acm International Collegiate Programming Contest

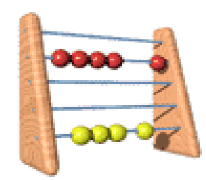
STU FIIT

ACM ICPC je opäť tu!

Lokálne kolo programátorskej súťaže na STU v rámci

CTU Open Contest

27. - 28. 10. 2017



Pošli registračný e-mail na

acm.icpc@fiit.stuba.sk

do stredy 25. 10. 2017 do 18.00 hod.

Viac informácii na:

www.fiit.stuba.sk/acm

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Správa pamäte pri vykonávaní programu

26. 9. 2017

zimný semester 2017/2018

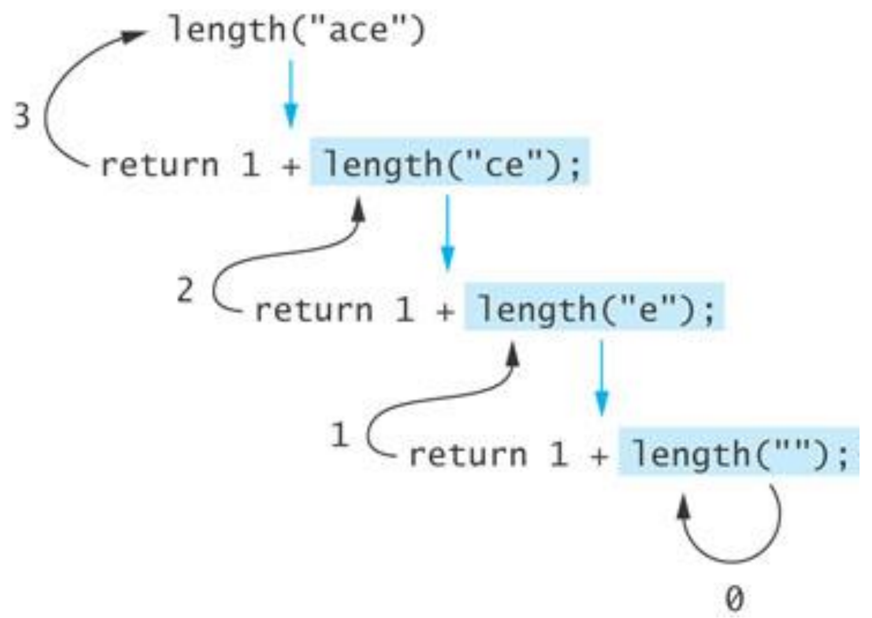
Príklad

- Rekurzívny algoritmus na určenie dĺžky reťazca
  - Ak je reťazec prázdny, výsledok je 0, inak
    výsledok je (1 + dĺžka reťazca bez prvého znaku)

  - Zdrojový kód:

```
int length(char *s)
{
  if (!s || *s == 0)
    return 0;
  return 1 + length(s+1);
}
```

Krokovanie **`length`**(“ace”):

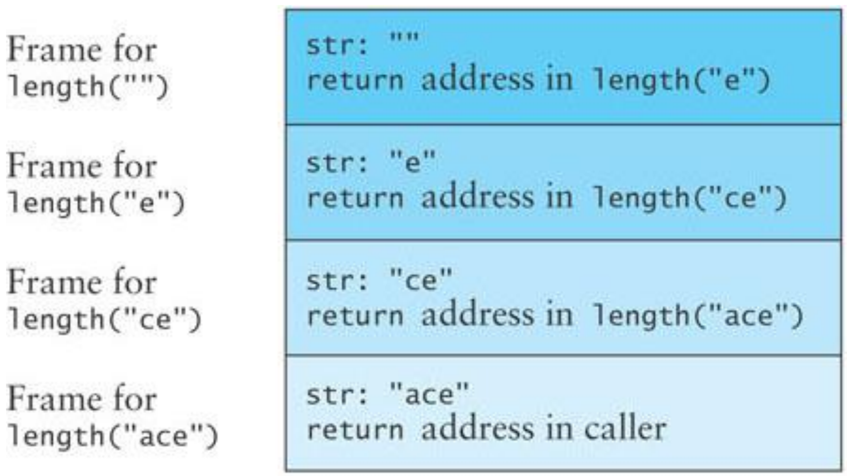


Aktivačný rámec

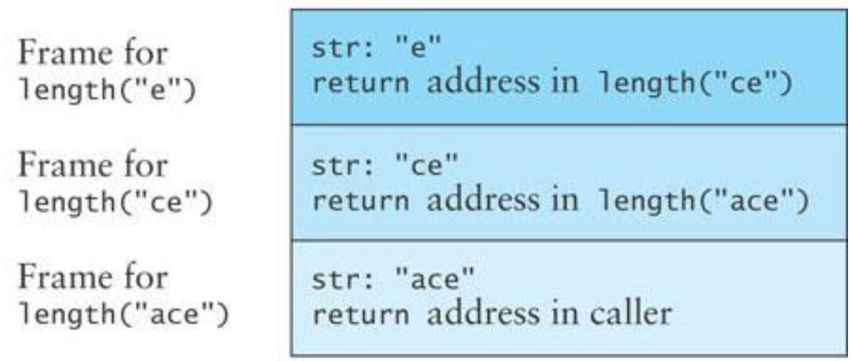
- Stavová informácia pre volanie funkcií
- Pre vykonanie volania funkcie je potrebné uchovať nasledovné informácie:
  - argumenty funkcie (hodnoty parametrov)
  - adresa, kam sa má vrátiť vykonávanie programu po ukončení volania funkcie (návratová adresa pre return)
  - lokálne premenné (hodnoty)
- Pre každé volanie funkcie sa vytvorí aktivačný rámec (stack frame) a vloží sa do zásobníka volaní (call stack)
- (Úmyselné) pretečenie zásobníka volaní predstavuje bezpečnostné riziko: **stack buffer overflow**

Zásobník volaní – ukážka

- Volanie length(“ace”):



obsah zásobníka po zavolaní length(“‘“), vrch je hore



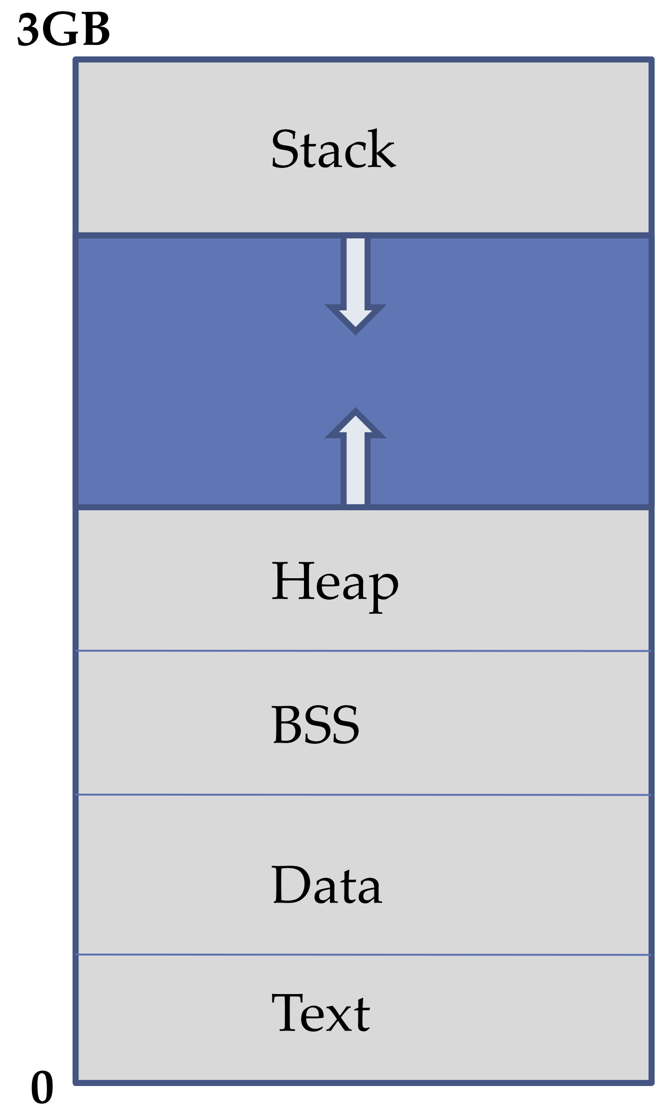
obsah zásobníka po návrate z vykonania length(“‘“)

Program vs. proces

- Program po preložení (**compile**), spojení s externými podprogramami (**link**) a načítaní do pamäti počítača (**load**) sa vykonáva (**execute**) – proces
- Riadiaci blok procesu (process control block)
  - Stav procesu – new, ready, running, waiting, …
  - Registre – %eip, %eax, …
  - Pamäť – všetko čo proces môže adresovať: kód, dáta, zásobník (stack), heap (halda)
  - I/O – stav otvorenia-čítania súborov
  - ...
- Program je statický kód a statické dáta
- Proces je dynamická inštancia kódu, dát a ďalšieho
- Bežiacemu procesu sa musí prideliť v počítači pamäť, aby mal kam zapisovať údaje (medzivýsledky atď.)

Adresný priestor procesu (Process address space)

- **Text**: obsahuje program v strojovom jazyku, ktorý sa vykonáva, reťazce, konštanty, a ďalšie údaje na čítanie
- **Data**: inicializované globálne a statické premenné
- **BSS**: (Block Started by Symbol) neinicializované globálne a statické premenné
- **Stack (zásobník)**: lokálne premenné bežiaceho procesu
- **Heap (halda voľnej pamäti)**:
  - dynamická pamäť procesu (môže sa zväčšovať aj zmenšovať)
  - toto je pamäť, ktorú prideľuje malloc()



Adresný priestor procesu – príklad

Data: globálna premenná

Text (read-only data)

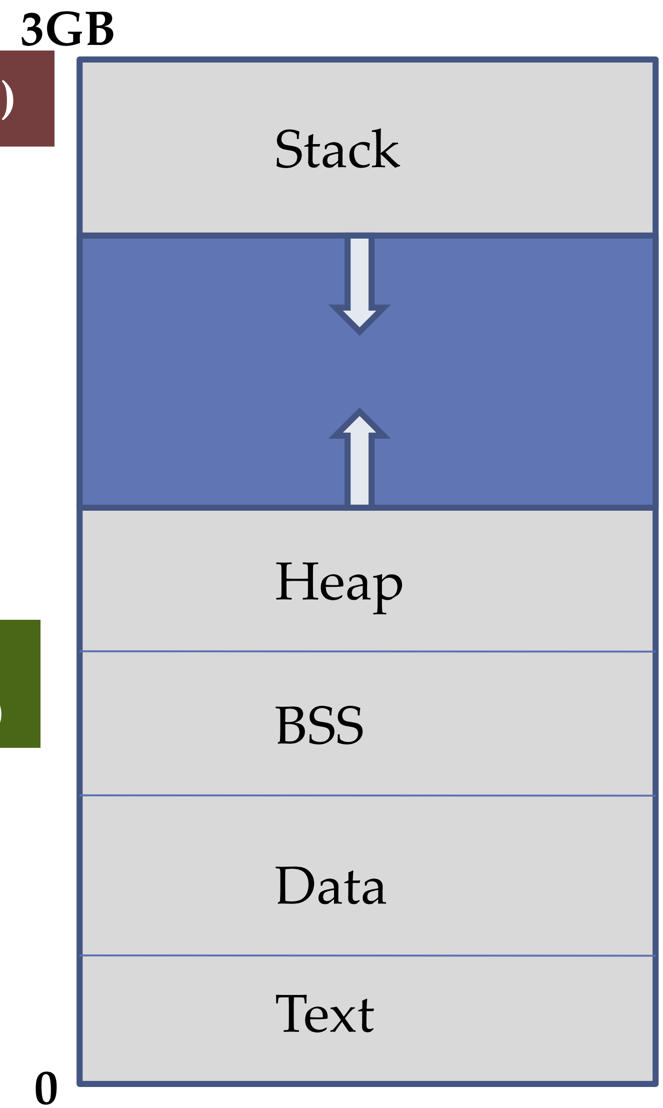
BSS

Stack (lokálna premenná)

Heap
(dynamická pamäť)

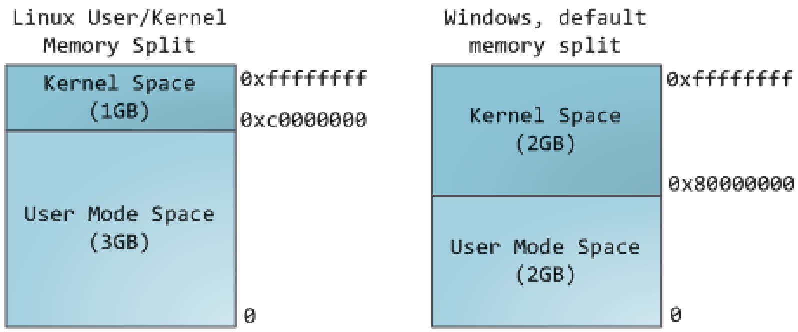
```
char str = “hello”;
int iSize;
char *f(void)
{
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

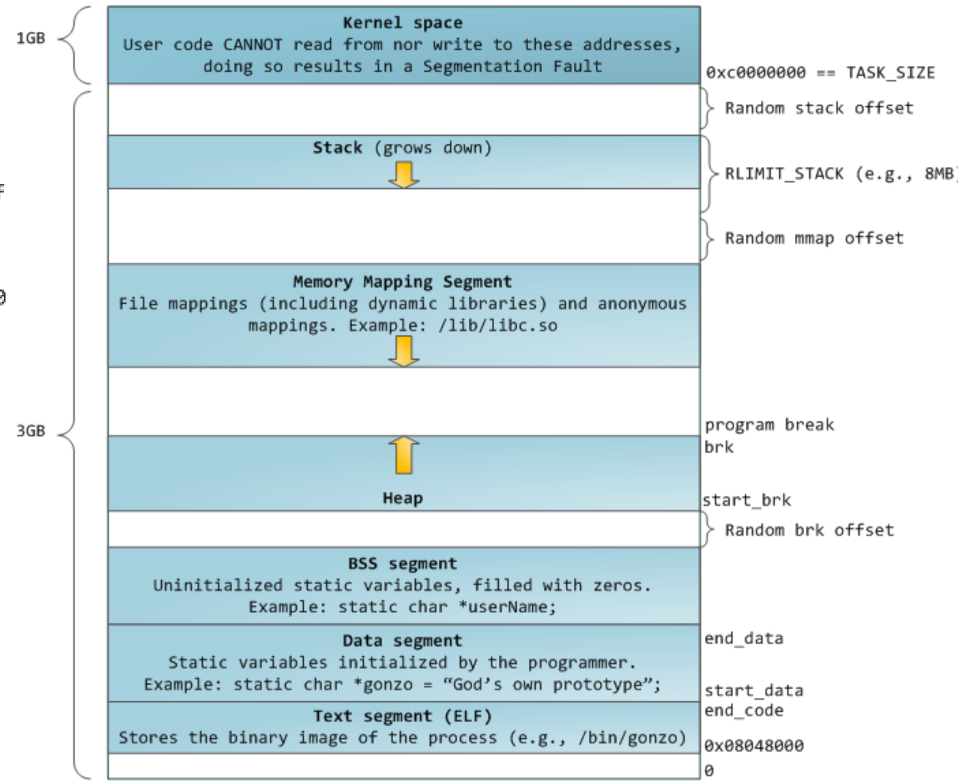
Preložený program je Text



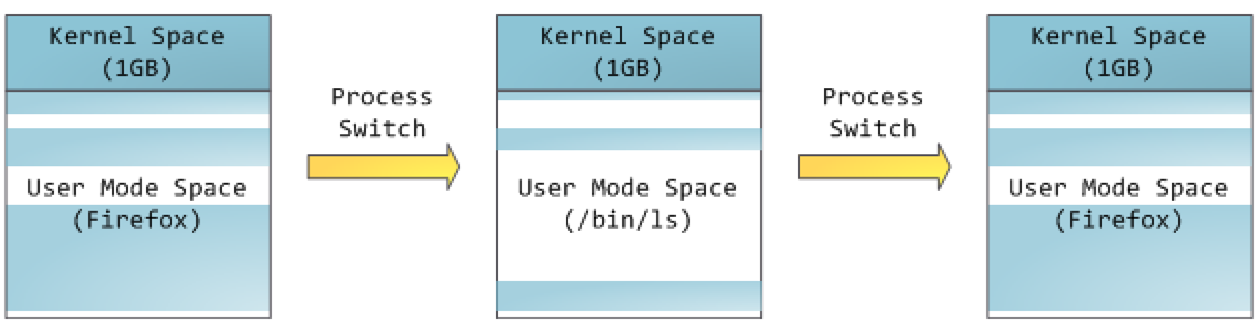
Detailnejší pohľad na pamäť v OS (32bit)

- Kernel vs. User mode





- Zmena vykonávaného procesu (context switch)



Typy prideľovania pamäte

- Statická veľkosť, statické prideľovanie
  - globálne premenné
  - spojovač (linker) pridelí definitívne virtuálne adresy
  - vykonateľný strojový program odkazuje na tieto pridelené adresy
- Statická veľkosť, dynamické prideľovanie
  - lokálne premenné
  - prekladač predpíše prideľovanie v zásobníku
  - posunutia voči ukazovateľu na vrch zásobníka (čo sú vlastne adresy premenných) sú priamo vo vykonateľnom strojovom programe
- Dynamická veľkosť, dynamické prideľovanie
  - ovláda programátor
  - prideľuje sa v dynamickej voľnej pamäti (heap – halda)

Prideľovanie dynamickej pamäti

- Dynamická pamäť sa prideľuje v čase výpočtu, nie v čase prekladu
- Veľkosť pridelenej pamäti nemusí byť známa až do okamihu pridelenia; napr. závisí od vstupného údaja zadaného používateľom
- Pretože veľkosť potrebnej pamäti môže byť rôzna, vyžiadanie jej pridelenia od procedúry **malloc** (apod) zahŕňa parameter veľkosť (**size**)

Funkcie pre prideľovanie pamäti v jazyku C

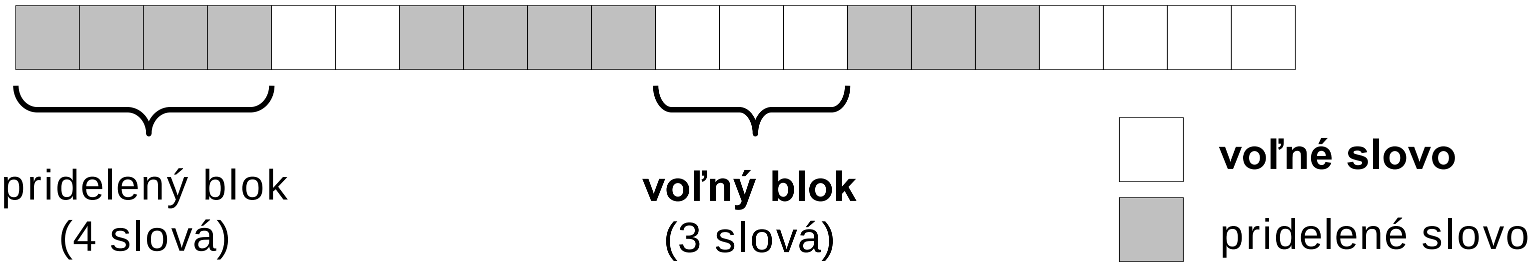
- Volanie **ptr ← malloc(size)** spôsobí, že sa pridelí pamäť veľkosťou čo najbližšia požadovanej
  - Pridelená pamäť nie je inicializovaná
- Volanie **free(ptr)** spôsobí, že pridelená pamäť (ptr) sa uvoľní – vráti späť do voľnej pamäti
- Veľkosť pridelenej pamäti možno zmeniť pomocou **newptr ← realloc(oldptr, size)**
- Volanie **ptr ← calloc(n, size)** spôsobí, že sa pridelí pamäť pre n prvkové pole s prvkami veľkosti size
  - Pridelená pamäť je inicializovaná na 0

Čo ak použitú pamäť nevrátime?

- Ak program nevráti (neuvoľní) pridelenú pamäť po tom, čo ju už netreba pre ďalší výpočet
  - stratí sa jediný odkaz na ňu
  - nebude sa dať jej obsah sprístupniť
  - je to trhlina v pamäti (**memory leak**)
- Ak sa v programe urobí odkaz na pamäť, ktorá bola medzitým uvoľnená, je to odkaz visiaci vo vzduchu (**dangling reference**)

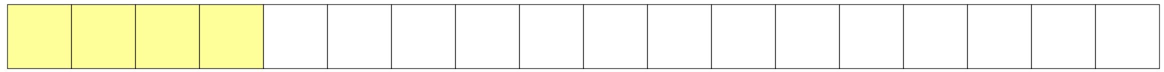
Ukážka prideľovania pamäte

- Pamäť sa adresuje po slovách
  - 4 byte (pre 32 bit architektúru)
- Na obrázkoch zobrazíme "štvorčeky" – slová
- Každé slovo môže obsahovať celé číslo (int) alebo smerník / ukazovateľ

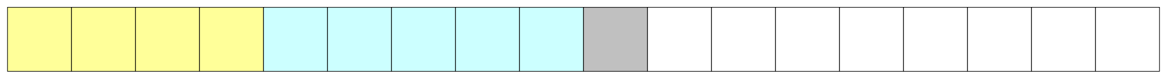


Ukážka prideľovania pamäte (2)

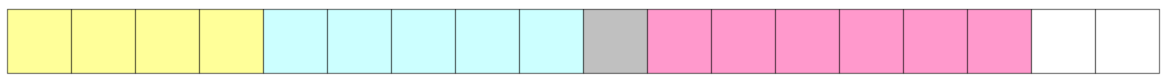
```
p1 = malloc(4*sizeof(int))
```



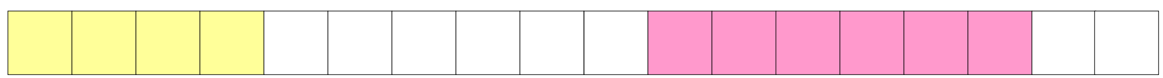
```
p2 = malloc(5*sizeof(int))
```



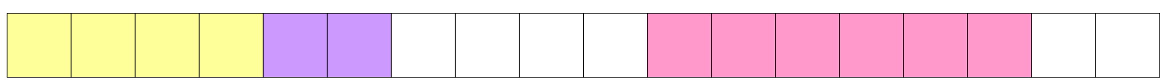
```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```



```
p4 = malloc(2*sizeof(int))
```



