



# 9. prednáška (25.4.2022)

## Hashovanie

alebo

ako rýchlo hľadať

alebo

**Agregovaná zložitost', HasSet, rolling Hash**



# Hľadanie prvku v poli

- Ako zistiť, či pole p obsahuje prvok hladanyPrvok?

```
public static boolean nachadzaSa(int[] p, int hladanyPrvok) {  
    for (int i = 0; i < p.length; i++) {  
        if (p[i] == hladanyPrvok)  
            return true;  
    }  
    return false;  
}
```

```
public static boolean nachadzaSa(String[] p, String hladanyPrvok) {  
    for (int i = 0; i < p.length; i++) {  
        if (p[i].equals(hladanyPrvok))  
            return true;  
    }  
    return false;  
}
```



# Hľadanie prvku v poli

- Ako zistiť, či pole  $p$  obsahuje prvok  $hladanyPrvok$ ?

```
public static boolean nachadzaSa(int[] p, int hladanyPrvok) {  
    for (int i = 0; i < p.length; i++) {  
        if (p[i] == hladanyPrvok)  
            return true;  
    }  
    return false;  
}
```

- Aká je časová zložitost' hľadania?
- $O(n)$



# Hľadanie prvku v poli

- Ak máme veľa hľadání ako viem hľadať rýchlejšie?
- 2. prednáška - Binárne vyhľadávanie
  1. Utriedime
    - $O(n \log n)$  QuickSort, MergeSort, HeapSort
    - $O(n^2)$  BubbleSort, SelectionSort
  2. Každé binárne vyhľadávanie v čase  $O(\log n)$

**Ušetrili sme čas, zarobili peniaze.**





**Ušetrili sme čas, zarobili peniaze.**

**ak nie sú otázky...**

**Ďakujem za pozornosť!**





# Hľadanie prvku v spájanom zozname

- Spájaný zoznam
  - hľadanie prvku v čase  $O(n)$
- Výhody spájaného zoznamu oproti poľu?
  - dynamická štruktúra
- Počet operácií v najhoršom prípade (= keď máme smolu):
  - get -  $O(n)$ , set -  $O(n)$
  - pridanie na začiatok a na koniec:  $O(1)$
  - pridanie/odobranie (bez nájdania pozície):  $O(1)$



# Pole s kapacitou

- Pole s kapacitou - pole, veľkosť (size)
- Výhody poľa s kapacitou
  - ak kapacita stačí, **add (remove)** na koniec zoznamu vieme realizovať v čase  $O(1)$ 
    - zvýšenie premennej s veľkosťou (size) o 1
    - uloženie hodnoty na príslušný index interného poľa
  - ak kapacita nestačí, musíme vyrobiť nové pole a kopírovať, t.j. časová zložitost' je  $O(n)$
- Ak je veľa neobsadených políčok, zmenšíme pole
- get -  $O(1)$ , set -  $O(1)$
- hľadanie -  $O(n)$



# Pole s kapacitou

- Pole s kapacitou - pole, veľkosť (size)
- Problémy pri zväčšovaní a zmenšovaní poľa:
  1. Ak je pole naplnené (veľkosť == kapacita) a chceme pridať ďalší prvok na koniec zoznamu tak kapacita nestačí, musíme vyrobiť nové pole a kopírovať obsah starého poľa, t.j. časová zložitost' je  $O(n)$
  2. Ak má pole veľa neobsadených políčok tak vyrobíme nové menšie pole a obsah starého poľa prekopírujeme, t.j. časová zložitost' je  $O(n)$
- Kedy a ako budeme pole zväčšovať a zmenšovať?
  - Ak je pole naplnené, tak jeho kapacitu zväčšíme na dvojnásobnú
  - Ak je pole obsadené na  $\frac{1}{4}$ , tak jeho kapacitu zmenšíme na polovicu





# Pole s kapacitou

- Pole s kapacitou - pole, veľkosť (size)
- Máme stále to isté pole s kapacitou a vieme ako funguje. Aká je zložitosť pridávania a odstraňovania prvkov podľa dokumentácie napríklad v implementácii ArrayList?

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires  $O(n)$  time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.



Prečo dokumentácia uvádza, že zložitosť je konštantná, teda  $O(1)$ ?

Dokumentácia uvádza amortizovanú zložitosť



# Amortizovaná zložitosť

- Kúpili sme si auto ktorého cena je 4000 eur
- Na jeho ročnú prevádzku minieme 1000 eur
- Za auto budeme platiť 4 splátky v nasledujúcich 4 rokoch
- Koľko nás stojí ročná prevádzka auta?
- $(4000+4*1000)/4=2000$



Ročná amortizácia



# Amortizovaná zložitost'

- Pridať  $n$  prvkov do prázdneho poľa s kapacitou, kde začiatočná kapacita je 1, trvá  $O(n)$ .
- Dôkaz (myšlienka):
  - Predpokladajme, že máme dve operácie vieme iba **vložiť prvok** ak je kapacita dostatočná a vieme **zväčšiť pole** (zdvojnásobiť)
  - Jedno vkladanie trvá čas  $O(1)$ , vkladanií vykonáme práve  $n$ , všetky vkladania trvajú spolu  $O(n)$
  - Ku zväčšovaniu dochádza práve vtedy, keď je aktuálny počet prvkov mocnina dvojky:  $2^0, 2^1, 2^2, \dots, 2^k$ , kde  $k$  je najväčšia mocnina dvojky taká, že  $2^k < n$
  - Tieto zväčšovania trvajú čas  $O(2^0 + 2^1 + 2^2 + \dots + 2^k)$  v zátvorke je geometrický rad, ktorého súčet je  $2^{k+1} - 1 < 2n$
  - Výsledná zložitost' je  $O(n) + O(2n) = O(n)$

čas kopírovania

$$2^k < n$$

$$2^{k+1} < 2n$$



# Amortizovaná zložitosť

- Pridať  $n$  prvkov do prázdneho poľa s kapacitou, kde začiatočná kapacita je 1, trvá  $O(n)$
- Podobne vieme ukázať, že ak vykonáme  $n$  odobraní prvkov z poľa, časová zložitosť bude opäť  $O(n)$

V priemernom čase trvá jedna operácia čas  $O(1)$

- Pri amortizovanej zložitosti nás nezaujímajú časová zložitosť vykonania operácie raz ale priemerná časová zložitosť operácie pri opakovanom vykonávaní



# Amortizovaná zložitost'

- Vieme ukázať, že vykonanie ľubovoľných  $n$  operácií v poli s kapacitou bude trvať  $O(n)$

Postupnosť operácií rozdelíme na bloky, kde každý blok končí zmenou kapacity.  
Viac na cvičeniach?

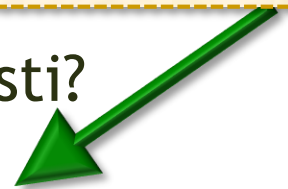
- Pomôže nám amortizovaná zložitost' pri riešení dnešného problému ako rýchlo vyhľadávat'?



