstup : Pole $A[0 \dots n-1]$ s definovaným uspořádáním na prvcích pole, i kořen budované haldy

Výstup: Halda s kořenem na indexu i

```
procedure Heapify(A, n, i)
    largest ← i;
    l ← 2 * i + 1;
    r ← 2 * i + 2;
    if l < n ∧ A[l] > A[largest] then  largest ← l ;
    if r < n ∧ A[r] > A[largest] then  largest ← r ;
    if largest ≠ i then
        Swap (A[i], A[largest]);
        Heapify (A, n, largest);
    end
end
```

Konstrukce haldy zdola nahoru

Vstup : Pole $A[0 \dots n-1]$ s definovaným uspořádáním na prvcích pole

Výstup: Halda v poli A

```
procedure MakeHeap(A, n)
    for i ← ⌊n/2⌋ − 1 downto 0 do
        Heapify(A, n, i);
    end
end
```

Konstrukce haldy zdola nahoru – časová složitost

Pro jednoduchost předpokládejme, že $n = 2^k - 1$, tj. halda tvoří úplný binární strom.

Výška haldy je pak $h = \lfloor \log_2 n \rfloor$, což lze psát jako

$$\begin{aligned} \lceil \log_2(n+1) \rceil - 1 &= \lceil \log_2(2^k - 1 + 1) \rceil - 1 \\ &= \lceil \log_2(2^k) \rceil - 1 \\ &= k - 1 \end{aligned}$$

Poznámka

Výraz $\lceil \log_2(n+1) \rceil$ lze interpretovat jako „výška haldy s $n+1$ prvky“. Předpokládali jsme úplný binární strom $\Rightarrow$ strom s $n+1$ prvky má určitě o jednu úroveň více než strom s n prvky.

Každý klíč z úrovně i se bude při konstrukci haldy, v nejhorší případě, posunovat až do listu, tj. na úroveň h.

Posun o jednu úroveň vyžaduje dvě porovnání:

1. nalezení většího z obou potomků a
2. test, zda je nutná výměna s rodičem.

Konstrukce haldy zdola nahoru – časová složitost (pokrač.)

Počet porovnání je tedy $2(h-i)$.

Celkový počet porovnání bude v nejhorším případě roven

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{h-1} \sum_{\text{klíče úrovně } i} 2(h-i) \\
&= \sum_{i=0}^{h-1} 2(h-i)2^i = 2h \sum_{i=0}^{h-1} 2^i - 2 \sum_{i=0}^{h-1} i2^i \\
&= 2n - 2\log_2(n+1))
\end{aligned}
$$

Konstrukce haldy s n prvky vyžaduje, v nejhorším případě, méně než $2n$ porovnání.

Poznámka

V odvození jsme použili vzorce:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

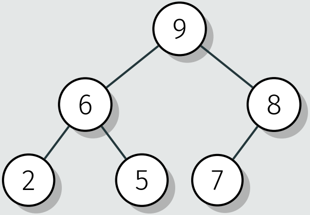
$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

Konstrukce haldy shora dolů

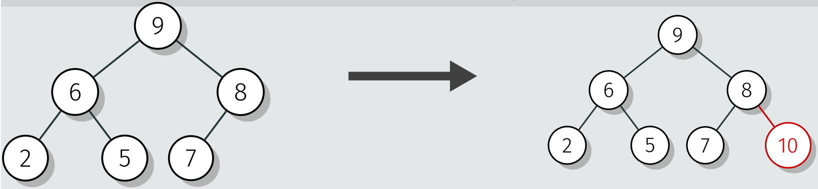
- Opakované vkládání nového klíče do již existující haldy.
  1. Nový klíč vložíme na konec haldy.
  2. Nový klíč porovnáme s rodičem a případně nový klíč přesuneme o patro výš.
  3. Takto postupujeme dokud nenarazíme na většího rodiče nebo dojdeme do kořene haldy.
- Výška haldy s n prvky je $\approx \log_2 n$, tudíž složitost vložení klíče do haldy je $O(\log n)$.
- Konstrukce shora dolů je tedy složitější než zdola nahoru.