Ohraničenia prideľovania

- Programy, ktoré sa vykonávajú:
  - môže mať ľubovoľnú postupnosť požiadaviek malloc a free
  - požiadavky na free sa musia vzťahovať na pridelenú pamäť

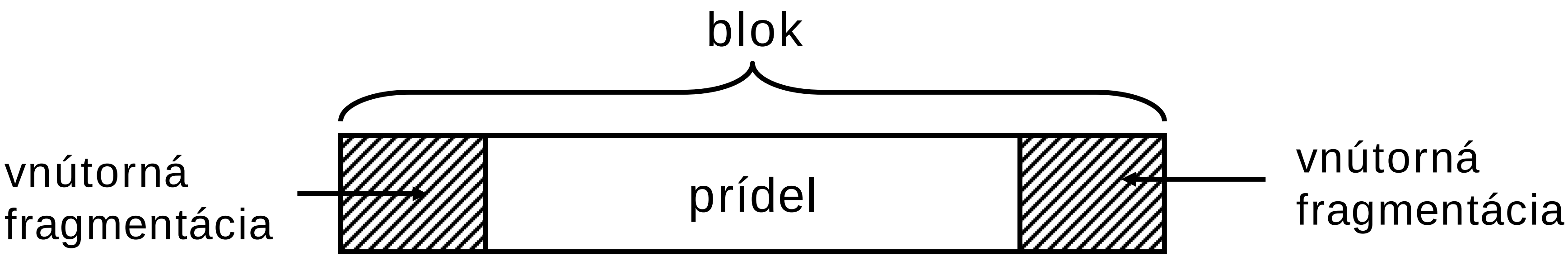
- Správca dynamickej pamäti
  - neovláda počet ani veľkosť prideľovaných blokov pamäti
  - musí vyhovovať všetkým požiadavkám okamžite
    (nemôže ich preusporiadať alebo odložiť na neskôr)
  - musí prideľovať pamäť z voľnej pamäti
  - musí zarovnať veľkosť bloku tak, aby splnila všetky požiadavky na zarovnávanie (zvyčajne na 8 byte-ov)
  - môže manipulovať a meniť iba voľnú pamäť
  - nemôže presúvať už pridelený blok pamäti
    (nebudeme predpokladať možnosť skompaktňovania)

Ciele dobrej implementácie predeľovania pamäte

- Dobrá časová efektívnosť **malloc** aj **free**
  - ideálne, v konštantnom čase (nie vždy možné)
  - určite by nemali potrebovať lineárny čas v závislosti od počtu blokov
- Dobré využívanie pamäti
  - pridelené bloky pamäti by mali využívať čo najväčšiu časť haldy
  - minimalizovať "fragmentáciu"
- Vlastnosti dobrej lokálnosti
  - štruktúry pridelené blízko v čase by mali byť blízko seba v pamäti
  - "podobné" objekty by mali byť umiestnené blízko seba
- Robustnosť
  - vie overiť, že **`free(p1)`** sa týka platného prideleného objektu **`p1`**
  - vie overiť, ukazovatele odkazujú do prideleného úseku pamäti

Vnútorná fragmentácia

- Pamäť nie je efektívne využitá celá – fragmentácia
  - vnútorná a vonkajšia
- **Vnútorná fragmentácia**
  - pre daný blok je vnútorná fragmentácia rozdiel medzi veľkosťou bloku a veľkosťou prídelu

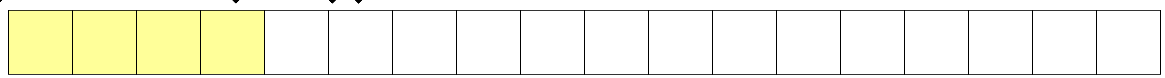


  - spôsobuje ju réžia (overhead) udržiavania dynamickej pamäti, zarovnávanie, prípadne rozhodnutia správy pamäti (napr. nerozbiť blok)
  - je určená tým, aké požiadavky boli doteraz, dá sa ľahko vyhodnotiť

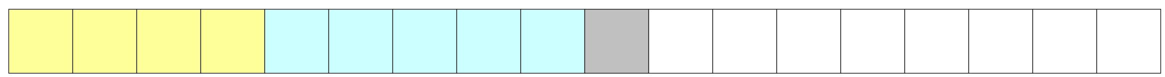
Vonkajšia fragmentácia

- nastáva, keď je síce dosť voľnej pamäti spolu (agregátne), ale žiadny voľný blok nie je dostatočne veľký

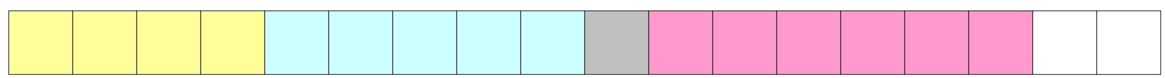
```
p1 = malloc(4*sizeof(int))
```



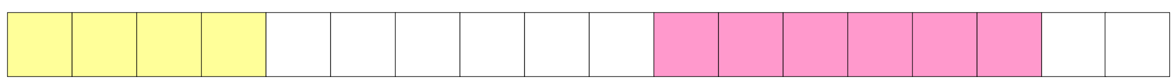
```
p2 = malloc(5*sizeof(int))
```



```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```



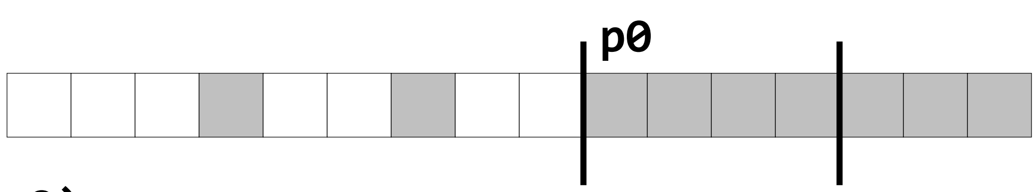
```
p4 = malloc(7*sizeof(int))
```

Hopla!

- vonkajšia fragmentácia závisí od toho, aké budú budúce požiadavky a preto sa nedá ľahko vyhodnotiť

Čo treba riešiť pri implementácii

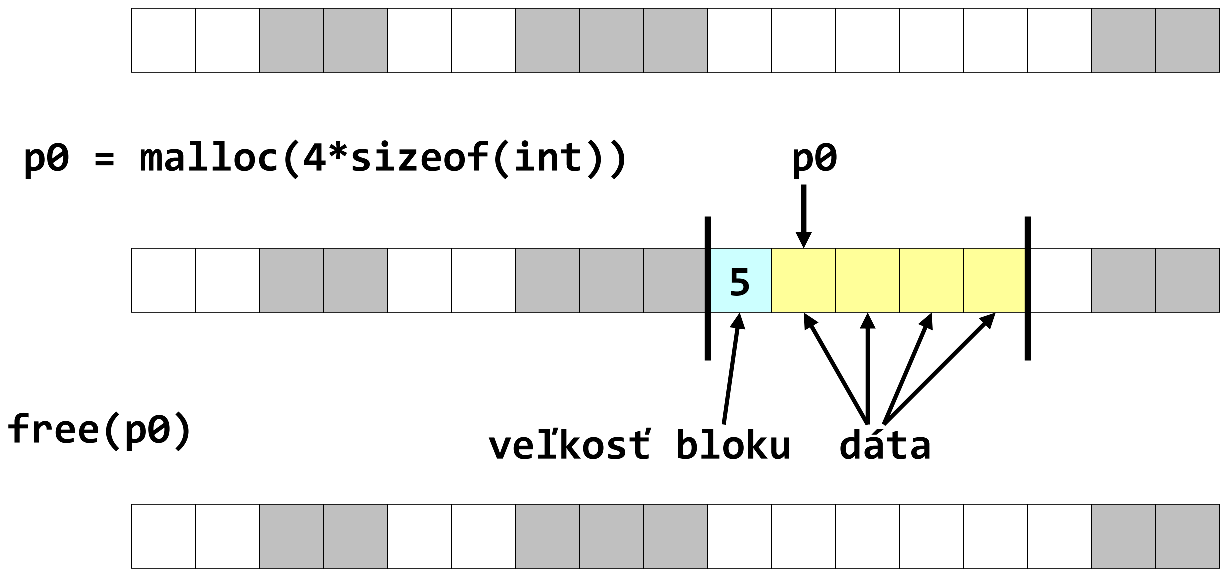
- Ako vieme, koľko pamäti sa má uvoľniť, keď **free** dostane len ukazovateľ?
- Ako si udržiavame záznam o tom, ktoré bloky sú voľné?
- Čo spravíme s nadbytočným kúskom pamäti keď prideľujeme pamäť štruktúre, ktorá je menšia než voľný blok, do ktorého ju umiestňujeme?
- Ako vyberieme blok, ktorý sa použije na pridelenie – môže ich byť viac vhodných?
- Ako vrátime uvoľnený blok do voľnej pamäti?




```
free(p0)
p1 = malloc(1)
```

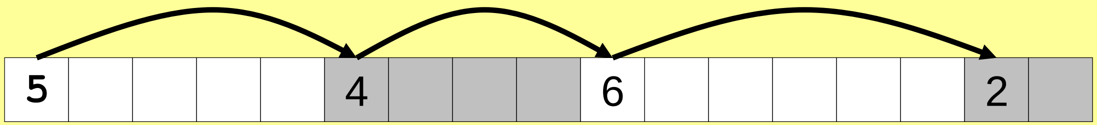
Čo (koľko) sa má vrátiť?

- zapísať dĺžku bloku do slova, predchádzajúceho bloku
  - toto slovo sa často nazýva hlavička
- vyžaduje jedno slovo navyše pre každý pridelený blok

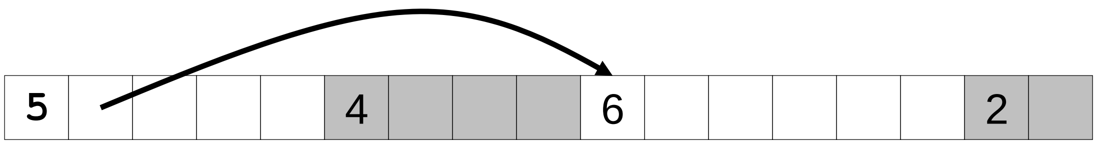


Udržiavanie voľnej pamäte

- **Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky**



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch

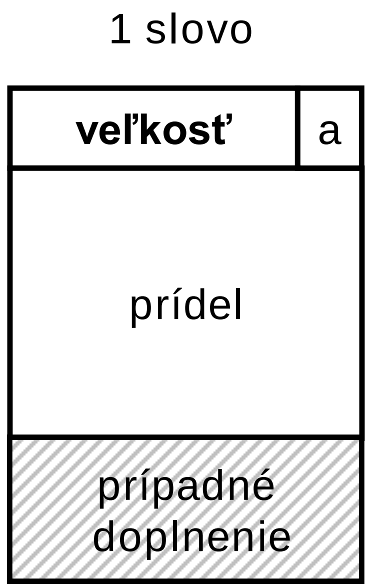


- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky
- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Implicitný zoznam blokov pamäti

- Treba rozpoznať (u každého bloku), či je voľný alebo pridelený
  - možno použiť 1 bit (navyše, niekde ho treba vziať)
  - bit možno vyhradiť v rovnakom slove, v ktorom je zapísaná veľkosť bloku ak sú veľkosti blokov vždy zarovnané aspoň na 2 (pri čítaní veľkosti sa maskuje najnižší bit)

Formát prideleného alebo **voľného** bloku



a = 1: pridelený blok
a = 0: **voľný blok**

veľkosť: **veľkosť bloku**

prídel: údaje vykonávaného programu (len v prípade prideleného bloku)

Nájdenie voľného bloku

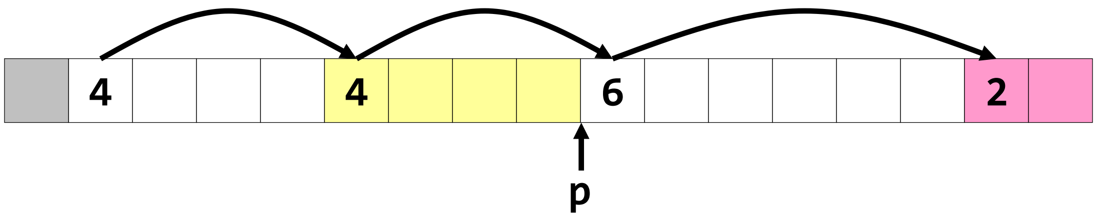
- **Prvý vhodný (first fit)**
  - prehľadáva sa zoznam od začiatku, vyberie sa prvý voľný blok, ktorý vyhovuje