# Pole s kapacitou

- Pole s kapacitou
- get -  $O(1)$ , set -  $O(1)$ ,  
add, remove -  $O(1)$  „občas“  $O(n)$ , amort.  $O(1)$ ,
- hľadanie -  $O(n)$
- Ako viem hľadať rýchlejšie?
- Držať utriedené pole
  - Binárne vyhľadávanie  $O(\log n)$
  - Môžeme používať indexy? Zmenia sa zložitosti?
  - Aby sme udržali pole vždy utriedene:
    - set -  $O(n)$ , add -  $O(n)$ , remove -  $O(n)$

Po každej operácii musíme znovu usporiadať





# Pole s „veštcom“

- Majme pole a „veštca“
- Veštec vie o každom prvku povedať na aký index vo zvolenom poli patrí







# Programujeme veštca

- Veštca nahradíme funkciou z množiny slov na indexy v poli
- Funkcia nám vypočíta hash pre daný reťazec, preto ju budeme nazývať hash-ovacou funkciou







# Hash-ovacia funkcia

```
public static int zaHashujRetazec1(String s, int velkost) {  
    int sucet = 0;  
    for (int i = 0; i < s.length(); i++) {  
        sucet += (int)(s.charAt(i));  
    }  
    int hash = sucet % velkost;  
    return hash;  
}
```








- Sčítame unicode hodnoty znakov z reťazca `s` a vypočítame zvyšok súčtu po delení `velkost`-ou
- *Hash-ovacou funkciou (hash function)* nazveme každú funkciu, ktorá priradzuje dátam ľubovoľnej veľkosti hodnotu z konečnej množiny



# Ukladáme na základe Hash-u

- Do 7 prvkového poľa chceme uložiť mená:  
Ali, Daniela, Marek, Nicol, Sophia, Daniel, Sona

0	Daniela
1	Daniel
2	Sona
3	Sophia
4	Nicol
5	Ali
6	Marek

meno	funkcia	index
Ali		5
Daniela		0
Marek		6
Nicol		4
Sophia		3
Daniel		1
Sona		2

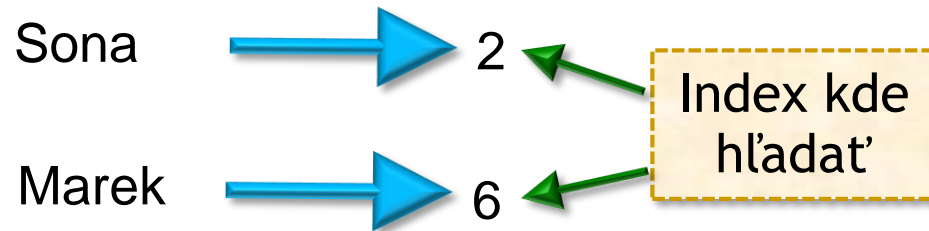


# Ukladáme na základe Hash-u

- Majme čiastočne naplnené pole
- Nachádza sa v poli Sona?
- Nachádza sa v poli Marek?

0	
1	Daniel
2	
3	Sophia
4	
5	
6	Marek

meno      funkcia      index







Hľadanie v konštantnom čase



# Ukladáme na základe Hash-u

- Chceme pridať ďalších ľudí  
Matej, Boris, Zuzka, Adam

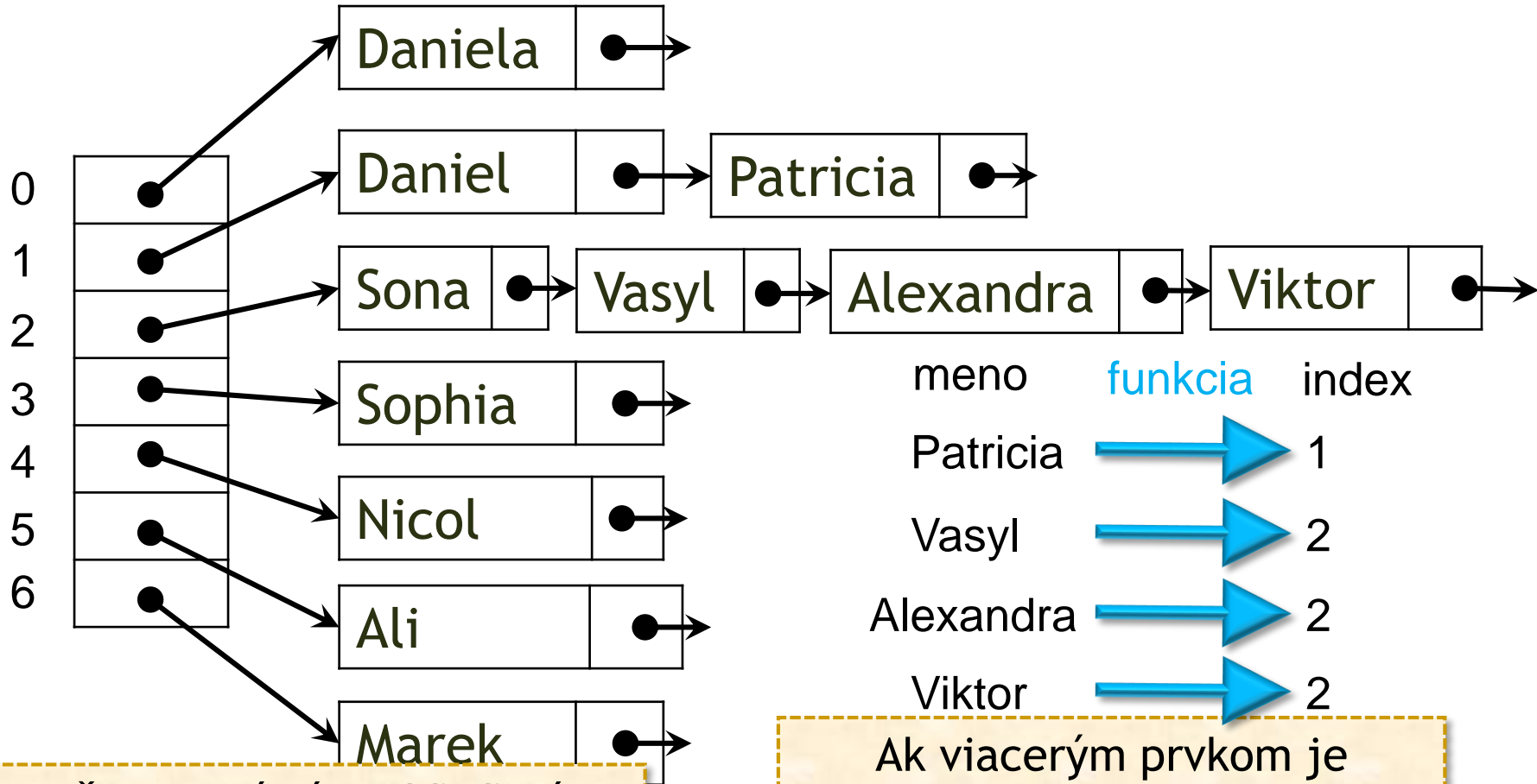
0	Daniela
1	Daniel
2	Sona
3	Sophia
4	Nicol
5	Ali
6	Marek

meno	funkcia	index
Patricia		1
Vasyl		2
Alexandra		2
Viktor		2



# Ako uložit' viac hodnôt?

- Pole spájaných zoznamov



meno	funkcia	index
Patricia	→	1
Vasyl	→	2
Alexandra	→	2
Viktor	→	2

Počet operácií pri hľadání v najhoršom prípade:  $O(n)$

Ak viacerým prvkom je priradený ten istý index nastala: kolízia



# Kolizie

- Kolízia nastáva ak dve objekty chceme priradiť na to isté miesto
- Vieme to riešiť napr. pomocou spájaného zoznamu alebo iných štruktúr
- Ako ovplyvnia kolízie zložitosť?  
V najhoršom prípade je zložitosť  $O(n)$