Konstrukce haldy shora dolů – příklad

Výchozí stav haldy

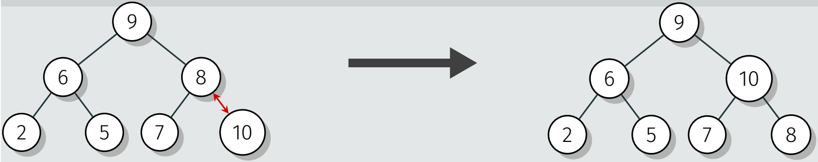


Krok 1 – vložení klíče 10 na konec haldy

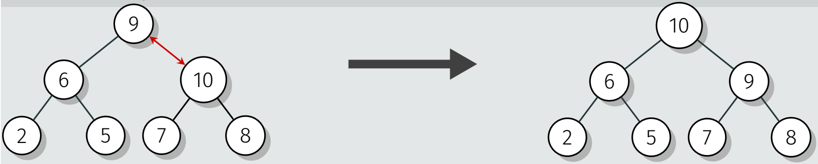


Konstrukce haldy shora dolů – příklad (pokrač.)

Krok 2a – porovnání klíče 10 s rodičem



Krok 2b – porovnání klíče 10 s rodičem



Odstranění největšího klíče z haldy

Princip algoritmu:

1. Výměna klíče v kořeni s klíčem na konci haldy.
2. Zmenšení haldy o jedna.
3. Obnova haldy – otestovat, zda je klíč v rodiči větší než klíče v obou potomcích a případně provést výměnu. Postup opakovat dokud nebude rodičovský klíč větší než klíče v potomcích.

Poznámka

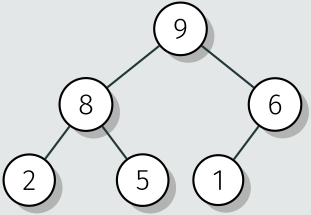
Principiálně, lze z haldy odebrat jakýkoliv klíč. Ale tato operace nemá praktický význam.

Odstranění největšího klíče z haldy – složitost algoritmu

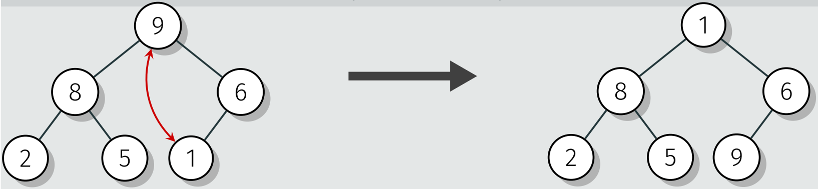
- Počet porovnání nutných pro obnovení haldy je úměrný výšce haldy – „posunujeme“ klíč z kořene po patrech dolů.
- Porovnáváme vždy rodiče s oběma potomky – musíme najít největšího z dané trojice.
- Výška haldy je $h \approx \log_2 n$, počet porovnání nebude tedy větší než $2h$.
- Složitost algoritmu je tedy $O(\log n)$.

Odstranění největšího klíče z haldy – příklad

Výchozí stav haldy

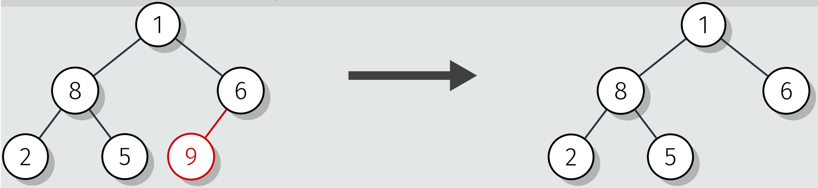


Krok 1 – záměna kořene s posledním prvkem

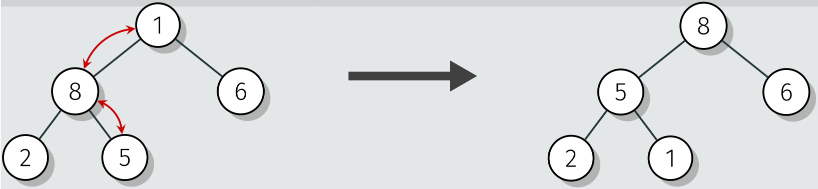


Odstranění největšího klíče z haldy – příklad (pokrač.)