```
p = start;
while ((p < end) &&       // nie sme na konci
       ((*p & 1) ||       // už pridelený
        (*p <= len)))     // príliš malý
  p = NEXT_BLKP(p);
```

  - môže vyžadovať čas lineárne úmerný celkovému počtu blokov
  - môže spôsobiť postupné vznikanie malých voľných blokov na začiatku zoznamu
- **Nasledujúci vhodný (next fit)**
  - ako metóda prvý vhodný, len sa prehľadávanie začne od miesta, kde skončilo predchádzajúce
  - skúsenosť hovorí, že fragmentácia je horšia
- **Najlepší vhodný (best fit)**
  - vyberie voľný blok s veľkosťou najbližšou k požadovanej (vyžaduje úplné prezretie celého zoznamu)
  - udržiava fragmenty malé
  - pomalší spôsob než prvý vhodný

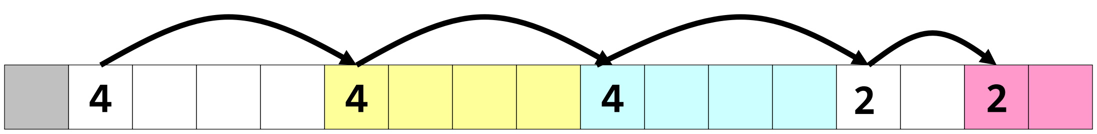
Pridelenie do voľného bloku

- Rozdelenie pôvodného voľného bloku
  - ak sa má prideliť menej pamäti než je veľkosť vybraného voľného bloku, môžeme ho rozdeliť




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;   // zvýš o 1 a zarovnaj hore (na 2)
  int oldsize = *p & ~0x1;               // zamaskuj najnižší bit
  *p = newsize | 0x1;                    // nastav novú dĺžku
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;    // nastav dĺžky v zostavájúcej
}                                        // časti bloku
```

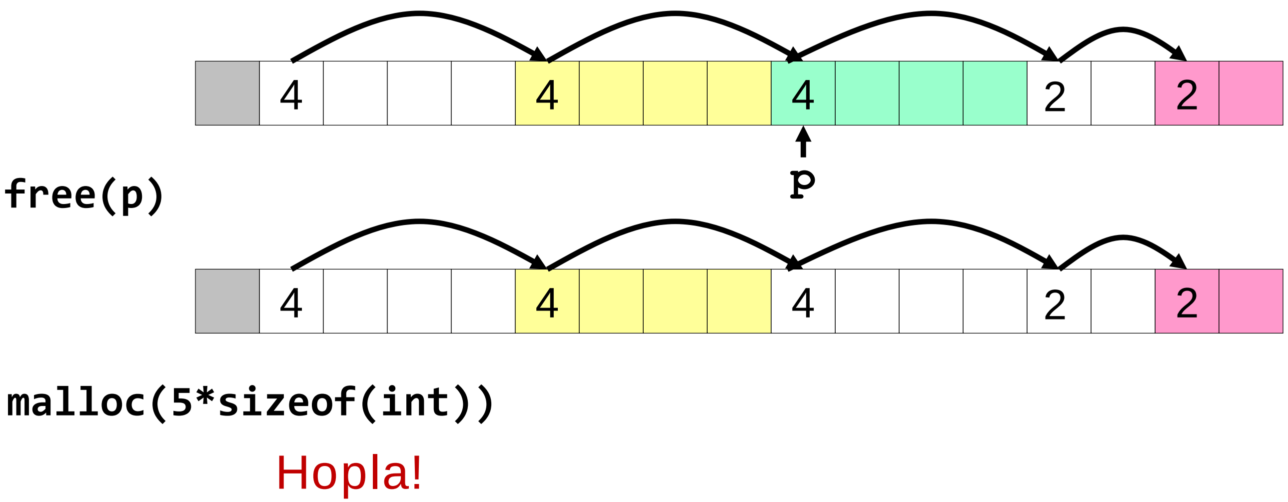
`addblock(p, 4)`



Uvoľnenie bloku

- Najjednoduchšia implementácia:
  - treba len nastaviť príznak voľnosti (najnižší bit na 0)

    ```
    void free_block(ptr p) { *p = *p & ~0x1}
    ```
  - môže však viesť ku "falošnej fragmentácii"

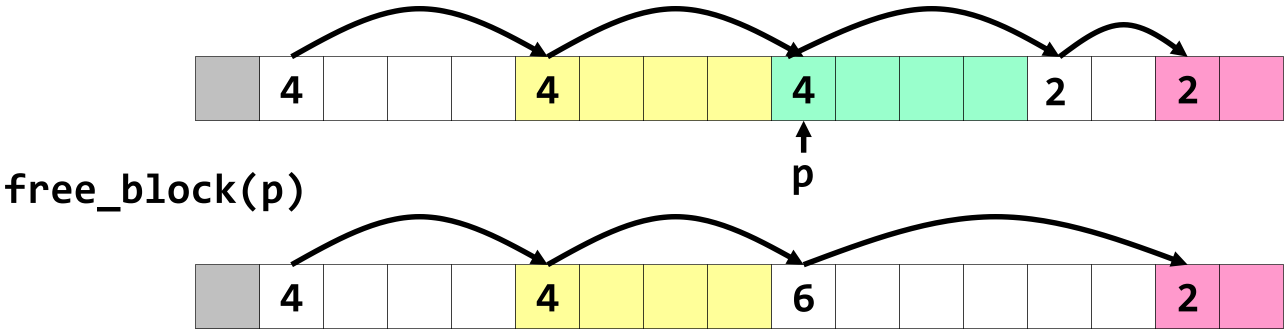


- **Síce je dosť voľnej pamäti na pridelenie bloku veľkosti 5, ale správca ju nevie nájsť!**

Spájanie

- Spojiť s nasledujúcim a/alebo predchádzajúcim blokom ak sú voľné
  - spojenie s nasledujúcim blokom

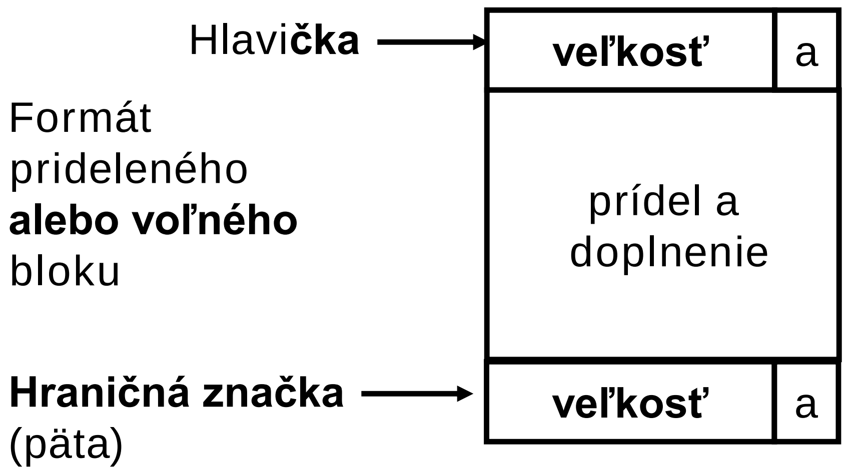
```
void free_block(ptr p) {
    *p = *p & ~0x1;          // vyčisti značku pridelenia
    next = p + *p;           // nájdi nasledujúci blok
    if ((*next & 0x1) == 0) // ak nie je pridelený
      *p = *p + *next;       // pridaj (dĺžku) k tomuto bloku
}
```



- Ale ako spojiť s predchádzajúcim blokom?

Obojsmerné spájanie

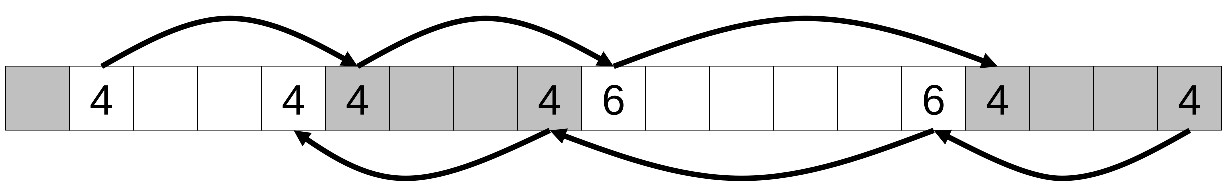
- Hraničné značky (boundary tags) [Knuth73]
  - skopírovať hlavičku aj na konci bloku
  - umožňuje prechádzať zoznam aj pospiatky, vyžaduje však pamäť navyše
  - dôležitá a všeobecná technika!



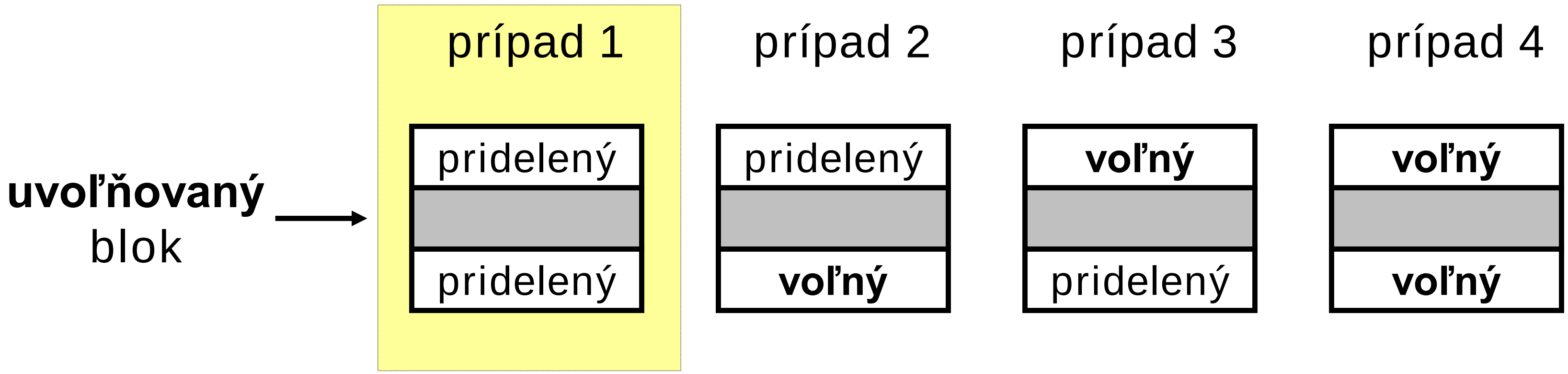
a = 1: pridelený blok
a = 0: **voľný blok**

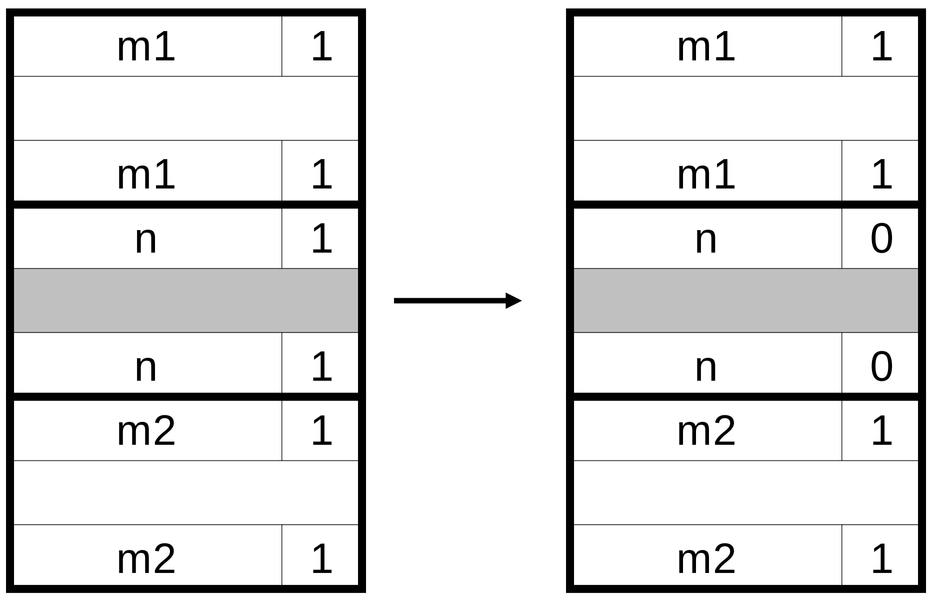
veľkosť: **celková veľkosť bloku**

prídel: údaje vykonávaného programu (len v prípade prideleného bloku)