# Ako riešiť kolízie?

- Vyhýbať sa kolíziám
  - Mat' dostatočnú kapacitu aby v ideálnom prípade mohol byť každý prvok sám
- Realita: Pole nenaplníme na viac  $\frac{3}{4}$  kapacity (defaultna hodnota pre HashMap a HashSet)
- Pomer medzi aktuálnym počtom prvkov a kapacitou sa nazýva *faktor naplnenia* alebo *hustota* alebo *Load Factor*

Nahliadnime do dokumentácie



# *Load faktor*

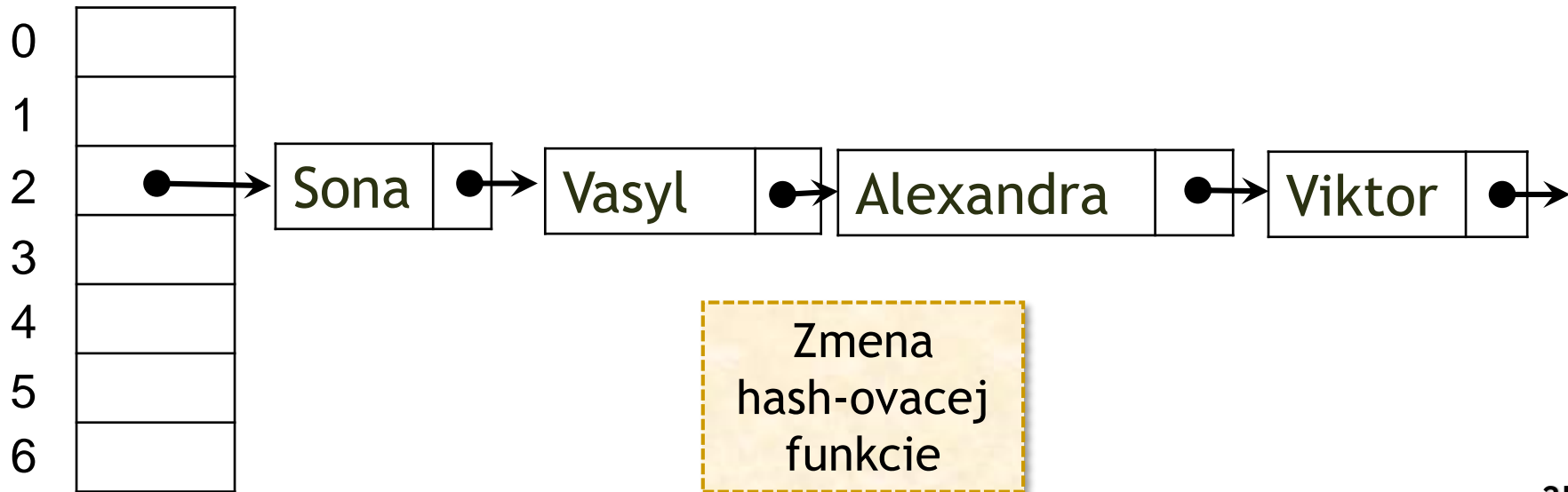
- Load faktor ovplyvňuje množstvo nevyužitej pamäte
- Malý load faktor - veľa nevyužitej pamäte
- Veľký load faktor - väčšia zložitost'





# Ako riešiť kolízie?

- Vyhýbať sa kolíziám
  - Mať dostatočnú kapacitu aby v ideálnom prípade mohol byť každý prvok sám
- Aj keď máme dostatočne malý faktor naplnenia môže nastať veľa kolízií





# Zmena hash-ovacej funkcie

- Doteraz:

1. Pre objekt sme vypočítali hodnotu  $x$  (suma kódov znakov)
2. Vypočítali sme  $x \% m$ , kde  $m$  je maximálna kapacita poľa (modulo označujeme  $\%$  alebo  $mod$ )

- Lineárna kongruencia

1. Rovnako spočítame  $x$
2. Zoberieme konštantu  $a$ , následne vypočítame  $ax \% m$

Vďaka rôznej voľbe  $a$  vieme dostať „rodinu“ hash funkcií

Vyskúšajte pre  $a = 2$ ,  $m$  párne

- Ďalšie dobré funkcie dostaneme ak  $m$  je prvočíslo a konštantu  $a$  je blízka  $0,618034 m$ , kde  $1,618034$  je zlatý rez



# Zmena hash-ovacej funkcie

- Zmena orezávania mocou zvýšku - vyššie bity súčtu  

$$\lfloor (ax \% 2^w) / 2^{w-l} \rfloor$$
 pre  $m=2^l$

Vyskúšajte pre  $m=8$  a  $w=5$

- Skalárny súčin  
 $(\sum_i a_i x_i) \bmod n$ , v našom prípade sú  $x_i$  jednotlivé písmená, vo všeobecnosti to môžu byť rôzne prvky postupnosti poľa alebo inej štruktúry
- Polynóm  
 Skalárny súčin vieme podobne nahradiť polynómom:  $(\sum_i a^i x_i) \bmod n$



# Hash function

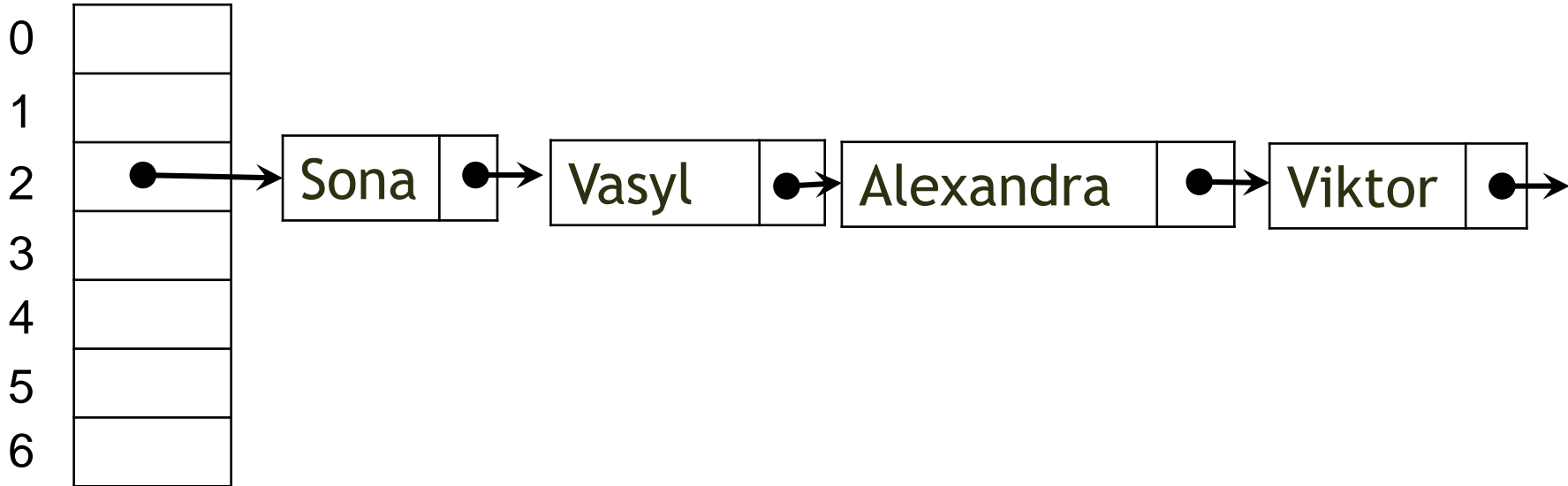
- Aké vlastnosti by mala spĺňať hash-ovacia funkcia
  - Minimalizovať kolízie (viac na pravdepodobnostných a aproximačných algoritmoch)
  - Rovnomerná distribúcia (viac na pravdepodobnosti a štatistike a pravdepodobnostných a aproximačných algoritmoch)
  - Ľahko vypočítateľná