Krok 2 – odstranění posledního uzlu



Krok 3 – obnovení haldy



Třídění haldou – HeapSort

Algoritmus pracuje ve dvou fázích:

Konstrukce haldy : pro dané pole je sestavena halda.

Odstranění maxima : $(n-1)$-krát je aplikován algoritmus pro odstranění největšího klíče z, postupně se zmenšující, haldy.

Vstup : Pole $A[0 \ldots n-1]$ s definovaným uspořádáním na prvcích pole

Výstup: Setříděné pole A

```
procedure HeapSort(A, n)
    MakeHeap(A, n);
    for i ← n − 1 downto 0 do
        Swap(A[0], A[i]);
        Heapify(A, i, 0);
    end
end
```

Třídění haldou – složitost algoritmu

- Složitost první fáze je $O(n)$.
- Ve druhé fázi postupně odstraňujeme největší klíč z haldy o klesající velikosti $n, n-1, \dots, 2$. Počet porovnání $C(n)$ je

$$
\begin{aligned}
C(n) &\le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \dots + 2\lfloor \log_2 1 \rfloor \\
&\le 2\sum_{i=1}^{n-1} \log_2 i \\
&\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n
\end{aligned}
$$

Platí tedy $C(n) \in O(n \log n)$.

Třídění haldou – složitost algoritmu (pokrač.)

- Pro obě fáze dostáváme $O(n) + O(n \log n) = O(n \log n)$.
- Další analýzou složitosti lze dokázat, že stejná složitost platí i pro průměrný případ. Tedy $\Theta(n \log n)$.
- Třídění haldou je srovnatelné s tříděním sléváním.
- V praxi je však pomalejší než QuickSort.

Zdroje pro samostatné studium

- Kniha [1], kapitola 6.4, strany 226 – 232
- Kniha [2], kapitoly 6.1 až 6.4, strany 161 – 172

Strategie řešení transformuj a vyřeš

Hornerovo schéma

Hodnota polynomu v bodě

Zadání

Máme dán polynom

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Naším úkolem je vypočítat hodnotu polynomu $p(x)$ v bodě x_0.

Motivace

- Polynomy se využívají pro aproximaci funkcí aneb
  1. Jak procesor vypočte hodnotu funkce $\sin(x)$?
  2. Kde se vzaly hodnoty funkce $\sin(x)$ v matematických tabulkách?

  Pomocí Taylorova rozvoje funkce, což je polynom!
- Rychlá Fourierova trasformace

Funkci $f(x)$, která má v bodě a konečné derivace do řádu $n+1$ lze, v okolí bodu a, psát jako rozvoj

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_{n+1}^{f,a}(x)$$

Pro $a = 0$ se rozvoj nazývá Maclaurinův

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + R_{n+1}^{f,0}(x)$$

Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0

$$\sin(x) = \sin(0) + \frac{\sin'(0)}{1!}x + \frac{\sin''(0)}{2!}x^2 + \cdots + \frac{\sin^{(n)}(0)}{n!}x^n + R_{n+1}^{\sin,0}(x)$$