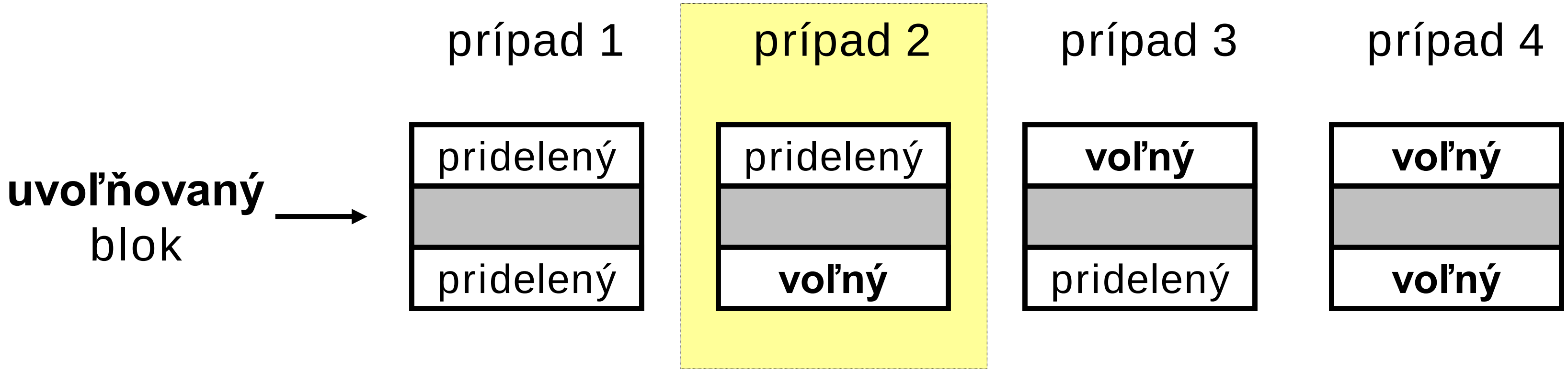


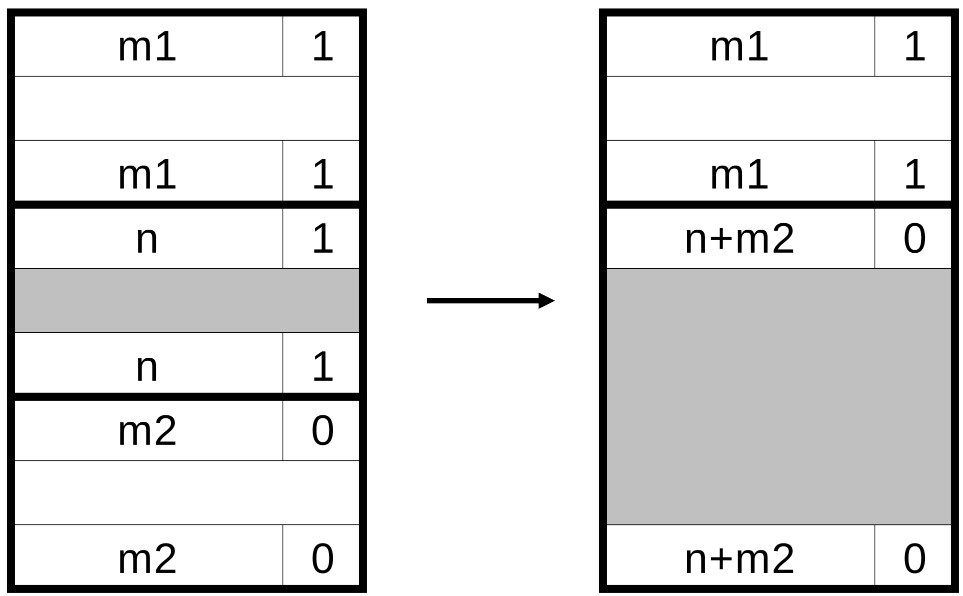
Spájanie v konštantnom čase



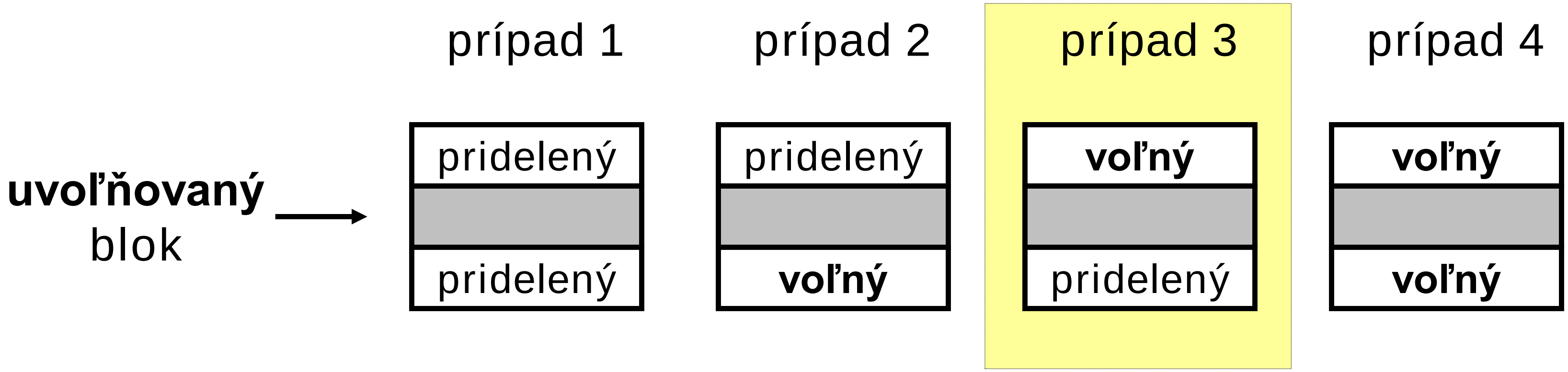


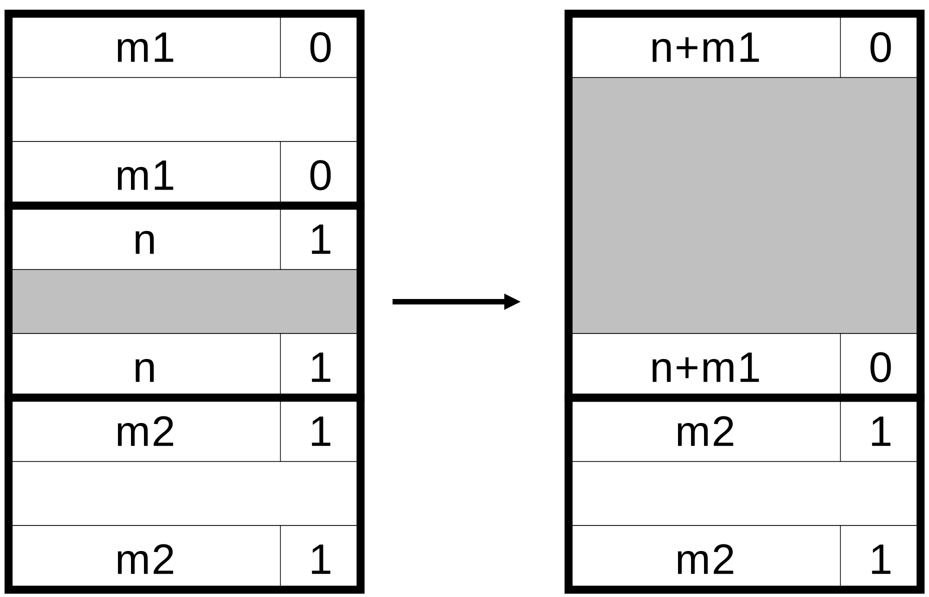
Spájanie v konštantnom čase



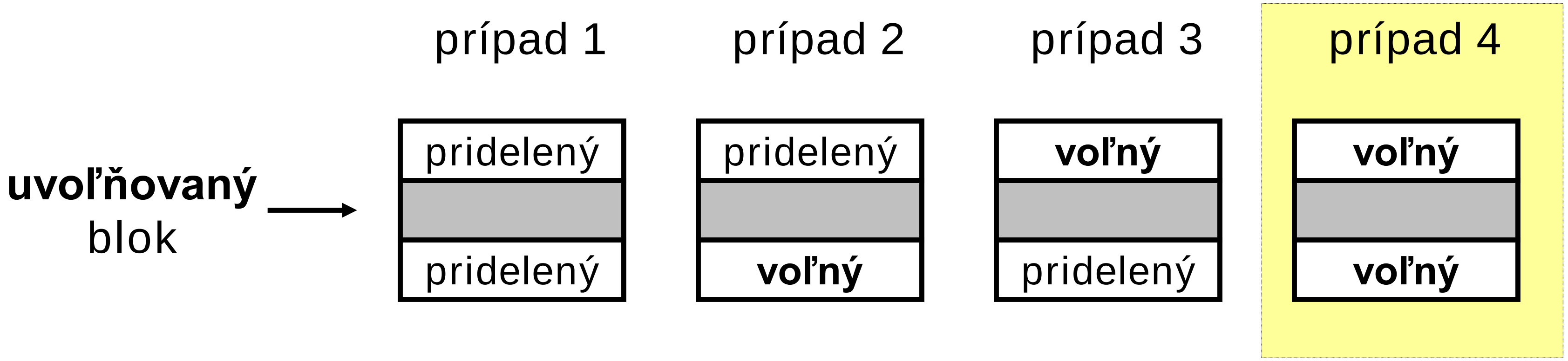


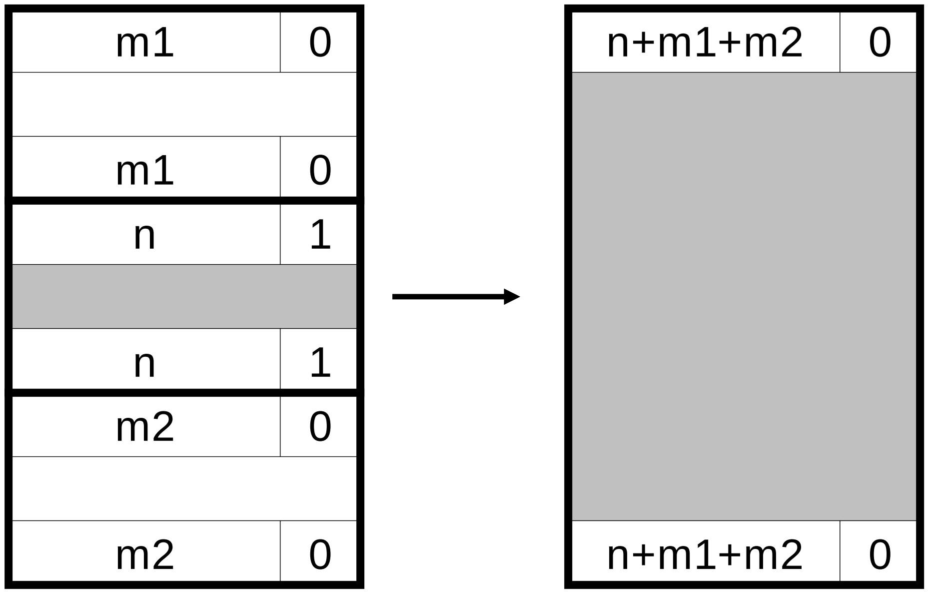
Spájanie v konštantnom čase





Spájanie v konštantnom čase





Rozhodovacie postupy správcu pamäti

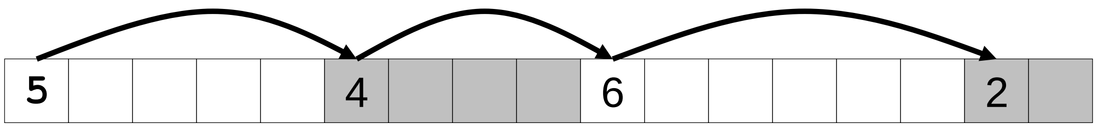
- Umiestnenie
  - Prvý vhodný, nasledujúci vhodný, najlepší vhodný, ...
  - nižšia priepustnosť za nižšiu fragmentáciu
- Rozdelenie
  - Kedy rozdeliť voľný blok?
  - Koľko vnútornej fragmentácie ešte pripustíme?
- Spájanie
  - Okamžité spájanie: spojiť susediace bloky vždy keď sa volá free
  - Odložené spájanie: skúsiť zrýchliť free odložením spájania dovtedy, kým to bude treba, napr.
    - spojiť až keď sa prezerá zoznam voľných blokov pre malloc
    - spojiť keď rozsah vonkajšej fragmentácie dosiahne nejaký určený prah

Implicitný zoznam blokov pamäti – zhrnutie

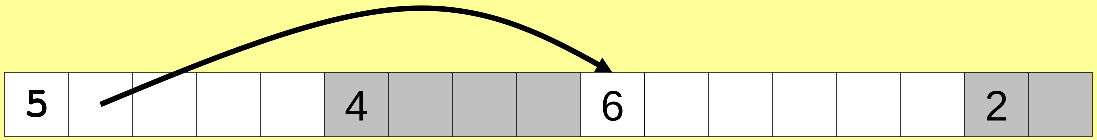
- Jednoduchá implementácia
- Pridelenie v lineárnom čase v najhoršom prípade
- Uvoľnenie v konštantnom čase v najhoršom prípade – dokonca aj so spájaním
- Využitie pamäti závisí od postupu prideľovania
  - Prvý vhodný
  - Nasledujúci vhodný
  - Najlepší vhodný
- V praxi sa nepoužíva pre **`malloc/free`** kvôli lineárnemu času pre prideľovanie
- Pojmy spájania a hraničnej značky sú všeobecné pre všetky metódy správy pamäti

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



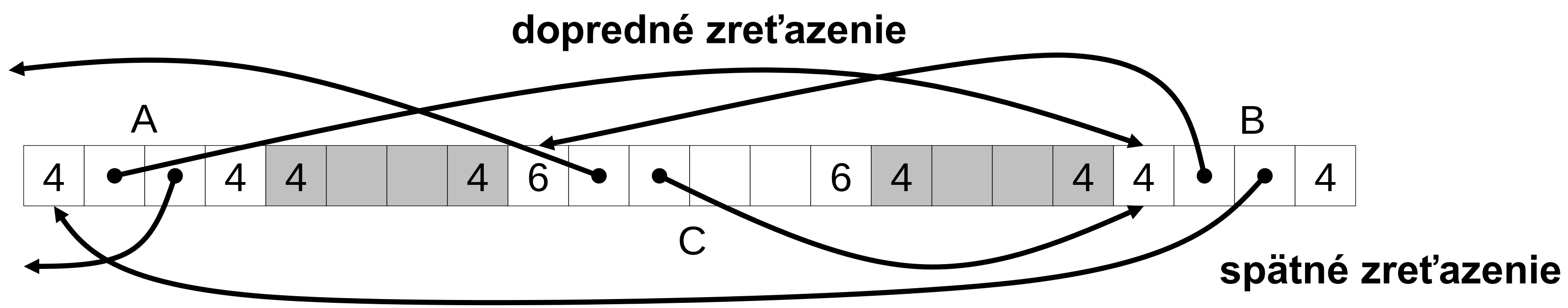
- **Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch**



- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky
- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Explicitný zoznam blokov voľnej pamäti

- používa sa pamäť pre údaje na ukazovatele
  - typicky sú obojsmerne zreťazené
  - aj tak treba hraničné značky na spájanie



  - poradie v zreťazení nemusí byť rovnaké ako poradie v pamäti

Uvoľnenie do explicitného zoznamu voľných blokov

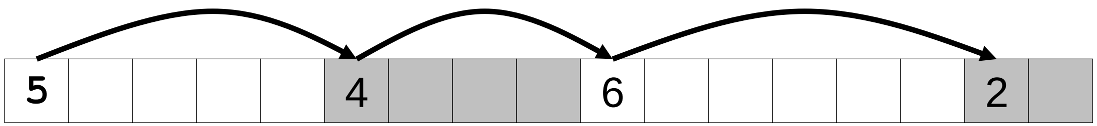
- Postup pre vloženie:
  Kam do zoznamu voľných blokov vložiť uvoľnený blok?
- Postup LIFO (last-in-first-out)
  - vložiť uvoľnený blok na začiatok zoznamu voľných blokov
  - za: jednoduchá implementácia, vykoná sa v konštantnom čase
  - proti: horšia fragmentácia ako pri postupe zachovávajúcom poradie v pamäti
- Postup zachovávajúci poradie v pamäti (usporiadanie podľa adries)
  - vkladať uvoľnené bloky tak, aby stále boli voľné bloky v zozname v takom poradí, v akom sú adresy, na ktorých sú zapísané v pamäti
    addr(predchádzajúci) < addr(aktuálny) < addr(nasledujúci)
  - proti: vyžaduje hľadanie
  - za: fragmentácia je lepšia ako pri LIFO

Explicitný zoznam blokov voľnej pamäti – zhrnutie

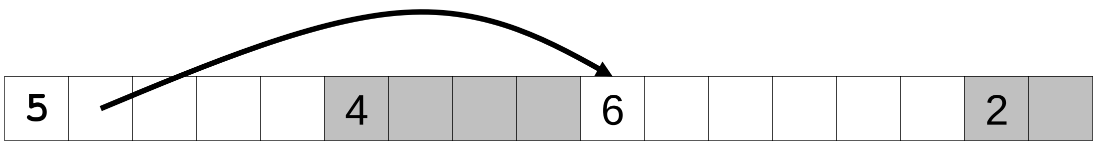
- Porovnanie s implicitným zoznamom:
  - pridelenie je v lineárnom čase závislé od počtu voľných blokov namiesto počtu všetkých blokov – je omnoho rýchlejšie keď je väčšina pamäti plná
  - trochu zložitejšie pridelenie aj uvoľnenie lebo treba zabezpečiť preskočenie bloku
  - o niečo viac pamäti treba na 2 ukazovatele (2 slová navyše treba pre každý blok)
- Hlavné použitie zreťazených zoznamov voľnej pamäti je v súvislosti s oddelenými zoznamami (Metóda 3)
  - udržiavať viacero reťazených zoznamov voľnej pamäti podľa veľkosti blokov alebo typu objektov

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch

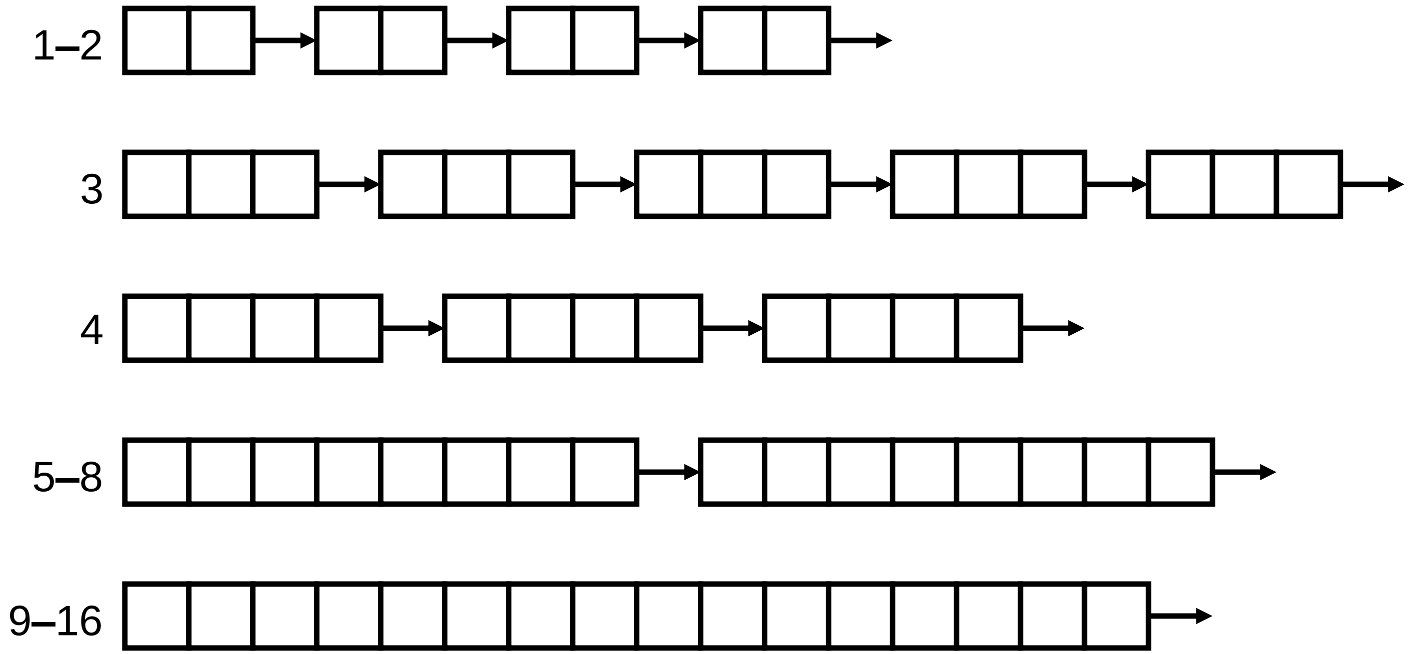


- **Metóda 3: oddelené zoznamy blokov voľnej pamäti**
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- Metóda 4: bloky usporiadané podľa veľkosti
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

Oddelená (segregovaná) pamäť

- Každá trieda veľkostí blokov má svoj zoznam



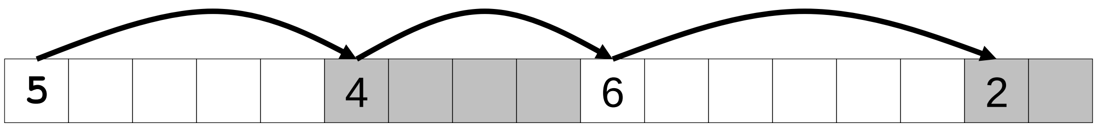
- Zvyčajne sú oddelené triedy pre každdú malú veľkosť (2,3,...)
- Väčšie veľkosti sa zoskupia podľa mocniny 2