*just remember to*  
**K.I.S.S.**  
Keep It Simple Stupid



# Ako riešiť kolízie?

- Pôvodne

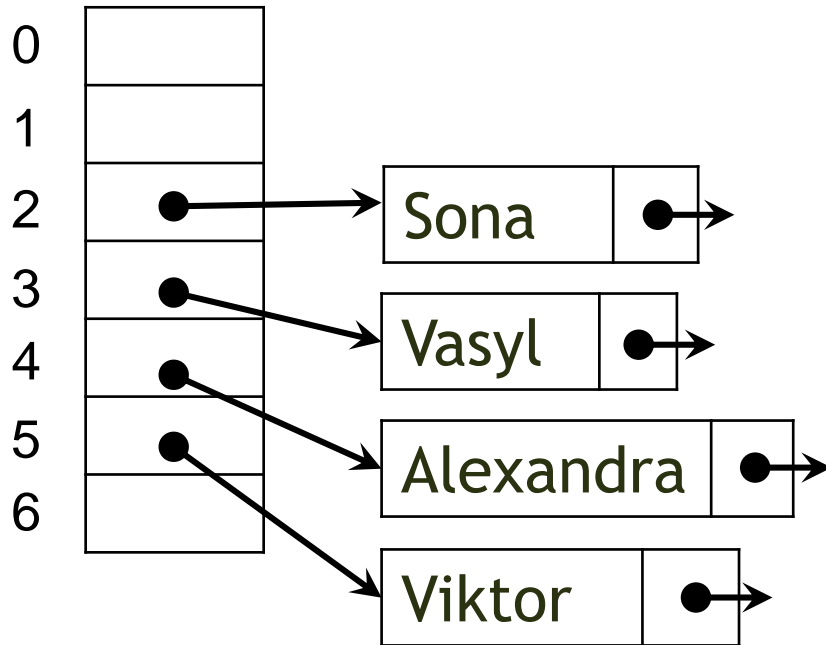


- Zachováme hash-ovaciú funkciu ale dovoľíme otvorené adresovanie



# Ako riešiť kolízie?

- Otvorené adresovanie:



meno	funkcia	index
Sona		2
Vasyl		2
Alexandra		2
Viktor		2

- Prvok sa zapíše na prvú nasledujúcu voľnú pozíciu
- Ako bude prebiehať čítanie?  
Čo ak chcem prvok vymazať?

Počet operácií pri hľadání v najhoršom prípade:  $O(n)$



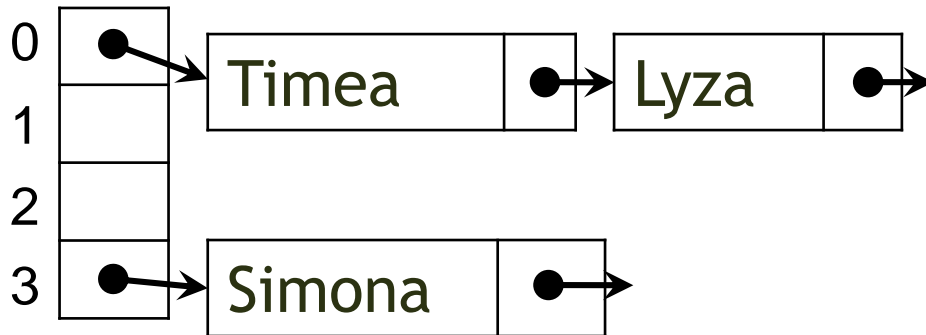
# *Pridávanie a vyhadzovanie*

- Prepodkladajme, že kolízie riešime ukladaním do spájaného zoznamu
- Do nášho HashSet-u vieme pridávať prvky a vyhadzovať z neho prvky
- Ako udržať Load Factor medzi 0,25 a 0,75?



# Rehasing

- Chceme udržať Load Factor medzi 0,25 a 0,75?
- Majme 3 prvky uložené v poli veľkosti 4



- Aktualne je Load Factor  $\frac{3}{4}$
- Chceme pridať prvok Norbert ak by sme ho pridali tak Load Factor bude 1
- Riešenie ako pri poli s kapacitou
  - Vytvoríme nové väčšie pole a prvky „skopírujeme“

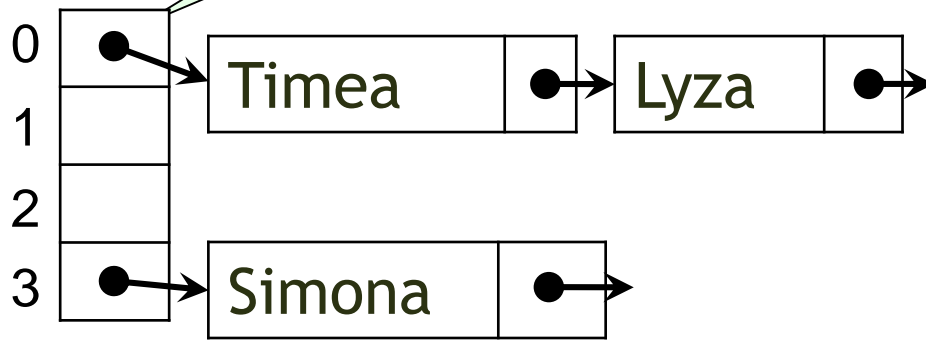




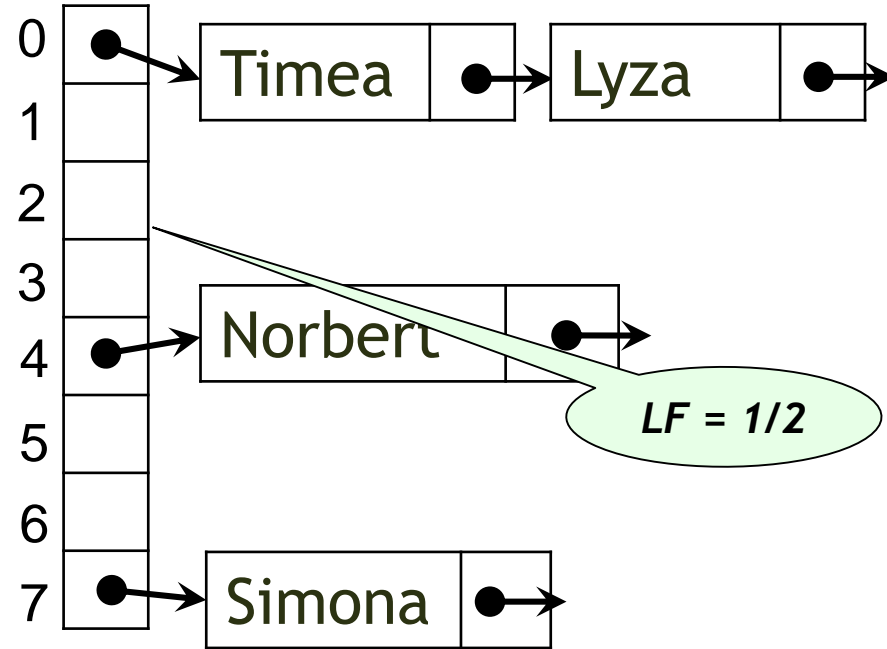
# Rehashing

Prvky sa môžu dostať na iné indexy lebo sa zmenila hashovacia funkcia

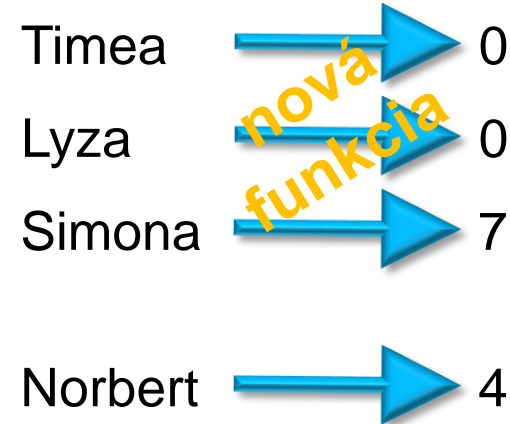
$LF = 3/4$



Hashovacia funkcia je závislá od veľkosti poľa!