Derivace

$$\sin^{(1)} 0 = \cos 0 = 1 \qquad \sin^{(2)} 0 = -\sin 0 = 0$$

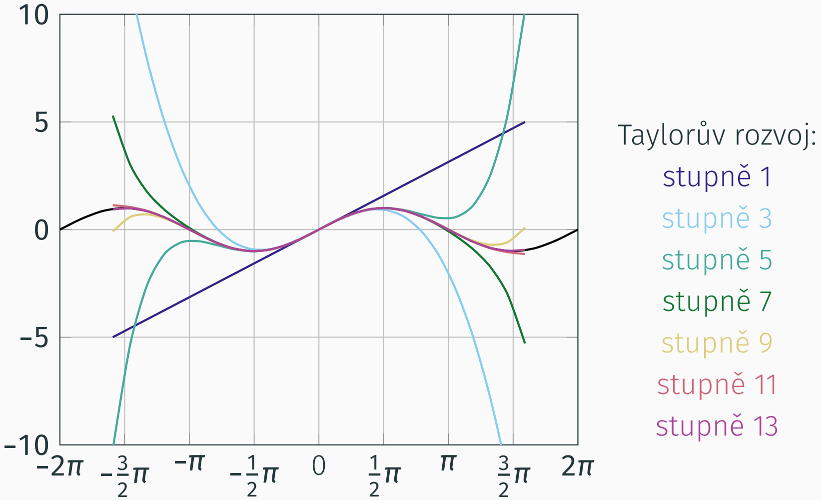
$$\sin^{(3)} 0 = -\cos 0 = -1 \qquad \sin^{(4)} 0 = \sin 0 = 0$$

$$\sin(x) = 0 + \frac{1}{1!}x + \frac{0}{2!}x^2 + \frac{-1}{3!}x^3 + \frac{0}{4!}x^4 + \cdots + R_{n+1}^{\sin,0}(x)$$

Aproximace polynomem 13-tého stupně

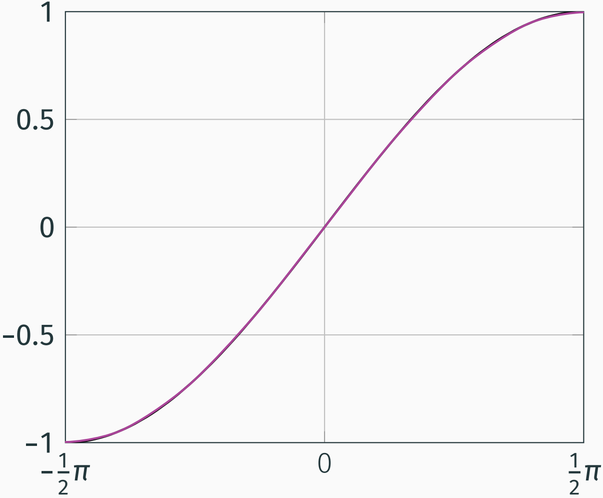
$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!}$$

Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0

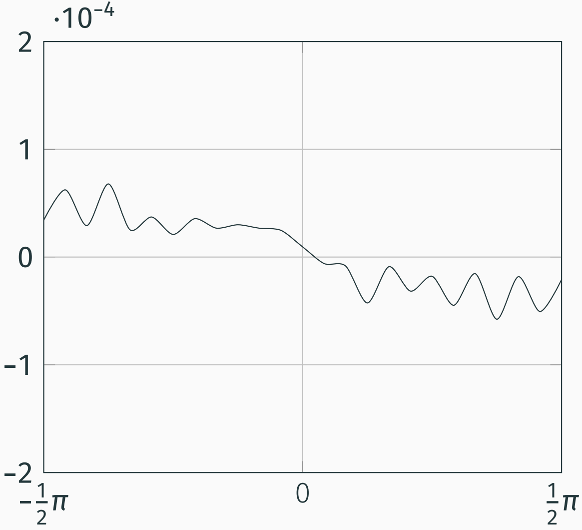


Funkce $y = \sin(x)$ je zobrazena černě.

Taylorův rozvoj funkce $y = \sin(x)$ stupně 13 v bodě 0



Taylorův rozvoj funkce $y = \sin(x)$ v bodě 0, chyba aproximace



Tabulky hodnot funkcí

- Taylorovým rozvojem lze aproximovat hodnotu požadované funkce a sestavit tabulky.
- Ruční výpočet – náročné a zatíženo obrovským množstvím chyb.
- Průlomová myšlenka – k numerickým výpočtům není nutná inteligence! Lze je provádět **strojově**!