Pridelenie a uvoľnenie v oddelenej pamäti

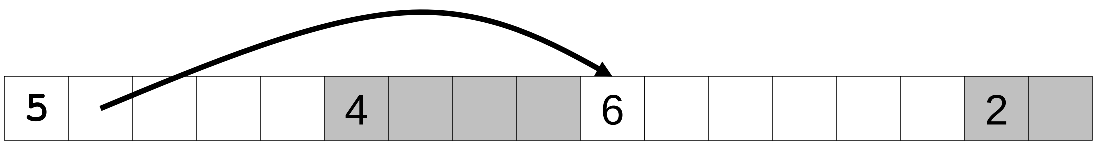
- Prideliť blok veľkosti n:
  - prehľadať vhodný zoznam voľných blokov hľadajúc blok veľkosti m >= n
  - ak sa nájde vhodný blok:
    - rozdeliť blok a umiestniť zvyšok do vhodného zoznamu (ak prichádza do úvahy)
  - ak sa nenájde vhodný blok v tomto zozname, skúsiť zoznam s triedou najbližších väčších blokov
  - opakuj, dokiaľ sa nájde blok
- Uvoľniť blok:
  - spojiť a umiestniť do vhodného zoznamu
- Vlastnosti
  - hľadanie je rýchlejšie než pri sekvenčnej organizácii (logaritmický čas pre triedy veľkostí podľa mocniny 2)
  - spájanie môže predĺžiť hľadanie
    - odloženie spájania to môže zlepšiť

Udržiavanie voľnej pamäte

- Metóda 1: implicitný zoznam s použitím dĺžok – spája všetky bloky



- Metóda 2: explicitný zoznam blokov voľnej pamäti pomocou ukazovateľov zapísaných priamo vo voľných blokoch



- Metóda 3: oddelené zoznamy blokov voľnej pamäti
  - rôzne zoznamy pre triedy blokov voľnej pamäti podľa dĺžky

- **Metóda 4: bloky usporiadané podľa veľkosti**
  - možno použiť vyvážený strom (napr. červeno-čierny) s ukazovateľmi zapísanými v každom voľnom bloku, dĺžka je kľúč

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Dátové štruktúry a algoritmy

Triedenie – Usporadúvanie

27. 9. 2016

zimný semester 2017/2018

Triedenie – Usporadúvanie

- Základná aplikácia počítačov
- Vstup:
  - Postupnosť: $a_1, a_2, a_3 \ldots a_n$
    $k(a_i)$ označíme kľúč k_i prvku a_i
  - Usporiadanie kľúčov < (binárna relácia)

    Lineárne usporiadaná množina K (total ordering)
    Pre $k_1, k_2 \in K$ budeme písať, že $k_1 \leq k_2$ akk $k_1 < k_2$ alebo $k_1 = k_2$.
- Výstup:
  - Permutácia π čísel 1, ..., n taká, že platí
    $k(a_{\pi(1)}) \leq k(a_{\pi(2)}) \leq \ldots \leq k(a_{\pi(n)})$

Triedenie – Usporadúvanie – príklad

- Vstup: **Postupnosť: a_1, a_2, ..., a_n**
  $k(a_i)$ označíme kľúč k_i prvku a_i
- Výstup: **Permutácia π čísel 1, ..., n taká, že platí**
  $k(a_{\pi(1)}) \leq k(a_{\pi(2)}) \leq \ldots \leq k(a_{\pi(n)})$

- Vstup:
  Peter, Jano, Milan, Miro, Filip
- Výstup?
- **π = (5, 2, 3, 4, 1)**
  Výsledné poradie kľúčov:
  Filip, Jano, Milan, Miro, Peter

Odhady zložitosti algoritmov – opakovanie

- Analýza najhoršieho prípadu
- Použitie O-notácie pre asymptotický horný odhad
- Klasifikujeme algoritmy podľa týchto zložitostí
- Nevýhoda tohto prístupu: **Nemôžeme použiť na predvídanie výkonu alebo porovnanie algoritmov!**
  - Quicksort – počet porovnaní v najhoršom prípade $O(N^2)$
  - Mergesort – počet porovnaní v najhoršom prípade O(N log N)
  - V praxi je však Quicksort zvyčajne dva krát rýchlejší a používa polovičné množstvo pamäti...

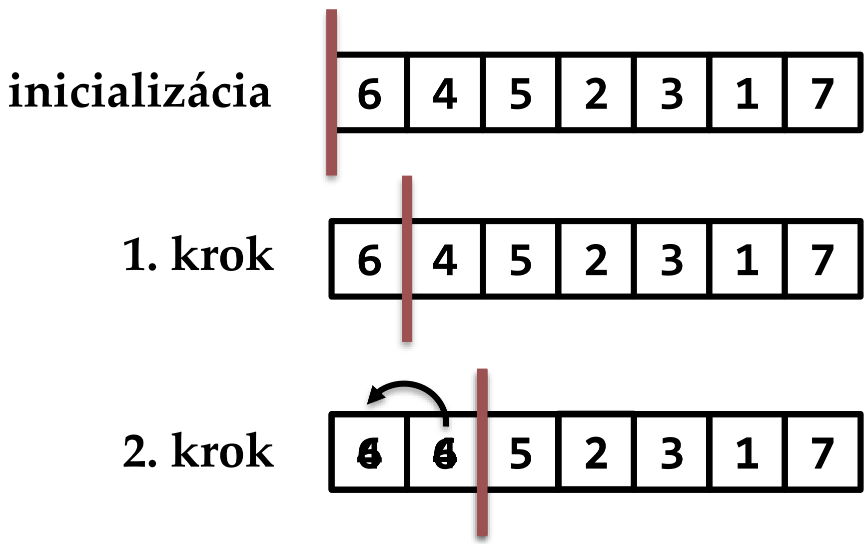
Triedenie priamym vkladaním (Insert sort)

- Insert sort spracúva vstupnú postupnosť postupne tak, že pojednom pridáva prvky na správne miesto do výslednej usporiadanej postupnosti (ktorá je najskôr prázdna a postupne sa rozširuje).

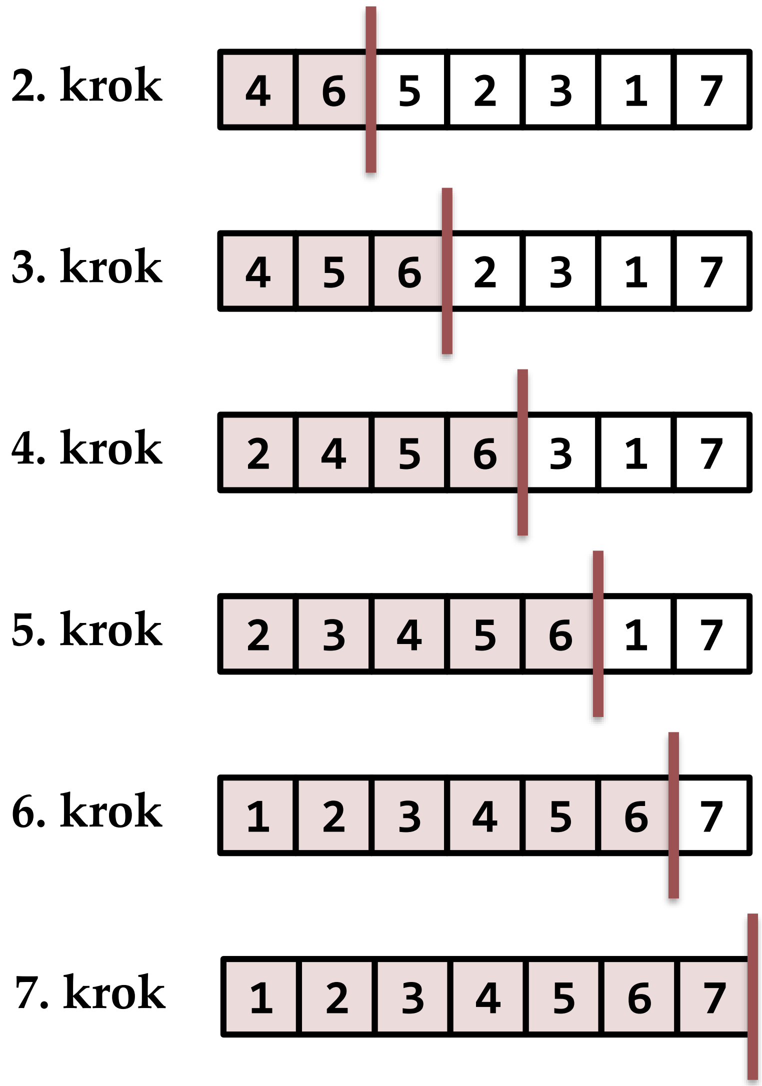
```
int* insert_sort(int *input, int n)
{
  int i, result[n];
  for (i = 0; i < n; i++)
    insert(input[i], result);
  return result;
}
```

Triedenie priamym vkladaním (Insert sort)

- Insert sort spracúva vstupnú postupnosť postupne tak, že pojednom pridáva prvky na správne miesto do výslednej usporiadanej postupnosti (ktorá je najskôr prázdna a postupne sa rozširuje).



Triedenie priamym vkladaním (Insert sort)



Triedenie priamym vkladaním (Insert sort)

```
int* insert_sort(int *input, int n)
{
  int i, result[n];
  for (i = 0; i < n; i++)
    insert(input[i], result);
  return result;
}
```

- Procedúra **insert**(prvok, pole) pomocou jednoduchého cyklu vloží prvok do poľa (v ktorom sú prvky v usporiadanom poradí) na správne miesto; vyžaduje rádovo ***L*** operácií, kde ***L*** je dĺžka poľa.

Analýza zložitosti (Insert sort)

- **Najlepší prípad: prvky sú už usporiadané**
  Procedúra insert vykoná O(1) presunov
  Celkovo – N krát insert O(1) = O(N) operácií
- **Najhorší prípad: prvky sú usporiadané opačne**
  Volania procedúry insert vykonajú koľko presunov?
  `0 + 1 + 2 + ... + N-1 = (N-1)*N/2`
  Celkovo $O(N^2)$ operácií
- **Priemerný prípad: prvky sú náhodne usporiadané**
  Volania procedúry insert vykonajú koľko presunov?
  Asi polovicu ako pri najhoršom prípade
  Celkovo $O(N^2)$ operácií

Triedenie zlučovaním (Merge sort)

- Merge sort vstupnú postupnosť rozdelí na dve polovice, každú rekurzívne utriedi, no a výslednú usporiadanú postupnosť všetkých prvkov určí zlúčením týchto menších usporiadaných postupností.

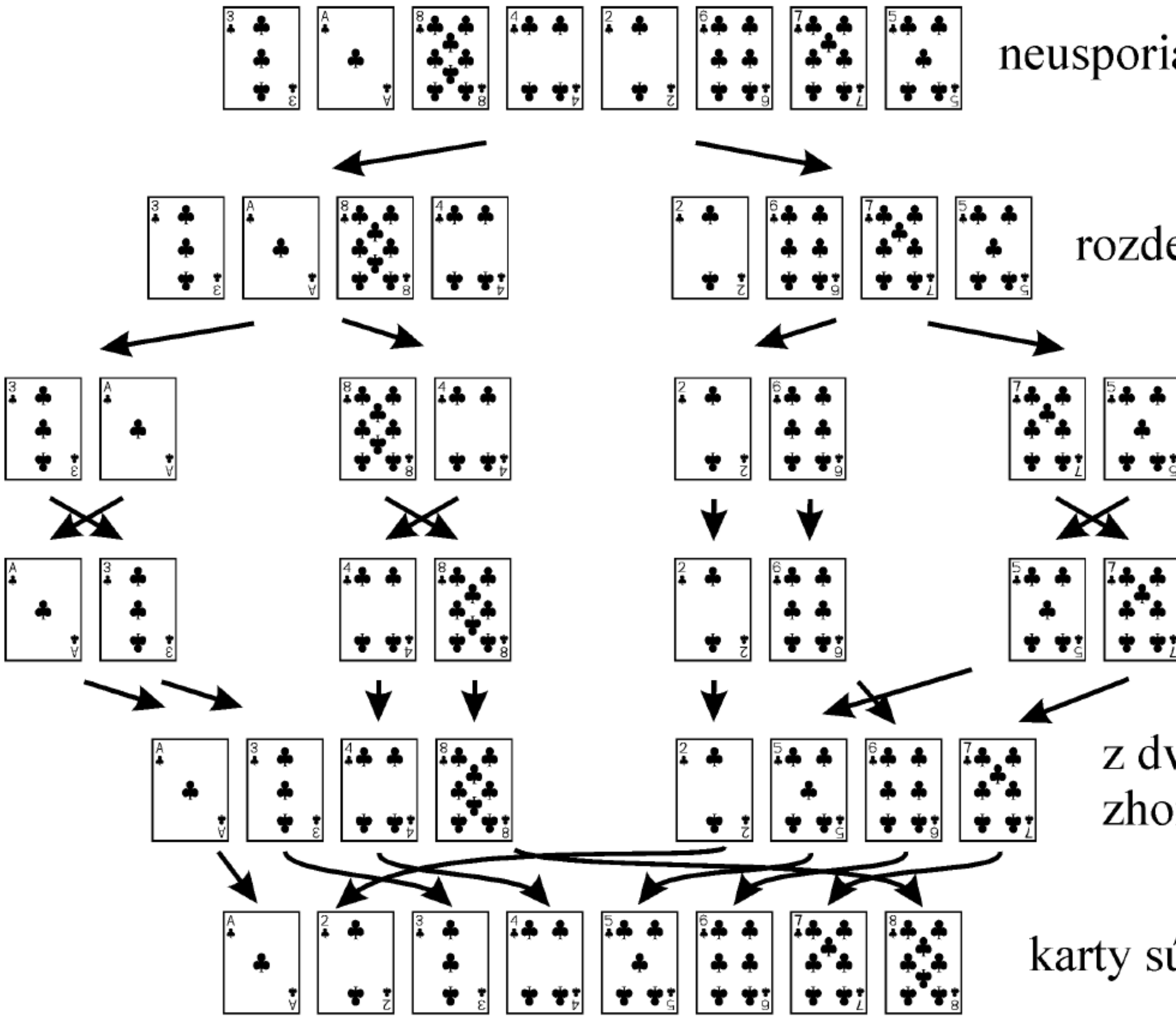
```
int* merge_sort(int *input, int left, int right)
{
  int mid = (left+right)/2;
  merge_sort(input,left,mid);
  merge_sort(input,mid+1,right);
  return merge(input,left,mid,right);
}
```

Triedenie zlučovaním (Merge sort)

```
int* merge_sort(int *input, int left, int right)
{
  int mid = (left+right)/2;
  merge_sort(input,left,mid);
  merge_sort(input,mid+1,right);
  return merge(input,left,mid,right);
}
```

- Procedúra **merge**(input, left, middle, right) pomocou jednoduchého cyklu spojí usporiadané postupnosti prvkov input[left, …, middle] a input[middle+1, …, right] do jednej usporiadanej postupnosti; vyžaduje rádovo right-left (dĺžka poľa vstupujúceho do operácie) operácií.