$LF = 1/2$



1. Vytvoríme nové väčšie pole
2. Prekopírujeme prvky do nového poľa na základe novej hash funkcie (prehashujeme)
3. Pridáme nový prvok



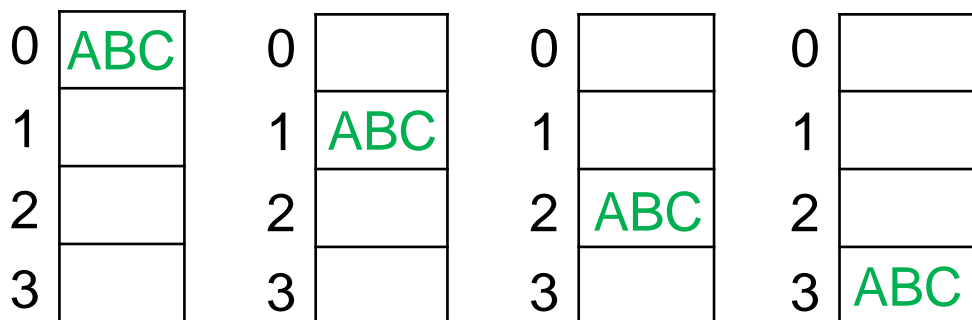
# Rehashing

- Postup pri odstraňovaní je podobný
- Ak by Load Factor mal klesnúť pod hodnotu 0,25 (nami zvolená hodnota), tak vytvoríme menšie pole a prehashujeme
- $LF = 0,25$  ak bude nové pole polovičnej veľkosti tak nový  $LF = 0,5$



# Zložitosť hľadania

- Kolízie riešime pomocou spájaného zoznamu
- Aká bude amortizovaná zložitosť hľadania?
- Príklad pre  $LF=0,75$  a uložené prvky **A**, **B**, **C**



Máme 4 možnosti ako umiestniť všetky prvky do jednej priehradky

Všetky prvky patria do rovnakej priehradky  
**Najhoršia možnosť**



# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky A, B, C

0	AB
1	C
2	
3	

0	AB
1	
2	C
3	

0	AB
1	
2	
3	C

...

0	B
1	
2	AC
3	

...

Dve prvky patria do rovnakej priehradky, jeden prvok je v inej

Dokopy 36 možností

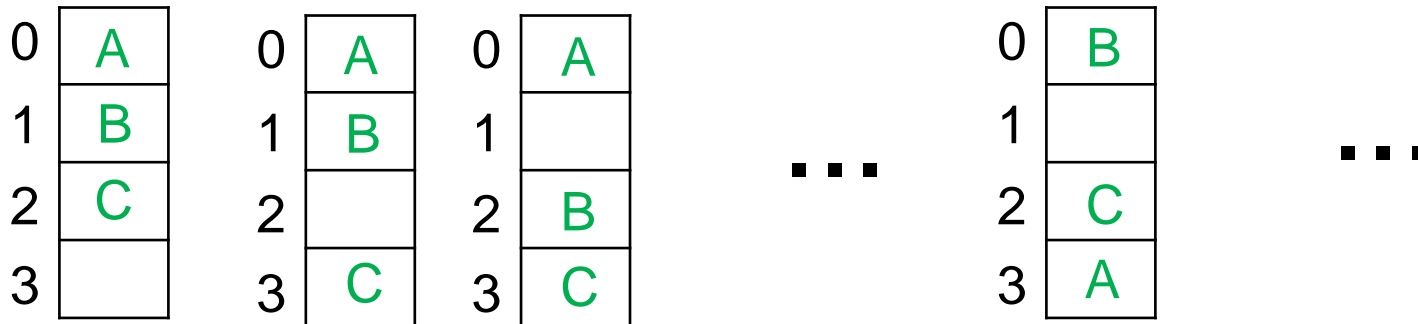


Viac na diskretnej matematike v 2. ročníku



# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky A, B, C



Každý prvok patrí do inej priehradky  
Najlepší prípad, lebo nemáme kolízie

Dokopy 24 možností

Viac na diskkrétnej matematike v 2. ročníku



# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky **A**, **B**, **C**
- Dokopy  $4 + 36 + 24 = 4^3 = 64$  možností





# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky **A**, **B**, **C**
- Aká je zložitosť hľadanie v najhoršom prípade?

0	ABC
1	
2	
3	

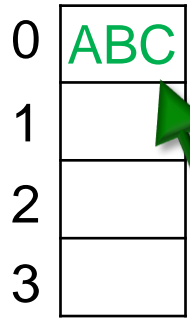
Prvky sú uložené v spájanom zozname

- Hľadáme jeden z prvkov **A**, **B**, **C**
  - Pri hľadaní **A** prechádzam cez 1 prvok (priamo A)
  - Pri hľadaní **B** prechádzam cez 2 prvky (A a B)
  - Pri hľadaní **C** prechádzam cez 3 prvky (A, B a C)
  - V prieme prechádzam cez 2 prvky



# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky **A**, **B**, **C**
- Aká je zložitosť hľadanie v najhoršom prípade?



- Hľadáme prvok **D**

- Ak je hash prvku **D** rovnaký ako **A**, **B**, **C**, tak prechádzame pokiaľ neprídeme na koniec spájaného zoznamu **A**, **B**, **C**, **null** pozreli sme sa 4 krát
- Ak je hash rôzny od hashu prvkov **A**, **B**, **C**, tak vieme odpovedať po pohľade do jednej priehradky, kde je zapísané **null**
- Priemer  $(4+1+1+1)/4 = 1,75$

Prvky sú uložené v spájanom zozname





# Zložitosť hľadania

- Videli sme zložitosť hľadania ak sa hľadaný prvok nachádza (2) a nenachádza v poli (1,75)
- Ďalej uvažujeme len s prípadom, že hľadaný prvok sa nachádza v poli

0	ABC
1	
2	
3	

4 možnosti  
priemerný  
čas hľadania  
2

0	AB
1	
2	C
3	

36 možnosti  
priemerný  
čas hľadania  
 $\frac{4}{3}$

0	A
1	B
2	C
3	

24 možnosti  
priemerný  
čas hľadania  
1



# Zložitosť hľadania

- Príklad pre  $LF=0,75$  a uložené prvky **A**, **B**, **C**
- Priemerná zložitosť hľadania:

$$\bullet \frac{\text{Čas všetkých hľadání}}{\text{Počet možností}} = \frac{4 \cdot 2 + 36 \cdot \frac{4}{3} + 24 \cdot 1}{4 + 36 + 24} = \frac{80}{64} = 1,25$$

- Vo všeobecnosti počítame pre nejaký Load Factor a počet uložených hodnôt  $n$

$$\lim_{n \rightarrow \infty} \frac{\text{Všetkých hľadání}}{\text{Počet možností}}$$

- Limita je rovná konštante, teda priemerná zložitosť hľadania je konštantná  $O(1)$



# Zložitosť pridávania

- Podobne vieme ukázať zložitosť pridávania a odoberania prvkov z HashSet-u, navyše využijeme podobné myšlienky ako pri poli s kapacitou
- Agregovaná zložitosť pridávania a odoberania prvkov je  $O(1)$



# HashMap

- Namiesto prvku a nasledovníka si pamätajme aj ďalšie informácie



Dáta z ktorých počítame hash a index sa nazýva kľúč (key)

Informácie uložené ku kľúču sa nazývajú hodnota (value)

Kľúče sa nemôžu zhodovať!