7.4 **cos x** (x v radiánech)

x	0	1	2	3	4	5	6	7	8	9
0,0	1,0000	1,0000	0,9998	9996	9992	9988	9982	9976	9968	9960
0,1	0,9950	9940	9928	9916	9902	9888	9872	9856	9838	9820
0,2	9801	9780	9759	9737	9713	9689	9664	9638	9611	9582
0,3	9553	9523	9492	9460	9428	9394	9359	9323	9287	9249
0,4	9211	9171	9131	9090	9048	9004	8961	8916	8870	8823
0,5	8776	8727	8678	8628	8577	8525	8473	8419	8365	8309
0,6	8253	8196	8139	8080	8021	7961	7900	7838	7776	7712
0,7	7648	7584	7518	7452	7385	7317	7248	7179	7109	7038
0,8	6967	6895	6822	6749	6675	6600	6524	6448	6372	6294
0,9	6216	6137	6058	5978	5898	5817	5735	5653	5570	5487
1,0	0,5403	5319	5234	5148	5062	4976	4889	4801	4713	4625
1,1	4536	4447	4357	4267	4176	4085	3993	3902	3809	3717
1,2	3624	3530	3436	3342	3248	3153	3058	2963	2867	2771
1,3	2675	2579	2482	2385	2288	2190	2092	1994	1896	1798
1,4	1700	1601	1502	1403	1304	1205	1106	1006	0907	0807
1,5	0707	0608	0508	0408	0308	0208	0108	0008		
1,5									−0,0092	−0,0192
1,6	−0,0292	0392	0492	0592	0691	0791	0891	0990	1090	1189
1,7	−0,1288	1388	1486	1585	1684	1782	1881	1979	2077	2175
1,8	−0,2272	2369	2466	2563	2660	2756	2852	2948	3043	3138
1,9	−0,3233	3327	3421	3515	3609	3702	3795	3887	3979	4070
2,0	−0,4161	4252	4342	4432	4522	4611	4699	4787	4875	4962
2,1	−0,5048	5135	5220	5305	5390	5474	5557	5640	5722	5804
2,2	−0,5885	5966	6046	6125	6204	6282	6359	6436	6512	6588
2,3	−0,6663	6737	6811	6883	6956	7027	7098	7168	7237	7306
2,4	−0,7374	7441	7508	7573	7638	7702	7766	7828	7890	7951
2,5	−0,8011	8071	8130	8187	8244	8301	8356	8410	8464	8517
2,6	−0,8569	8620	8670	8720	8768	8816	8863	8908	8953	8998
2,7	−0,9041	9083	9124	9165	9204	9243	9281	9318	9353	9388
2,8	−0,9422	9455	9487	9519	9549	9578	9606	9633	9660	9685
2,9	−0,9710	9733	9755	9777	9797	9817	9836	9853	9870	9885
3,0	−0,9900	9914	9926	9938	9948	9958	9967	9974	9981	9987
3,1	−0,9991	9995	9998	9999	−1,0000	−1,0000				

k	1	2	3	4	5	6	7	8	9	10
$k \cdot 2\pi$	6,283	12,566	18,850	25,133	31,416	37,699	43,982	50,265	56,549	62,832