Ukážka triedenia zlučovaním hracích kariet



neusporiadané karty

rozdelíme na 2 kôpky

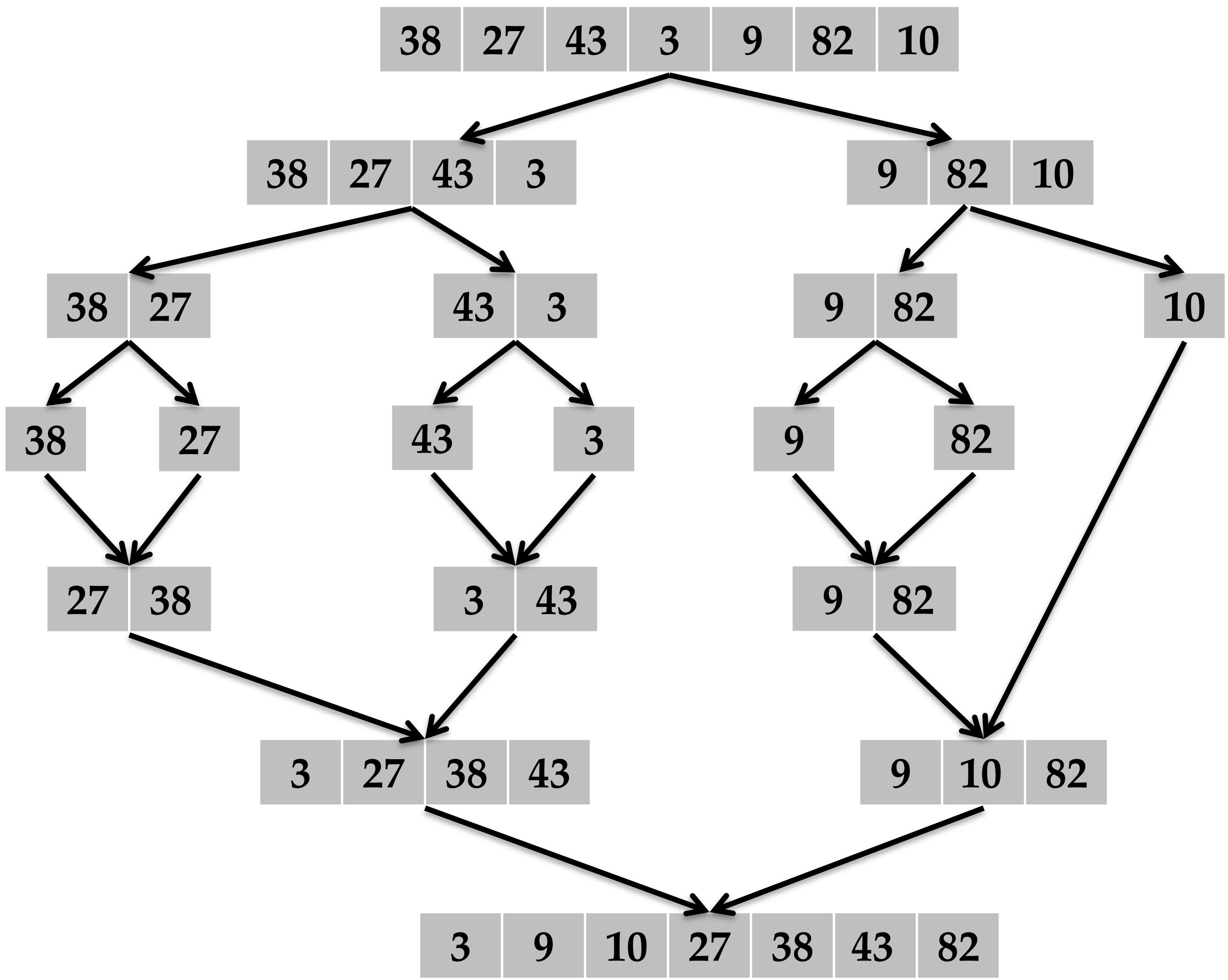
delíme až na 8 "kôpok" samostatných kariet

z dvojíc "kôpok" zoberieme zhora vždy menšiu kartu

z dvojíc "kôpok" zoberieme zhora vždy menšiu kartu

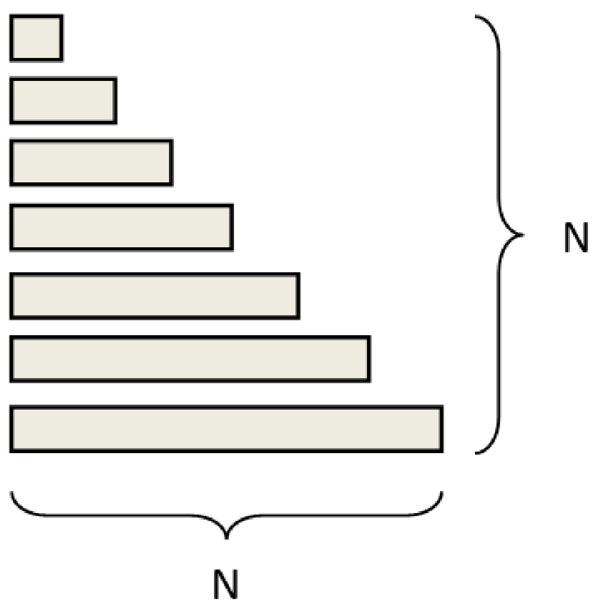
karty sú usporiadané

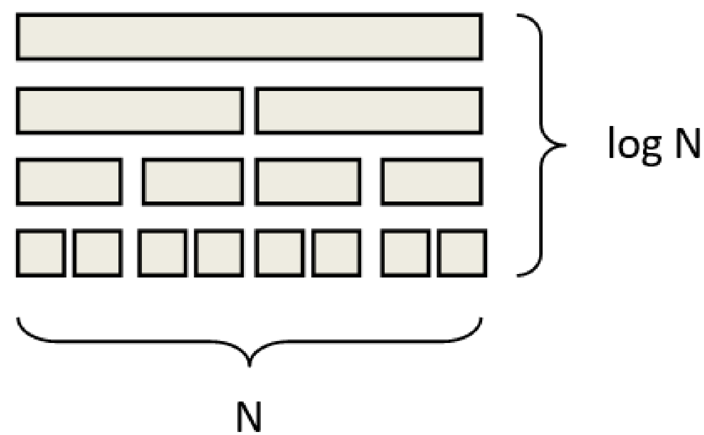
Priebeh rekurzívnych volaní (Merge sort)



Zložitosť Insert sort vs. Merge sort

- Obe operácie **insert** a **merge** vykonajú počet operácií lineárne závislý od veľkosti poľa na vstupe.
  Celkový počet operácií, ktoré algoritmy vykonajú je však odlišný!





Analýza zložitosti (Merge sort)

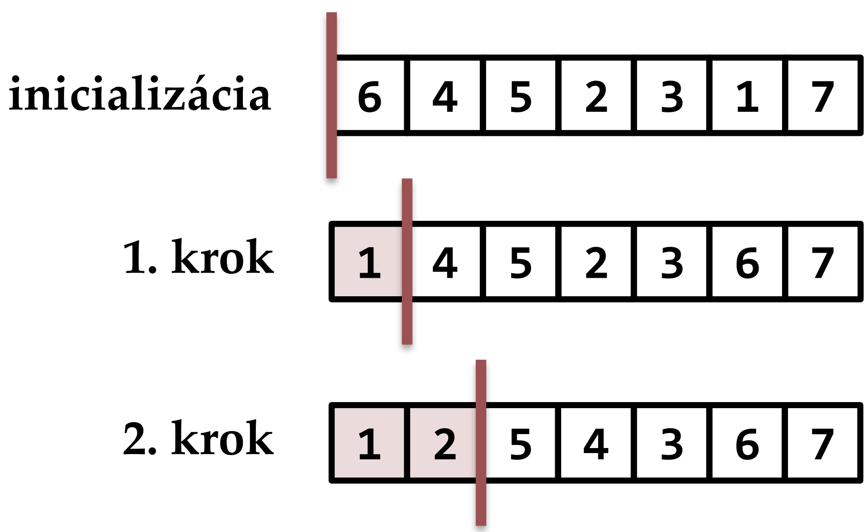
- Analýza výpočtovej zložitosti
  - Aký je **najlepší prípad**?
  - Aký môže byť **najhorší prípad**?
  - **V priemernom prípade** ide ako rýchlo?
  - Vždy vykoná rovnaké množstvo operácií bez ohľadu na vstupnú postupnosť
- Pamäťová náročnosť
  - Aký priestor je potrebný navyše okrem vstupnej postupnosti?
  - Potrebné pomocné pole pri zlučovaní

Usporadúvanie vo vonkajšej pamäti

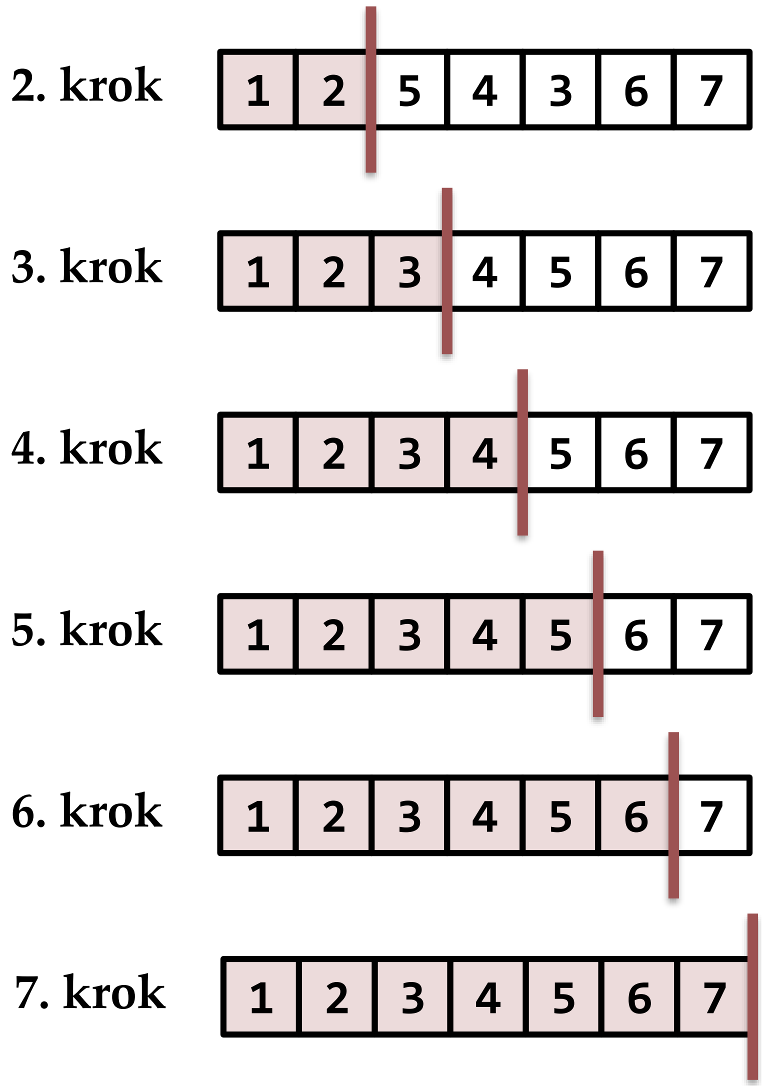
- Väčšinou sa predpokladá, že dátový súbor je v pamäti (model RAM)
- Čo keď chcete usporiadať výrazne väčšiu postupnosť?
  - Napr. 20 TB, ale pamäť máme len 8GB
- Hlavný princíp externého triedenia: **Rozdeliť postupnosť na menšie časti**, ktoré sa zmestia do dostupnej pamäte, usporiadať ich štandardnými algoritmami a nakoniec **zlúčiť ich** algoritmom zlučovania s lineárnou zložitosťou
- Externé zlučovanie je výhodné aj v prípade externých záznamových médií so sekvenčným prístupom (magnetické pásky, rotačné disky, ...)

Triedenie výberom (Select sort)

- Najjednoduchší-najprirodzenejší algoritmus
- Algoritmus:
  - Najmenší prvok môžeme zaradiť na začiatok vstupného poľa (najmenší vymeníme s prvkom, ktorý je nazačiatku)
  - Najmenší prvok zo zvyšku poľa bude druhý najmenší, atď.
- Označujeme aj MinSort / MaxSort



Triedenie výberom (Select sort)



Najlepší prípad?

Najhorší prípad?

Priemerný prípad?

Miera usporiadanosti vstupnej postupnosti poľa nemá vplyv na časovú zložitosť – vždy sa vykoná maximálny počet porovnaní. Ovplyvniť môžeme len počet výmen, ktorých je ale vždy menej ako porovnaní.

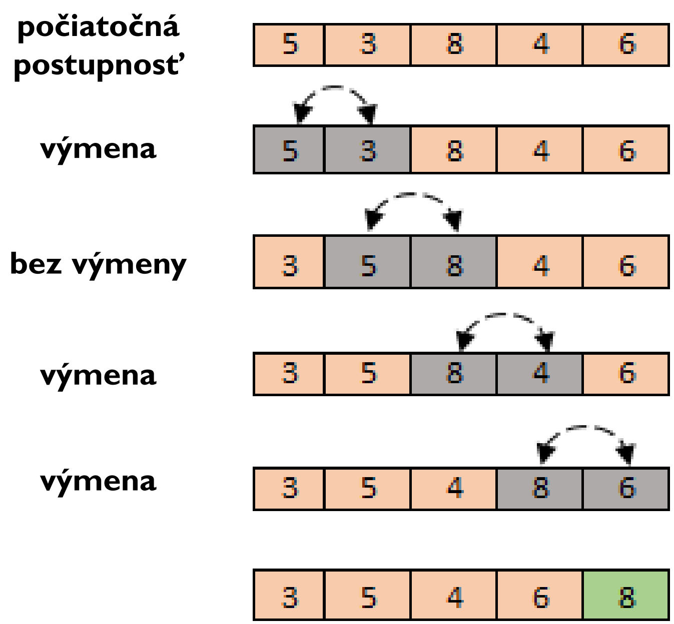
Usporadúvanie výmenami (Bubble sort)

- Pri usporadúvaní porovnáva dva susedné prvky a ak nie sú v správnom poradí, vymenia sa
- Procedúra sa opakuje, až kým nie sú prvky usporiadané (nie sú potrebné ďalšie výmeny)

```
koniec = 0;
while (!koniec)  // opakujeme, kym su neusporiadane
{
  koniec = 1;
  for (i = 0; i < n-1; i++)
    if (a[i] > a[i+1])
    {
      swap(&a[i], &a[i+1]); // vymen prvky i a i+1
      koniec = 0;
    }
}
```

Bubble sort – príklad

- Jeden prechod vnútorného cyklu
  - presun najväčšieho prvku na koniec



- Opakujeme prechody, až kým nie je všetko usporiadané
  - Zistím tak, že spravím prechod pri ktorom nebolo potrebné vymeniť žiadnu dvojicu susedných prvkov

Analýza zložitosti (Bubble sort)

- jeden prechod = presun najväčšieho prvku na koniec
- i-tý prechod: n-i+1 operácií
- Najlepší prípad: 1 prechod :) O(n)
- Najhorší prípad:
  $(n - 1) + (n - 2) + \dots + 1 = (n - 1) * n / 2 = O(n^2)$
- Implementačne jednoduchý ale výpočtovo neefektívny

Problémy:

- Čo keď najmenší prvok je na konci?
  Až v poslednom prechode bude na začiatku
- Vylepšenia sa snažia vylepšiť tento (a podobné) prípady

Shell sort

- Usporadúvanie vkladaním so zmenšovaním prírastku
- Zovšeobecnenie triedenia vkladaním (Insert sort) a bublinkového (Bubble sort)
- Dobrá implementácia je jedna z najrýchlejších pre usporiadanie kratších postupností (do 1000 prvkov)
- Netriedi naraz celú postupnosť, ale pre prírastok h utriedi Insert sort-om vybranú podpostupnosť prvkov vzdialených h (pre všetky možné začiatky i):

```
for(h = n/2; h > 0; h = h/2) // zmensujuce sa prirastky
  for (i = 0; i < h; i++)
    insert_sort(a[i,i+h,i+2*h,...]);
```

- Postupnosť zmenšujúcich sa prírastkov, posledný h=1

Shell sort – príklad

1. krok, prírastok 4 (n/2),

(vyznačené čísla sa usporiadajú vkladaním)

6 4 5 2 **8** 3 1 7 –> **6** 4 5 2 **8** 3 1 7

6 **4** 5 2 8 **3** 1 7 –> 6 **3** 5 2 8 **4** 1 7

6 3 **5** 2 8 4 **1** 7 –> 6 3 **1** 2 8 4 **5** 7

6 3 1 **2** 8 4 5 **7** –> 6 3 1 **2** 8 4 5 **7**

2. krok, prírastok 2

6 3 **1** 2 **8** 4 **5** 7 –> **1** 3 **5** 2 **6** 4 **8** 7

1 **3** 5 **2** 6 **4** 8 **7** –> 1 **2** 5 **3** 6 **4** 8 **7**

3. krok, prírastok 1

1 2 5 3 6 4 8 7 –> 1 2 3 4 5 6 7 8

Analýza zložitosti (Shell sort) – prírastky

- Rôzne voľby postupnosti prírastkov vedú k rôzne efektívnym verziám algoritmu
- Prírastky podľa Shell-a: $\lfloor n/2 \rfloor$, $\lfloor n/2^2 \rfloor$, $\lfloor n/2^3 \rfloor$,…, 1 alebo (ešte horšie) postupnosť prírastkov mocniny 2: **$O(n^2)$**
- Knuth: 1, 4, 13, 40, 121, 364, 1093, 3280, 9841,... t.j. $h_1 = 1, h_{i+1} = 3*h_i + 1$: blízko $O(n \log^2 n)$ a **$O(n^{1.25})$**
- Hibbard: 1, 3, 7,...2^{k-1}: **$O(n^{3/2})$**
- Sedgewick: 1, 8, 23, 77, 281, 1073, 4193, 16577..., t.j. $4^{i+1} + 3 \cdot 2^i + 1$ pre $i > 0$, má byť lepšia než Knuth
- Pratt: $\log^2 n$ prírastkov $2^i 3^j < \lfloor n/2 \rfloor$ (1, 2, 3, 4, 6, 8, 9, 12, 16, …) ... **$O(n \log^2 n)$**

Analýza zložitosti (Shell sort)

- Najlepší prípad: postupnosť je už usporiadaná – bude treba menej porovnaní
- Najhorší prípad (pre postupnosť prírastkov podľa Pratta): $O(n \log^2 n)$
- Priemerný prípad (pre postupnosť prírastkov podľa Pratta): $\Theta(n \log^2 n)$

Rýchle usporiadanie (Quicksort)

- Quicksort alebo usporadúvanie rozdeľovaním je jeden z najrýchlejších známych algoritmov založených na porovnávaní prvkov
- Priemerná doba výpočtu Quicksort-u je najlepšia zo všetkých podobných algoritmov
- Nevýhodou je, že pri nevhodnom usporiadaní vstupných dát môže byť časová aj pamäťová náročnosť omnoho väčšia
  - Quicksort – počet porovnaní v najhoršom prípade $O(N^2)$
  - Mergesort – počet porovnaní v najhoršom prípade $O(N \log N)$
  - V praxi je však Quicksort zvyčajne dva krát rýchlejší a používa polovičné množstvo pamäti...