# *HashSet a HashMap realita*

- Na implementáciu HashSet-u sa bežne používa HashMap, kde sú všetky hodnoty (value) prázdne
- Čo o tom vraví dokumentácia?



# HashSet sumár

- Čo sme spomenuli:
  - Štruktúru HashSet-u
  - Ako hash-ovat' (rôzne prístupy)
  - Problémy pri hash-ovani
  - Priemernú zložitost' hľadania
  - Agregovanú zložitost' pridávania a odoberania



# Hľadanie v textoch

- Obsahuje nasledujúci text slovo „null“?
- *Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*



# Hľadanie v textoch

- Jednoduché hľadanie v texte

text: 

A	A	B	A	B	C	B	C	A	B
---	---	---	---	---	---	---	---	---	---

hľadaný  
reťazec: 

A	B	C
---	---	---

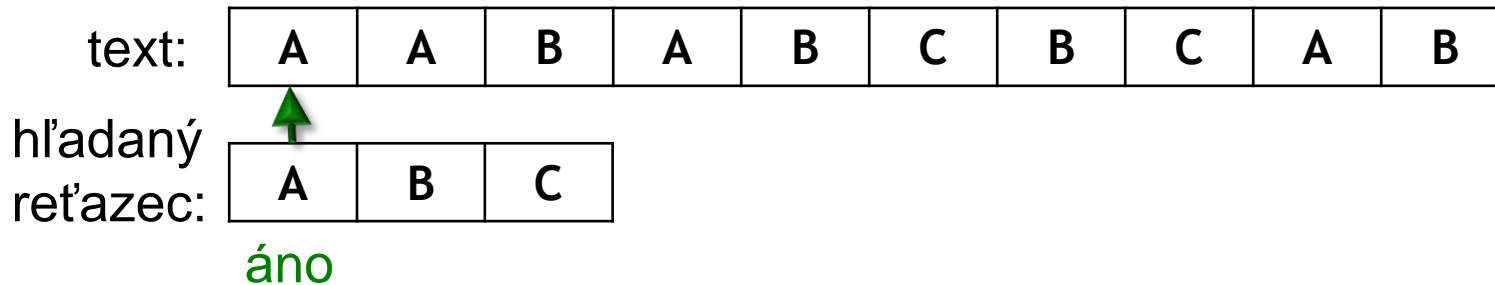
- Postupne „prikladám“ hľadaný reťazec ku textu a zist'ujem, či nastala zhoda





# Hľadanie v textoch

- Jednoduché hľadanie v texte

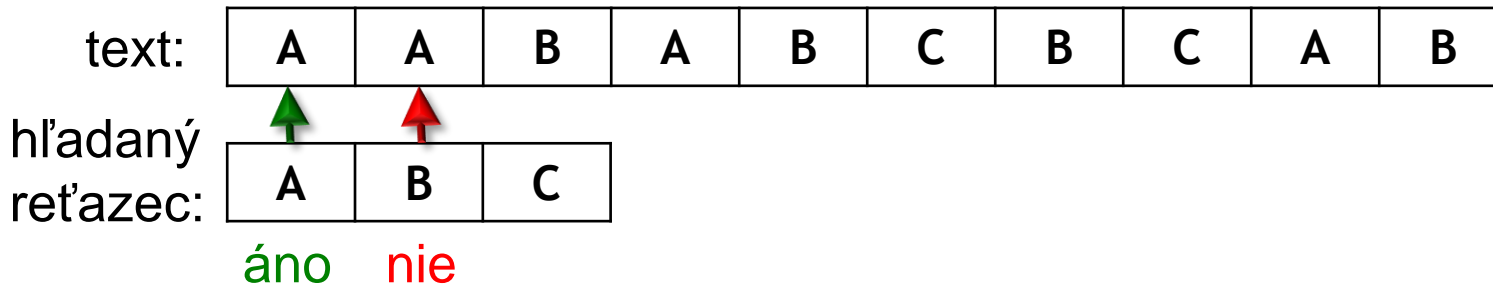


- Postupne „prikladám“ hľadaný reťazec ku textu a zist'ujem, či nastala zhoda



# Hľadanie v textoch

- Jednoduché hľadanie v texte

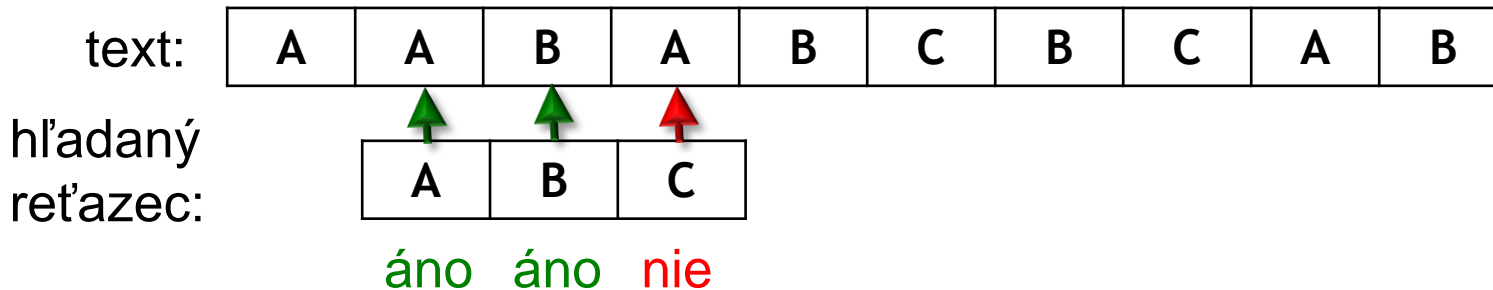


- Postupne „prikladám“ hľadaný reťazec ku textu a zistujem, či nastala zhoda
- Ak zhoda nenastala, tak posuniem hľadaný reťazec o jeden znak ďalej a skúšam znova