$\cos(-x) = \cos x$ $\cos x = \cos(x + k \cdot 2\pi),\ k \in \mathbb{Z}$

sin x, tg x

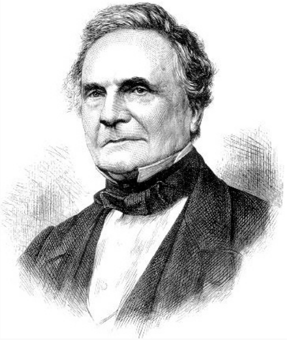
87

Charles Babbage – Difference Engine

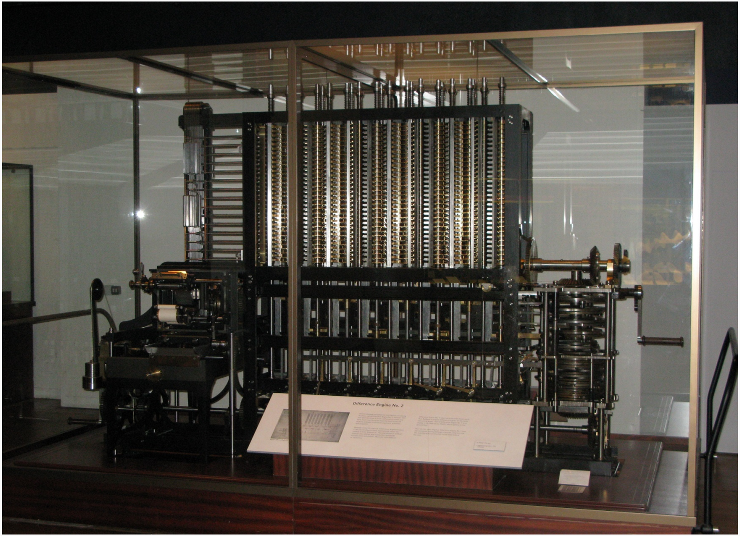
Difference Engine

- první programovatelný počítač na světě
- 1819 – zahájení prací
- 1822 – dokončen prototyp
- 1823 – zahájeny práce na velkém stroji
- 1833 – přerušení prací
- 1842 – ukončení vládní podpory, na projekt vynaloženo 17 tisíc liber, stroj nebyl nikdy dokončen
- 1991 – funkční replika!

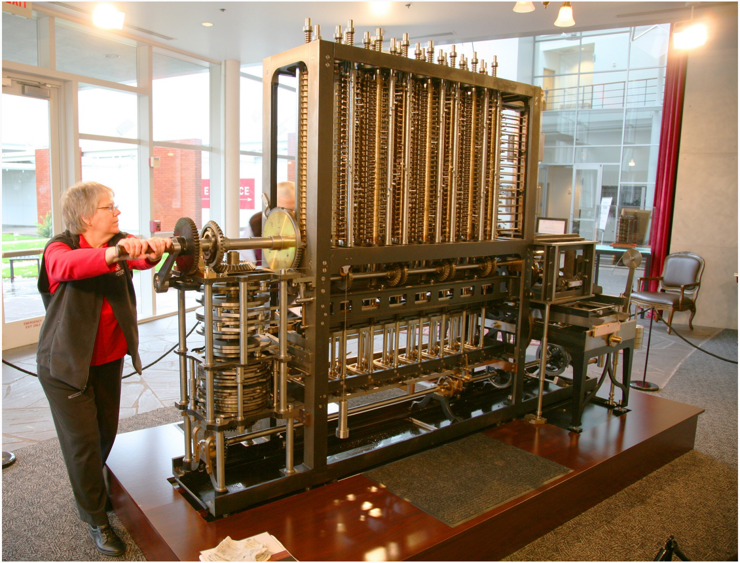
Charles Babbage
(1791 – 1871)



Difference Engine

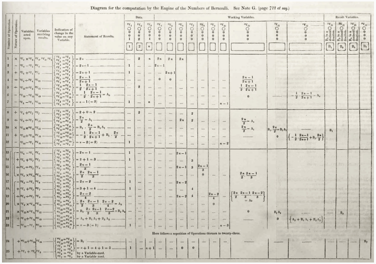


Difference Engine



Augusta Ada King, hraběnka z Lovelace (1815 – 1852)

Programátorka **Analytical Engine**, (Babbage 1837), což byl první obecně použitelný turingovsky úplný počítač.



Hornerovo schéma – transformace

Základní myšlenka:

- transformace polynomu na jiný tvar,
- postupně vytýkáme z částí polynomu proměnnou x.

$$
\begin{aligned}
p(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} + a_nx^n \\
&= a_0 + x\left(a_1 + a_2x + \dots + a_{n-1}x^{n-2} + a_nx^{n-1}\right) \\
&= a_0 + x\left(a_1 + x\left(a_2 + \dots + a_{n-1}x^{n-3} + a_nx^{n-2}\right)\right) \\
&\vdots \\
&= a_0 + x\Big(a_1 + x\big(a_2 + \dots + x(a_{n-1} + a_nx)\dots\big)\Big)
\end{aligned}
$$

Že tato rovnost platí, je snadno vidět postupným roznásobením všech závorek.