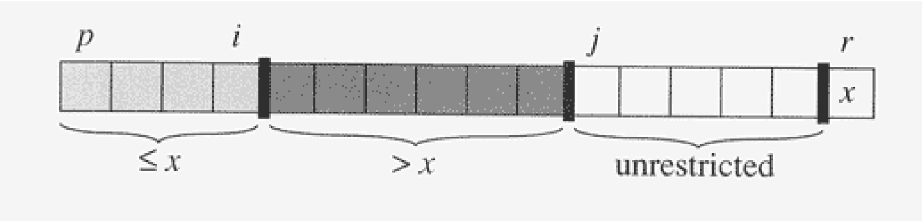
Quicksort – hlavná myšlienka

- Jeden prechod = rozčlenenie prvkov na dve podpostupnosti podľa pivota x: prvky ≤ x, prvky > x
- Rekurzívne usporiadať podpostupnosti

```
QUICKSORT(A, p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

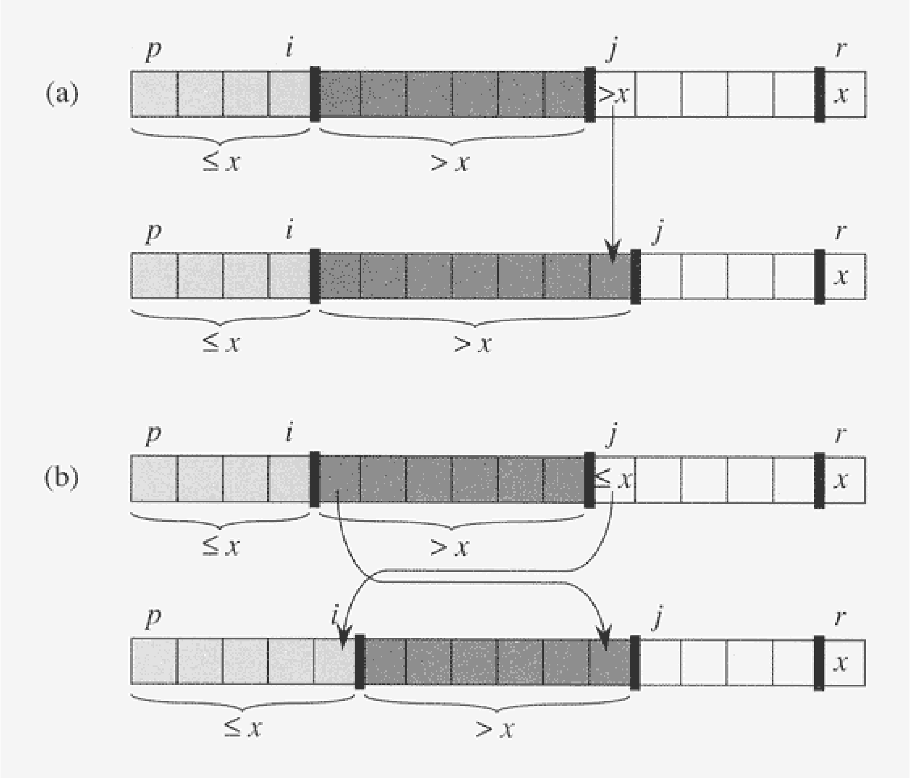
```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Quicksort – rozčlenenie



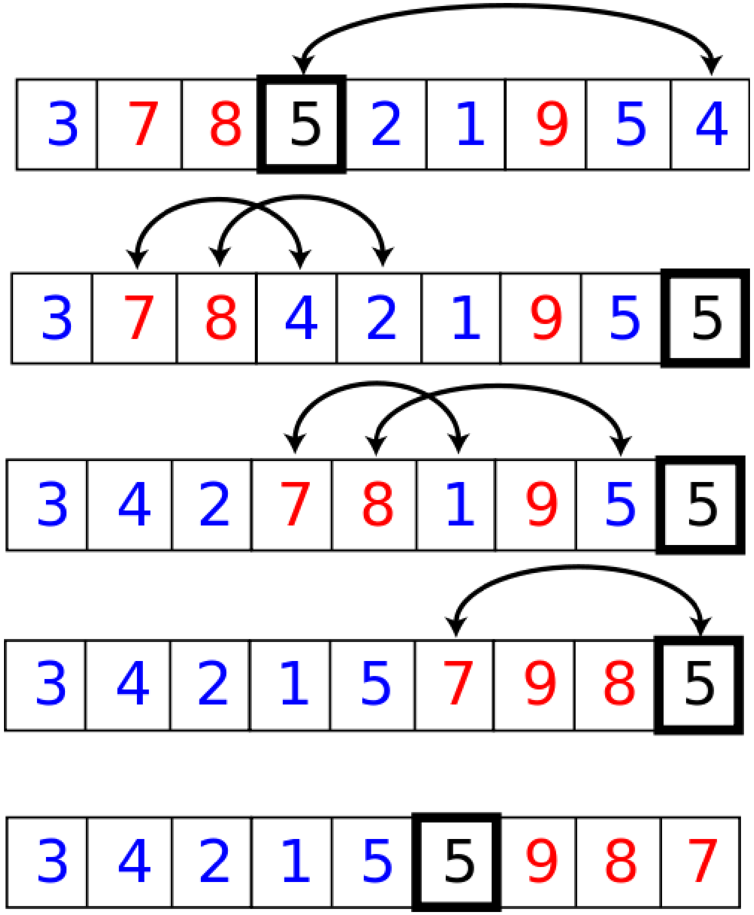
- Prvky A[p..i] sú menšie alebo rovné x
- Prvky A[i+1..j-1] sú väčšie x
- Prvky A[j..r-1] sú ešte nerozčlenené

- Pri rozčleňovaní ďalšieho prvku (j) môžu nastať dva prípady:
  a. A[j] > x, len posuniem j
  b. A[j] ≤ x, presuniem prvok na i-tu pozíciu, posuniem i a j



Quicksort – rozčleňovanie – príklad

1. Voľba pivota – ktorý prvok?
2. Presunúť pivot na koniec
3. Rozčlenenie postupnosti
4. Presunúť pivot medzi rozčlenené podpostupnosti
5. Rekurzívne usporiadanie podpostupností



Analýza rýchleho usporiadania

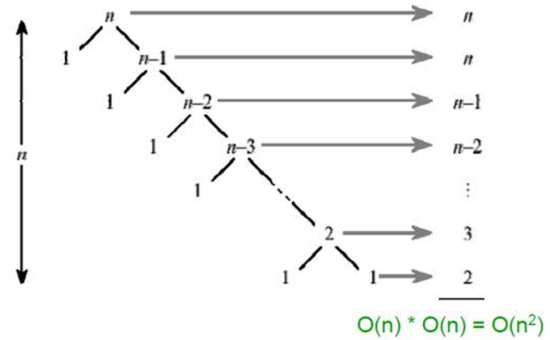
- Najhorší prípad: vždy zle vyvážené rozčlenenie

  Usporiadaná postupnosť :(

  $T(1) = \Theta(1)$

  $T(n) = T(n - 1) + \Theta(n)$

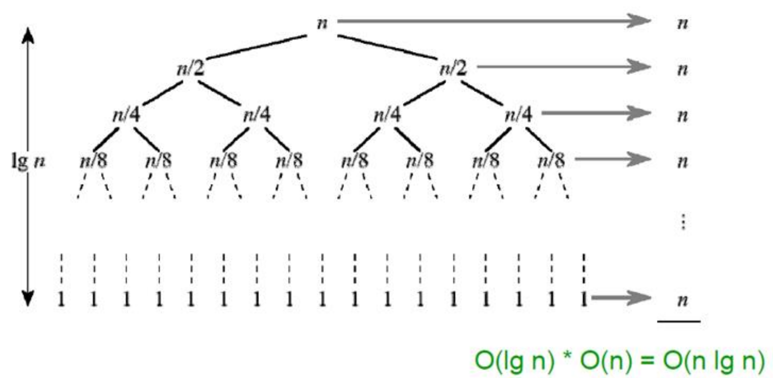
  $T(n) = \Theta(n^2)$



- Najlepší prípad: vždy dokonale vyvážené rozčlenenie

  $T(n) = 2T(n/2) + \Theta(n)$

  $T(n) = \Theta(n \lg n)$



- Priemerný prípad: náhodný

  Napr. aj pre rozčlenenie 9 ku 1

  $T(n) = T(9n/10) + T(n/10) + n$

  $T(n) = \Theta(n \lg n)$

Výber pivota

- Krajný prvok – konštantný čas O(1)
  nemusí byť vhodný
- Ktorý by bol najlepší?
  - Taký, pre ktorý je počet prvkov rozčlenených podpostupností rovnaký (alebo čo najbližšie k sebe)
  - **Medián**
- Dobrý algoritmus: **vybrať náhodný pivot**

```
RANDOMIZED-PARTITION(A, p, r)
  i ← RANDOM(p, r)
  exchange A[r] ↔ A[i]
  return PARTITION(A, p, r)
```

Nájdenie k-teho najmenšieho prvku

- Daná postupnosť A[1..n] (neusporiadaných) čísel a celé číslo k ($1 \leq k \leq n$). Úloha je nájsť k-te najmenšie číslo v A.
- Špeciálne prípady
  - pre k=1 ide o nájdenie najmenšieho prvku,
  - pre k=n ide o nájdenie najväčšieho prvku,
  - ak n je nepárne, k=(n+1)/2 dá medián
  - ak n je párne, podľa dohody je medián niečo medzi prípadmi k=floor((n+1)/2) a k=ceiling((n+1)/2)

Nájdenie k-teho najmenšieho prvku

- Prvý nápad na riešenie:
  usporiadať pole A a vybrať A[k]
  - usporiadať vieme na mieste O(n log n). Výber A[k] je O(1). Spolu O(n log n).
- Dá sa to rýchlejšie?
  - Ak máme už dané k, tak usporiadaním sa urobilo viac práce ako je potrebné na určenie k-teho najmenšieho prvku. Prečo?
  - Ak by k nebolo vopred dané, tak by práve usporiadané pole dávalo k-ty najmenší prvok pre ľubovoľné k.

Nájdenie k-teho najmenšieho prvku

- Druhý nápad:
  nájsť najmenší prvok v poli A a odstrániť ho. Pokračovať nájdením najmenšieho prvku vo zvyšku poľa A, odstránením atď. Opakovať k-krát.
  - nájsť minimum a odstrániť ho vieme O(n).
    Spolu O(k*n).
- Je to rýchlejšie?
  - závisí od porovnania k a log(n).
  - pre veľké k (väčšie ako log n) je lepší prvý nápad
  - pre malé k je druhý nápad lepší

Nájdenie k-teho najmenšieho prvku

- Dá sa to rýchlejšie?
  - všimnime si, že v prvom aj druhom prípade dostaneme na výstupe **k najmenších prvkov usporiadaných**.
  - Ale toto (usporiadanie) týchto k prvkov nepotrebujeme! Robíme robotu navyše.
  - Výzva je určiť LEN k-ty prvok a urobiť to v čase O(n).

- Blum, Floyd, Pratt, Rivest a Tarjan, 1973
  Algoritmus s mediánom mediánov ako pivotom:

```
Select (A, k)
1.  x = median(A) //akurát, že zatiaľ nevieme ako v O(n)
2.  rozčleň A podľa pivota x. Nech je m-1 prvkov takých, že A[i]<x.
    Potom bude A[m]=x a n-m prvkov bude takých, že A[i]>x.
3.  if k=m then return x
            else if k<m then Select (A[1..m-1], k)
                    else Select (A[m+1..n], k-m)
```

Algoritmus medián mediánov – zložitosť

$T_{select}(n)=T_{select}(n/2) + n + T_{\mathrm{median}}(A)$

- predpokladajme, že $T_{\mathrm{median}}(A)$ je $O(n)$ a preto

  $T_{select}(n)=T_{select}(n/2)+n$
- riešenie je n+n/2+n/4+…+1 = 2n-1

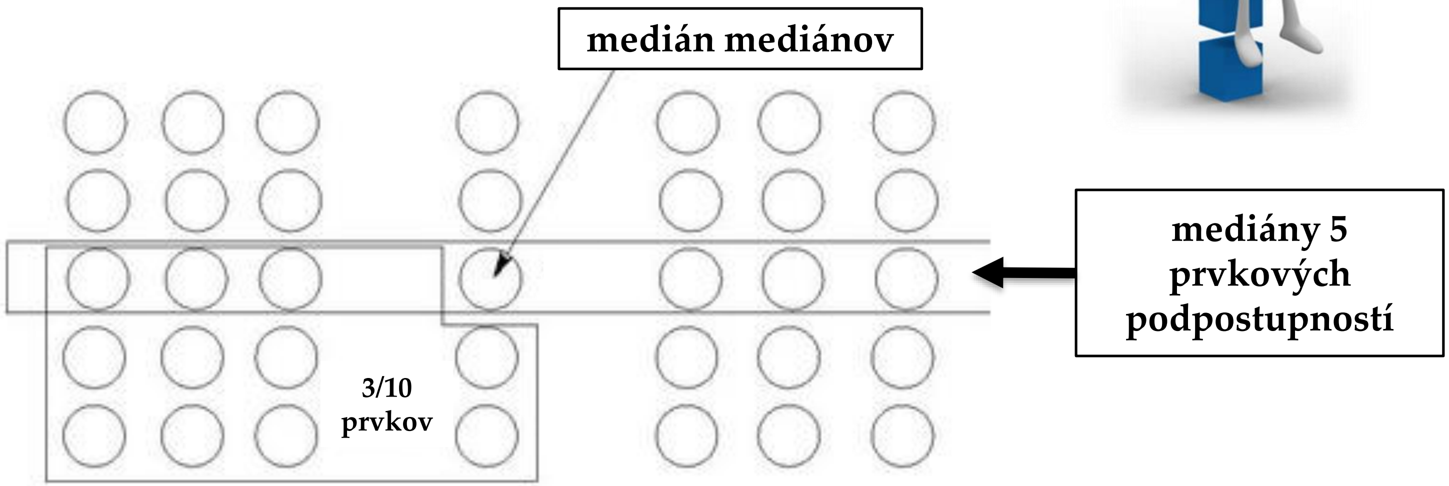
Celkovo $O(n)$

- Predpokladáme, že $T_{\mathrm{median}}(A)$ je $O(n)$ a teda nám stačí približný medián, napr. taký, ktorý je zaručene väčší než 3/10 všetkých prvkov a menší než 3/10 všetkých prvkov. Presnejšie, uvažujme približný medián taký, že je x-tý najmenší zo všetkých prvkov v A a platí

$$3n/10 \le x \le 7n/10$$

Medián mediánom z piatich

1. Rozdeľ vstupnú postupnosť n prvkov do skupín po piatich (a možno jednej zvyškovej)
2. Nájdi medián každej skupiny (usporiadaním alebo natvrdo tretí najmenší) – dostaneš n/5 mediánov.
3. Rekurzívne Select(„n/5 mediánov“, n/10)



Zložitosť: $T_{select}(n) = T_{select}(n/5) + T_{select}(7n/10) + n$ – *Celkovo O(n)*

Porovnávacie algoritmy

- Algoritmus usporadúvania, ktorý prechádza vstupné kľúče a na základe operácie **porovnávania** rozhoduje, ktorý z dvoch prvkov sa má v usporiadanom poli objaviť ako prvý

- Operácia **porovnávania** musí mať tieto vlastnosti:

  Ak $a \le b$ a $b \le c$, tak $a \le c$

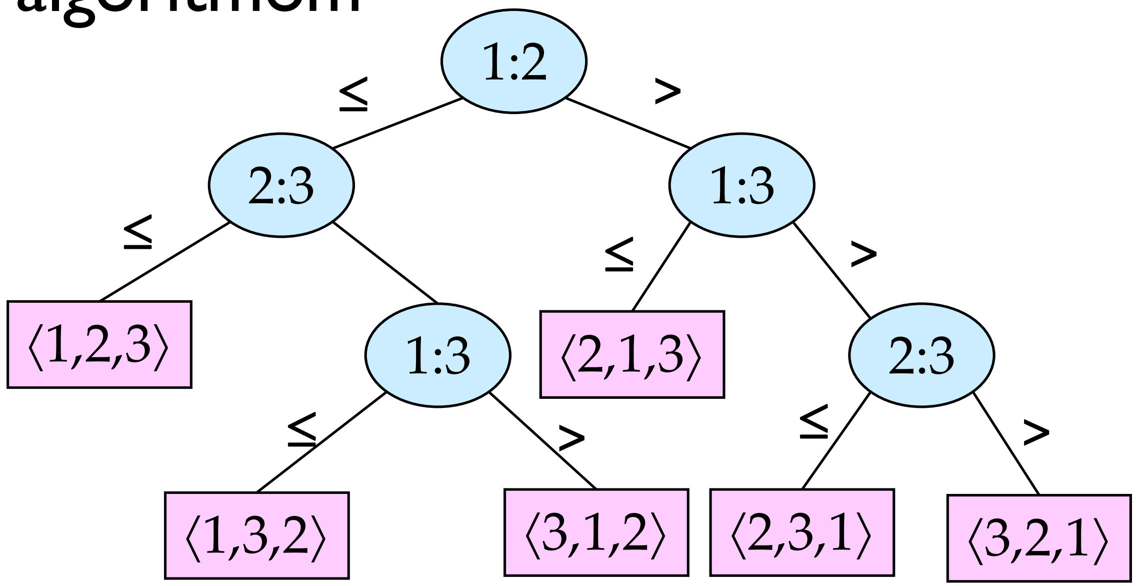
  Pre všetky a a b, buď $a \le b$ alebo $b \le a$