# Hľadanie v textoch

- Jednoduché hľadanie v texte

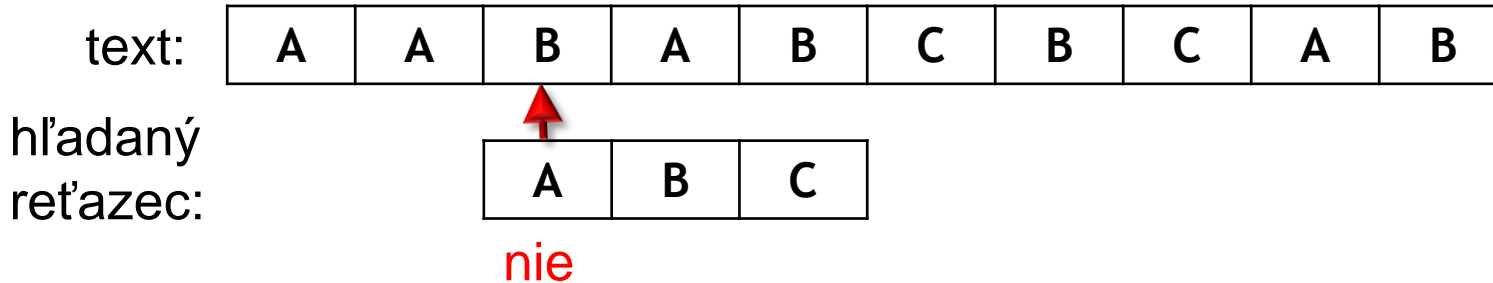


- Postupne „prikladám“ hľadaný reťazec ku textu a zist'ujem, či nastala zhoda
- Ak zhoda nenastala, tak posuniem hľadaný reťazec o jeden znak d'alej a skúšam znova



# Hľadanie v textoch

- Jednoduché hľadanie v texte

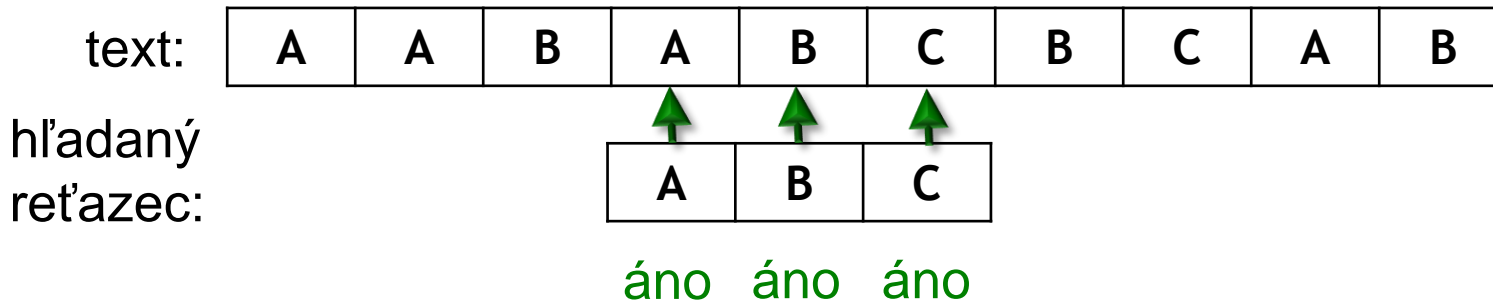


- Postupne „prikladám“ hľadaný reťazec ku textu a zist'ujem, či nastala zhoda
- Ak zhoda nenastala, tak posuniem hľadaný reťazec o jeden znak d'alej a skúšam znova



# Hľadanie v textoch

- Jednoduché hľadanie v texte

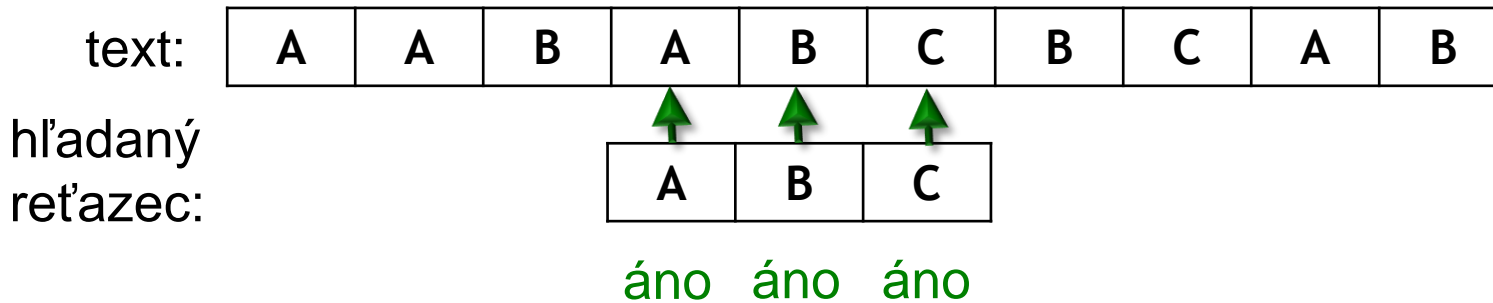


- Postupne „prikladám“ hľadaný reťazec ku textu a zist'ujem, či nastala zhoda
- Ak zhoda nenastala, tak posuniem hľadaný reťazec o jeden znak d'alej a skúšam znova
- Ak nastala zhoda vo všetkých znakoch našli sme hľadaný reťazec, vrátíme pozíciu jeho začiatku



# Hľadanie v textoch

- Jednoduché hľadanie v texte

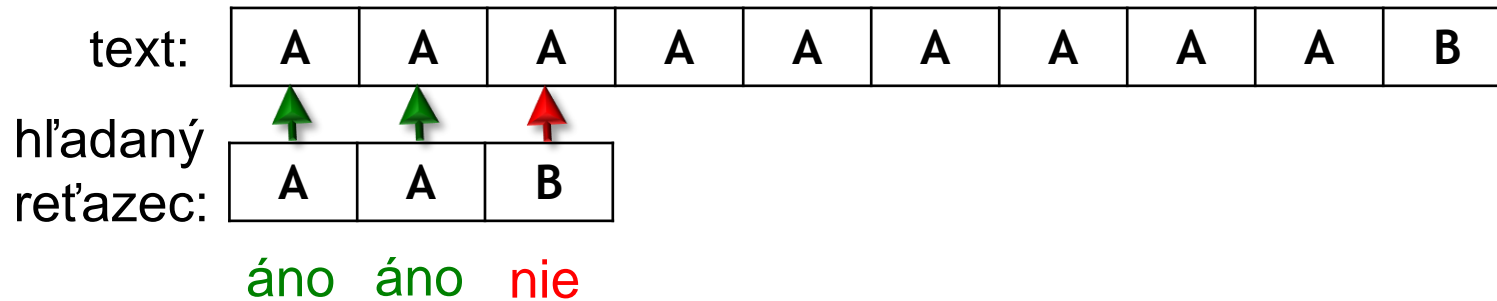


- Aká je zložitost' tohto algoritmu? Uvažujme, že dĺžka textu je  $n$  a dĺžka hľadaného reťazca je  $m$
- $O(n.m)$