Hornerovo schéma – výpočet

Hodnotu $p(x_0)$ počítáme „z vnitřku" závorek, postupně počítáme hodnoty b_i

$$
\begin{aligned}
b_n &= a_n \\
b_{n-1} &= a_{n-1} + b_n x_0 \\
b_{n-2} &= a_{n-2} + b_{n-1} x_0 \\
&\vdots \\
b_0 &= a_0 + b_1 x_0
\end{aligned}
$$

Hodnota b_0 je pak rovna $p(x_0)$, neboť

$$p(x_0) = a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)\ldots\big)\Big)$$

Hornerovo schéma – výpočet (pokrač.)

a postupným dosazováním za $\boldsymbol{b_i}$ dostáváme

$$
\begin{aligned}
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(a_{n-1} + b_n x_0)\ldots\big)\Big) \\
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0(b_{n-1})\ldots\big)\Big) \\
p(x_0) &= a_0 + x_0(b_1) \\
p(x_0) &= b_0
\end{aligned}
$$

Hornerovo schéma – ruční výpočet

Vypočtěte hodnotu polynomu $p(x) = 2x^3 - 6x^2 + 2x - 1$ v bodě $x_0 = 3$.

x_0	x^3	x^2	x^1	x^0
3	2	-6	2	-1
		6	0	6
	2	0	2	5

Běžný výpočet

$$
\begin{aligned}
p(3) &= 2 \times 3^3 - 6 \times 3^2 + 2 \times 3 - 1 \\
&= 2 \times 27 - 6 \times 9 + 2 \times 3 - 1 \\
&= 54 - 54 + 6 - 1 = 5
\end{aligned}
$$

ALGORITHM *Horner*$(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n-1$ **downto** 0 **do**
  $p \leftarrow x * p + P[i]$
return p

Hornerovo schéma – časová složitost algoritmu

Je zřejmé, že počet násobení $M(n)$ a počet sčítání $A(n)$ je roven

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

Výpočet hrubou silou

Jen pro výpočet $a_n x^n$ je zapotřebí:

- $n - 1$ násobení pro výpočet mocniny
- 1 násobení pro vynásobení a_n.

Za shodný počet násobení zvládne Hornerův algoritmus vypočítat i zbývajících $n - 1$ členů polynomu!!!

Zdroje pro samostatné studium

- Kniha [1], kapitola 6.5, strany 234 – 239
- Kniha [2], kapitola 30.1, strany 879 – 880

Strategie řešení transformuj a vyřeš

Redukce problému

Smyslem redukce je převedení řešeného problému na problém jiný, který umíme vyřešit.

Postup redukce

1. **Problém 1** – to, co chceme řešit
2. Redukce **Problému 1** na **Problém 2**
3. **Problém 2** – řešitelný algoritmem A
4. Provedení algoritmu A
5. **Řešení Problému 2**

Nejmenší společný násobek

Nejmenší společný násobek $lcm(m, n)$ dvou přirozených čísel m a n definujeme jako nejmenší přirozené číslo, které je dělitelné m a n zároveň.

Řešení pomocí prvočíselného rozkladu

$$\begin{aligned} 24 &= 2^3 \cdot 3^1 \\ 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ \mathrm{lcm}(24, 60) &= 2^3 \cdot 3^1 \cdot 5^1 = 120 \end{aligned}$$

Řešení pomocí největšího společného dělitele

Lze dokázat, že

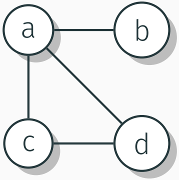
$$\mathrm{lcm}(m, n) = \frac{mn}{\mathrm{gcd}(m, n)}$$

$\mathrm{gcd}(m, n)$ lze vypočítat, efektivním, Euklidovým algoritmem.

Počet sledů v grafu

Zadání: Vypočítat počet sledů mezi dvojicemi vrcholů v daném grafu G.

Řešení: Lze dokázat, že počet různých sledů délky k mezi vrcholy i a j je roven prvku a_{ij} matice $\mathbf{A}^k$, kde A je matice sousednosti grafu G.



$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \quad \mathbf{A}^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

Z a do a vedou tři sledy délky 2: $a - b - a$, $a - c - a$, $a - d - a$

Z a do c vede jeden sled délky 2: $a - d - c$

Maximalizační problém – nalezení maxima funkce $f(x)$

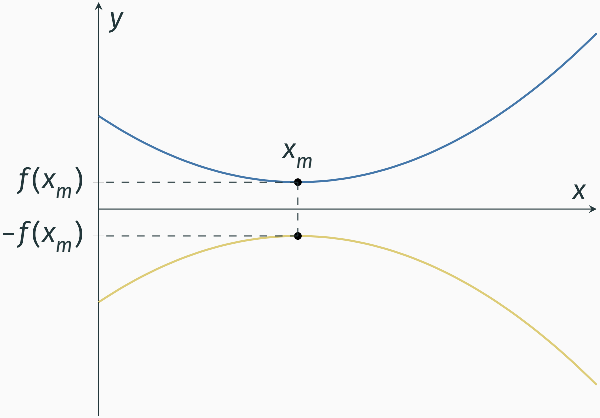
Minimalizační problém – nalezení minima funkce $f(x)$

Jak řešit situaci?

- Máme minimalizovat funkci $f(x)$, ale
- k dispozici máme pouze maximalizační algoritmus.

Lze využít maximalizační algoritmus pro minimalizační problém? Případně naopak?

Redukce optimalizačních problémů



$$\min f(x) = -\max\left[-f(x)\right]$$
$$\max f(x) = -\min\left[-f(x)\right]$$

Koza, vlk a zelí

- Na břehu řeky je převozník, koza, vlk a zelí.
- Převozník má převézt kozu, vlka a zelí na druhý břeh pomocí loďky.
- Na loďku se, mimo převozníka, vejde nejvýše jedna z převážených entit.
- Na stejném břehu se nesmí bez převozníkova dozoru ocitnout dvojice (koza, zelí) a (vlk, koza).
- Úkolem je sestavit plán převozu nebo dokázat, že řešení neexistuje.

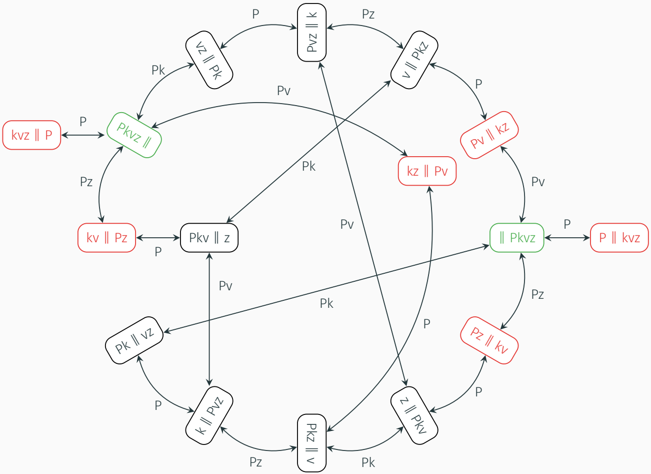
Nejstarší písemná podoba úlohy pochází z 9. století...

Koza, vlk a zelí – stavový prostor

Stav – reprezentuje obsazení obou břehů oddělených řekou, např. Pkv||z

Přechod mezi stavy – cesta z jednoho břehu řeky na druhý, s případným převozem

Koza, vlk a zelí – graf stavového prostoru



Řešení problému – nalezení orientované cesty z počátečního stavu do koncového stavu průchodem do šířky.

Zdroje pro samostatné studium

- Kniha [1], kapitola 6.6, strany 240 – 248

Děkuji za pozornost