# Hľadanie v textoch

- Jednoduché hľadanie v texte

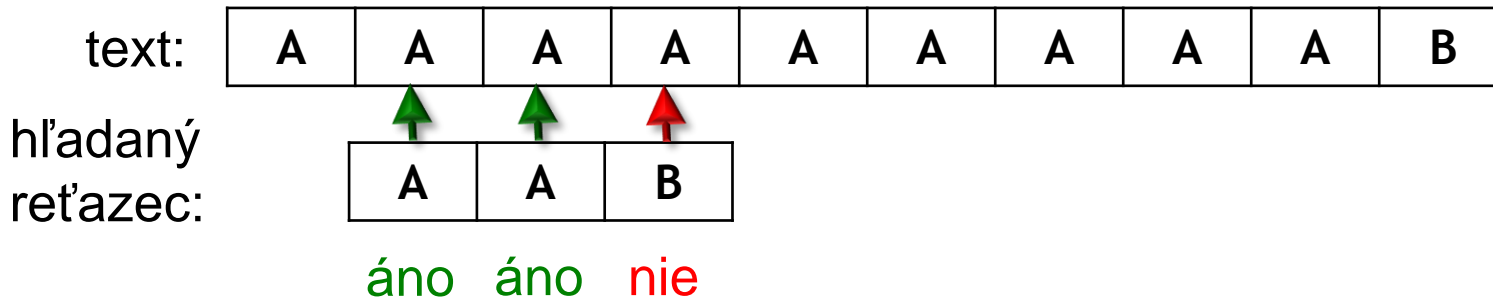


- Najhorší prípad
- O nerovnosti viem rozhodnúť až pri poslednom porovnávaní



# Hľadanie v textoch

- Jednoduché hľadanie v texte



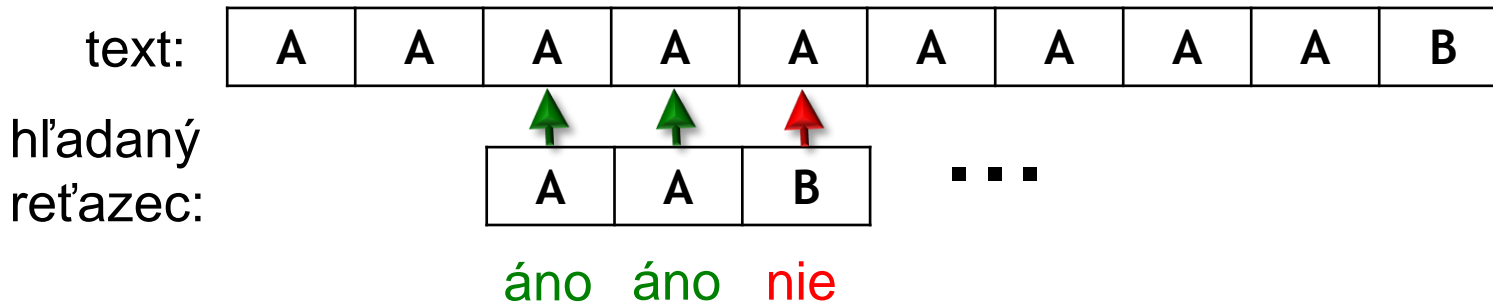
- Najhorší prípad
- O nerovnosti viem rozhodnúť až pri poslednom porovnávaní





# Hľadanie v textoch

- Jednoduché hľadanie v texte

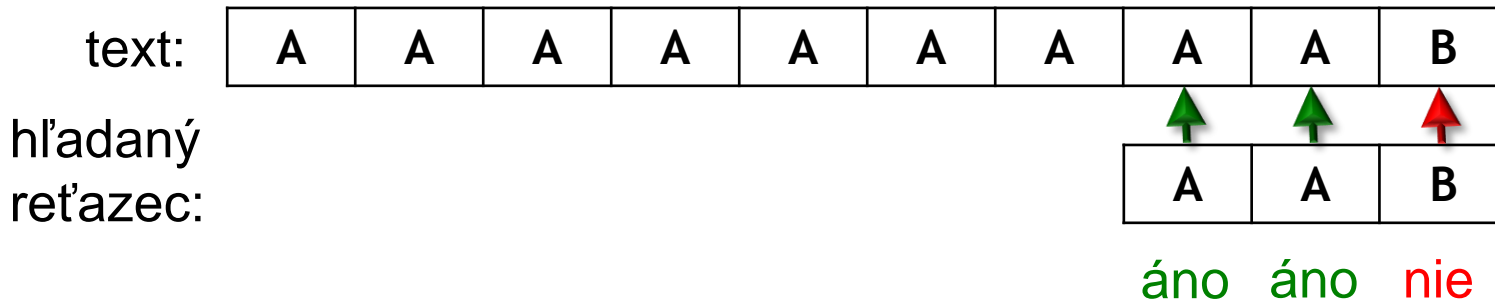


- Najhorší prípad
- O nerovnosti viem rozhodnúť až pri poslednom porovnávaní



# Hľadanie v textoch

- Jednoduché hľadanie v texte



- Najhorší prípad
- O nerovnosti viem rozhodnúť až pri poslednom porovnávaní



# Využitie hash-u

- Majme text zložený iba z malých a veľkých znakov anglickej abecedy
- Ich unicode hodnoty sú medzi 65 - 'A' až 122 - 'z'
- Hash reťazca získame ako polynóm

$$\left( \sum_i a^i x_i \right) \text{mod } n$$

- Kde  $x_i$  je hodnota znaku na pozícii  $i$
- Za  $a$  zvolíme číslo 123 (väčšie ako hodnota najväčšieho znaku)
- z reťazca „AAB“ dostaneme hodnotu (znaky od konca)  
 $123^0 * 66 + 123^1 * 65 + 123^2 * 65 = 991446$



# Využitie hash-u

- AAB zodpovedá 991446
- Ako ho využiť pri hľadaní v texte?

991445

text: 

A	A	A	C	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---

hľadaný

reťazec:

A	A	B
---	---	---

991446

- AAA zodpovedá  $1*65 + 123*65 + 123^2*65 = 991445$
- Namiesto postupného porovnávania znakov porovnáваме hash-e



- AAB zodpovedá 991446
- Ako ho využiť pri hľadani v texte?

???

text: 

A	A	A	C	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---

hľadaný  
reťazec:

A	A	B
---	---	---

991446

- AAA zodpovedá  $1*65 + 123*65 + 123^2*65 = 991445$
- Ako z hash-u pre AAA získať hash pre AAC?



- Ako z hashu pre AAA získať hash pre AAC?

- $991445 - 123^2 * 65 = 8060$

Z hashu pre AAA  
odstránim prvý znak  
získam hash pre \_AA

- Posuniem znaky a jednoducho vypočítam hash posunu

- $8060 * 123 = 991380 = (65 + 123 * 65) * 123 = 65 * 123 + 123^2 * 65$

Posun z \_AA na  
AA \_ a počítam  
hash

- Pripočítame hodnotu pridávaného znaku

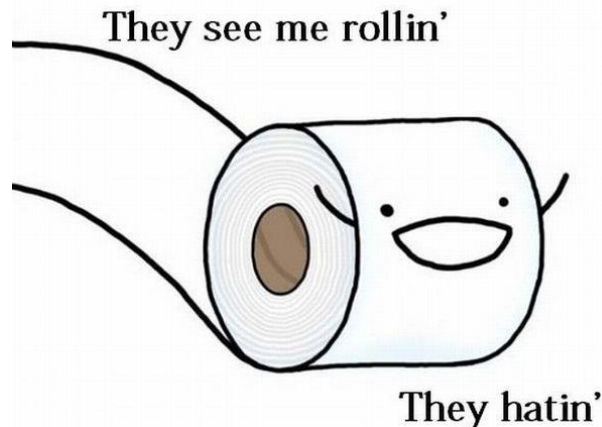
- $991380 + 67 = 991447$

z AA \_ dosnateme  
AAC



# Rolling Hash

- Cely hash nepočítame znova, ale počítame ho na základe starého hash-u a zmien
- Dostaneme tzv. Rolling Hash, prevaľuje sa cez objekt a podľa toho sa mení



Viac na cvičeniach



ak nie sú otázky...

# Ďakujem za pozornosť!