- Základným limitom je dolné ohraničenie počtu potrebných porovnávaní **Ω(n log n)**, ktoré je v najhoršom prípade potrebné na usporiadanie postupnosti

- **Prečo Ω(n log n)?**

Rozhodovací strom

- Model porovnaní vykonaných porovnávacím algoritmom



obsahuje 3! = 6 listov
= všetkých 6 možných výsledkov
do každého listu ide práve jedna cesta
počet uzlov pozdĺž cesty = počet nutných porovnaní

vstup: 3 prvky

rozhodovací strom zaznamenáva všetky nutné porovnania, aby pre ľubovoľnú vstupnú permutáciu prvkov našlo ich usporiadanie

uzol *i:j* znamená: porovnaj kľúče $k(a_i)$ s $k(a_j)$.

listy ukazujú permutácie π indexov také, že platí

$k(a_{\pi(1)}) \leq k(a_{\pi(2)}) \leq (a_{\pi(3)})$

Dolný odhad najhoršieho prípadu

- Výška rozhodovacieho stromu = Dĺžka najdlhšej cesty z koreňa do ľubovoľného listu v rozhodovacom strome príslušného algoritmu usporadúvania
- Označme **h** výšku stromu
- V binárnom strome výšky **h** nie je viac než $\mathbf{2^h}$ uzlov
- Tento strom má **n!** listov, teda: **n!** $\leq$ $\mathbf{2^h}$ preto **h** $\geq$ **log(n!)**
- Odhadnime $n! \geq n \cdot (n-1) \cdot \ldots \cdot 2 \geq n \cdot (n-1) \cdot \ldots \cdot \frac{n}{2} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$
- Potom:

$$h \geq \log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{n}{2} \log \left(\frac{n}{2}\right) = \frac{1}{2} n(\log n - 1) = \Omega(n \log n)$$

Výhody porovnávacích algoritmov

- Výhody porovnávacách algoritmov
  - Použiteľné as-is pre rôzne dátové typy
    - Čísla, reťace, ...
  - Jednoduchá implementácia porovnávania n-tíc v lexikografickom usporiadaní
  - Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

- Ako prekonať teoretický limit **Ω(n log n)** porovnávacích algoritmov?
  - Zbaviť sa porovnávania prvkov :)
    (budeme vyšetrovať štruktúru hodnôt kľúčov)
  - Obetovať priestorovú zložitosť

Usporadúvanie spočítavaním (Counting sort)

- Usporadúvanie výpočtom poradia
  - Neporovnávame kľúče!
  - Pokúsime sa priamo určiť jeho poradie v postupnosti
- Vstup:
  n čísel v rozsahu 0..k-1

- Určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli

Usporadúvanie spočítavaním (Counting sort)

- Určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli

- Algoritmus pracuje s tromi poliami:
  - Pole *a[0..n-1]* obsahuje údaje, ktoré sa majú usporiadať
  - Pole *b[0..n-1]* obsahuje konečný usporiadaný zoznam údajov
  - Pole *c[0..k-1]* je použité na počítanie počtu prvkov

```
// pocet vyskytov konkretnej hodnoty
for(i = 0; i < n; i++)
  c[a[i]]++;

// prefixove sucty: urcime index posledneho prvku s hodnotou j
for(j = 1; j < k; j++)
  c[j] = c[j] + c[j-1];

// prvky z pola a vložíme na prislusny index v poli b
for(i = n-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
```

Usporadúvanie spočítavaním (Counting sort)

- Koľko operácií algoritmus vykoná?
  - **rádovo n+k+n**
- Koľko pomocnej pamäte potrebuje?
  - **pole veľkosti n a pole veľkosti k**
- Vhodný len pre malé k << n
- Pre veľký rozsah (int) je potrebné veľa pomocnej pamäte

```
// pocet vyskytov konkretnej hodnoty
for(i = 0; i < n; i++)
  c[a[i]]++;

// prefixove sucty: urcime index posledneho prvku s hodnotou j
for(j = 1; j < k; j++)
  c[j] = c[j] + c[j-1];

// prvky z pola a vložíme na prislusny index v poli b
for(i = n-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
```

Counting sort – príklad

vstup

	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	7	1	3	1	2	4	5	7	2	4	3

k

pomocná pamäť

	1	2	3	4	5	6	7
$C[1..k]$	2	2	2	2	1	0	2

dve 1 v A

prefixové sumy

	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

6 prvkov ≤ 3

Counting sort – príklad

$A[1..n]$

výstup

$B[1..n]$

1	2	3	4	5	6	7	8	9	10	11
					3					

$B[6] = B[C[3]] = B[C[A[11]]] = A[11] = 3$

C

1	2	3	4	5	6	7
2	4	6	8	9	9	11

$C[A[11]] = C[A[11]] - 1$

C

1	2	3	4	5	6	7
2	4	5	8	9	9	11

Stabilný algoritmus

- Algoritmus usporadúvania je stabilný, ak vždy zachová pôvodné poradie prvkov s rovnakými kľúčmi

- Ak prvky s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu (napr. ak kľúčom je samotný prvok)

- Zachovať pôvodné poradie prvkov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa priezviska a potom podľa mena.

Stabilný algoritmus (2)

- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamätá pôvodné poradie prvkov a pri zhodných kľúčoch sa berie do úvahy toto poradie

- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča usporiadania, ktorý je zložený z primárneho, sekundárneho, atď.
  - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť

Stabilný algoritmus (3)

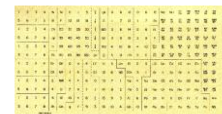
- Príklad – dvojice (kľúč, prvok):
  (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
  (2, 7) (2, 3) (4, 5) (5, 6) – zachované poradie prvkov s kľúčmi 2 – stabilné usporiadanie
  (2, 3) (2, 7) (4, 5) (5, 6) – zmenené poradie prvkov s kľúčmi 2 – nestabilné usporiadanie

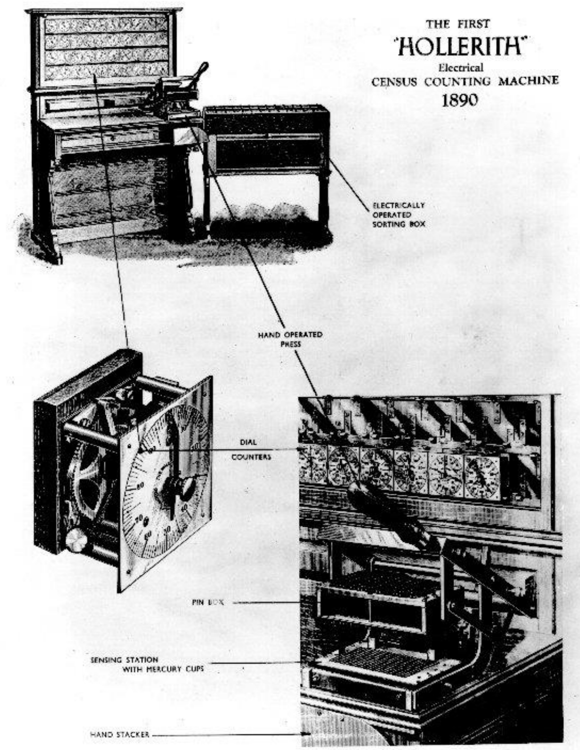
- Príklad na viacnásobné usporiadanie – dvojice (kľúč 1, kľúč 2):
  (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa kľúča 2, potom podľa kľúča 1:
  (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2
  (**2**, 3) (**2**, 7) (**4**, 5) (**4**, 6) – podľa kľúča 1
- Usporiadanie najprv podľa kľúča 1, potom podľa kľúča 2:
  (**2**, 7) (**2**, 3) (**4**, 5) (**4**, 6) – podľa kľúča 1
  (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2 – narušené poradie

- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa kľúčov so zvyšujúcou sa prioritou

Radixové usporadúvanie

- Spracovanie sčítania ľudu USA 1880 trvalo skoro 10 rokov (robí sa každých 10 rokov)
- Herman Hollerith (1860-1929)
- Ako prednášateľ na MIT navrhol prototyp strojov na spracovanie diernych štítkov, doba spracovania ďalšieho sčítania ľudu v 1890 sa tým skrátila na 6 týždňov
- Základná myšlienka: **začni triediť podľa najnižšieho rádu**
- Založil firmu Tabulating Machine Company (1911), ktorá sa spojila s ďalšími firmami v 1924 - vznikla IBM (International Business Machines)





Radixové usporadúvanie – schéma

```
RADIX-SORT(A, d)
    for i ← 1 to d
      do stabilné usporadúvanie(A) podľa i-tej číslice
```

- Radixové usporadúvanie **neporovnáva dva celé kľúče**, ale spracúva a **porovnáva len časti kľúčov**
- Kľúče považuje za čísla zapísané v číselnej sústave so základom *k (radix, koreň)*, pracuje s jednotlivými číslicami:

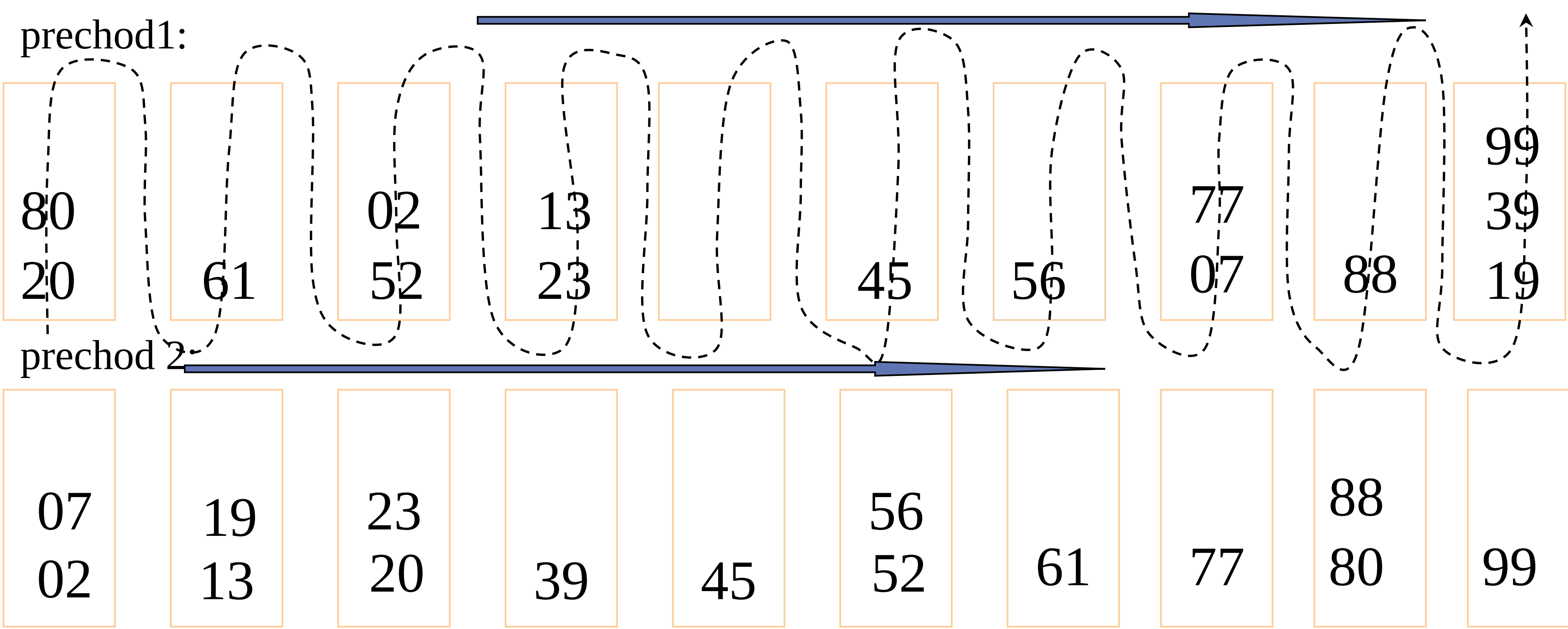
$$\text{hodnota} = x_{d-1}k^{d-1} + x_{d-2}k^{d-2} + \dots + x_2k^2 + x_1k^1 + x_0k^0$$

- Dokáže usporadúvať čísla, znakové reťazce, dáta, ...
(počítače reprezentujú všetky údaje ako postupnosti 1 a 0 – binárna sústava => 2 je základ)
  - Uvažujme problém: usporiadať milión 64-bitových čísiel
  - Prvé riešenie: 64 prechodov cez milión čísiel?
  - Lepšie riešenie: interpretovať ich ako čísla v sústave so základom (radixom) 2^{16}, budú to najviac 4-miestne čísla ... vtedy to algoritmus usporiada len v 4 prechodoch!

Radixové usporadúvanie – príklad

- usporiada množinu čísiel vo viacerých prechodoch, začínajúc od číslic najnižšieho (jednotkového) rádu, potom usporiada podľa číslic najbližšieho vyššieho (desiatkového) rádu atď.

príklad: 23, 45, 7, 56, 20, 19, 88, 77, 61, 13, 52, 39, 80, 2, 99



Radixové usporadúvanie – príklad

now		sob		tag		ace
for		nob		ace		bet
tip		ace		bet		dim
ilk		tag		dim		for
dim		ilk		tip		hut
tag		dim		sky		ilk
jot	→	tip	→	ilk	→	jot
sob		for		sob		nob
nob		jot		nob		now
sky		hut		for		sky
hut		bet		jot		sob
ace		now		now		tag
bet		sky		hut		tip

Radixové usporiadanie

- LSD Radix sort (least significant digit) – usporadúvanie podľa číslic postupuje od poslednej číslice (s najmenšou váhou) k prvej číslici (s najväčšou váhou)– stabilný.
- MSD Radix sort – od prvej číslice k poslednej – lexikografické usporiadanie – nestabilný
- Je dôležité na samotné usporadúvanie podľa jednotlivých číslic použiť nejaký stabilný algoritmus, aby sa nemenilo poradie prvkov s rovnakými číslicami jednej váhy pri usporadúvaní podľa inej váhy.
- Keďže počet možných číslic (ak k=10) je len 10, tak na usporiadanie podľa nich je výhodné použiť usporadúvanie spočítavaním.

Radixové usporiadanie – správnosť

- Predpokladajme, že postupnosť je podľa číslic nižších rádov {j: j<i} usporiadaná
- Treba ukázať, že usporiadanie podľa nasledujúcej číslice rádu i zanechá postupnosť usporiadanú (podľa nižších rádov ale už vrátane i)
  - ak sú dve číslice na i-tom ráde (mieste odspodu) rôzne, usporiadanie dvoch čísiel podľa tohto rádu je správne (je vyšší rád než všetky, podľa ktorých sa usporadúvalo doteraz, keďže sa ide od najnižšieho, nižšie rády sú irelevantné)
  - ak sú dve číslice na i-tom ráde (mieste odspodu) rovnaké, čísla sú už usporiadané podľa nižších rádov. ak používame stabilný algoritmus, čísla zostanú v správnom poradí

Aký algoritmus použiť na usporiadanie podľa číslic?

- Ponúka sa usporadúvanie spočítavaním (Counting sort):
  - usporiada *n* čísiel podľa číslic v sústave so základom *k*, t.j. rozsah číslic je 0..*k*-*1*
  - čas: O(*n* + *k*)
- Každý prechod cez *n* *d*-miestnych čísiel (s *d číslicami)* si vyžiada čas O(*n*+*k*), takže celkový čas je O(*dn*+*dk*)
  - ak *d* je konštantné a *k*=O(*n*), vyžiada si čas O(*n*)

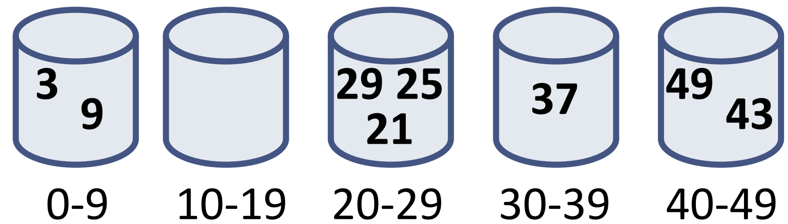
Vedierkové usporadúvanie (Bucket sort)

- Predpokladá, že vstup je akoby generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale
- Rozdelí interval na *n* rovnako veľkých disjunktných podintervalov (vedierok – bucketov) a potom do nich rozmiestni vstupné čísla
- Osobitne v každom vedierku sa potom tieto čísla usporiadajú

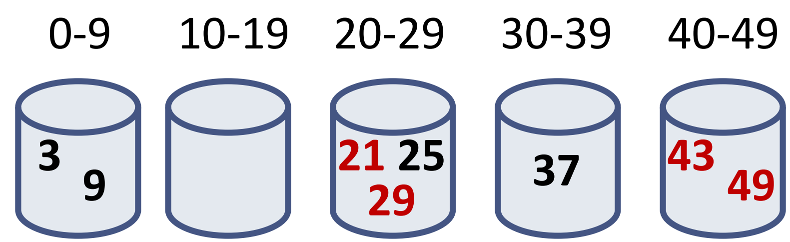
Vedierkové usporadúvanie – príklad

- Vytvoria sa prázdne vedierka veľkosti M/n (M – maximálna hodnota vstupného poľa, n – počet prvkov vstupného poľa)
- Rozptýlenie – prechádzanie vstupným poľom a rozmiestnenie každého prvku do prislúchajúceho vedierka

29 25 3 49 9 37 21 43

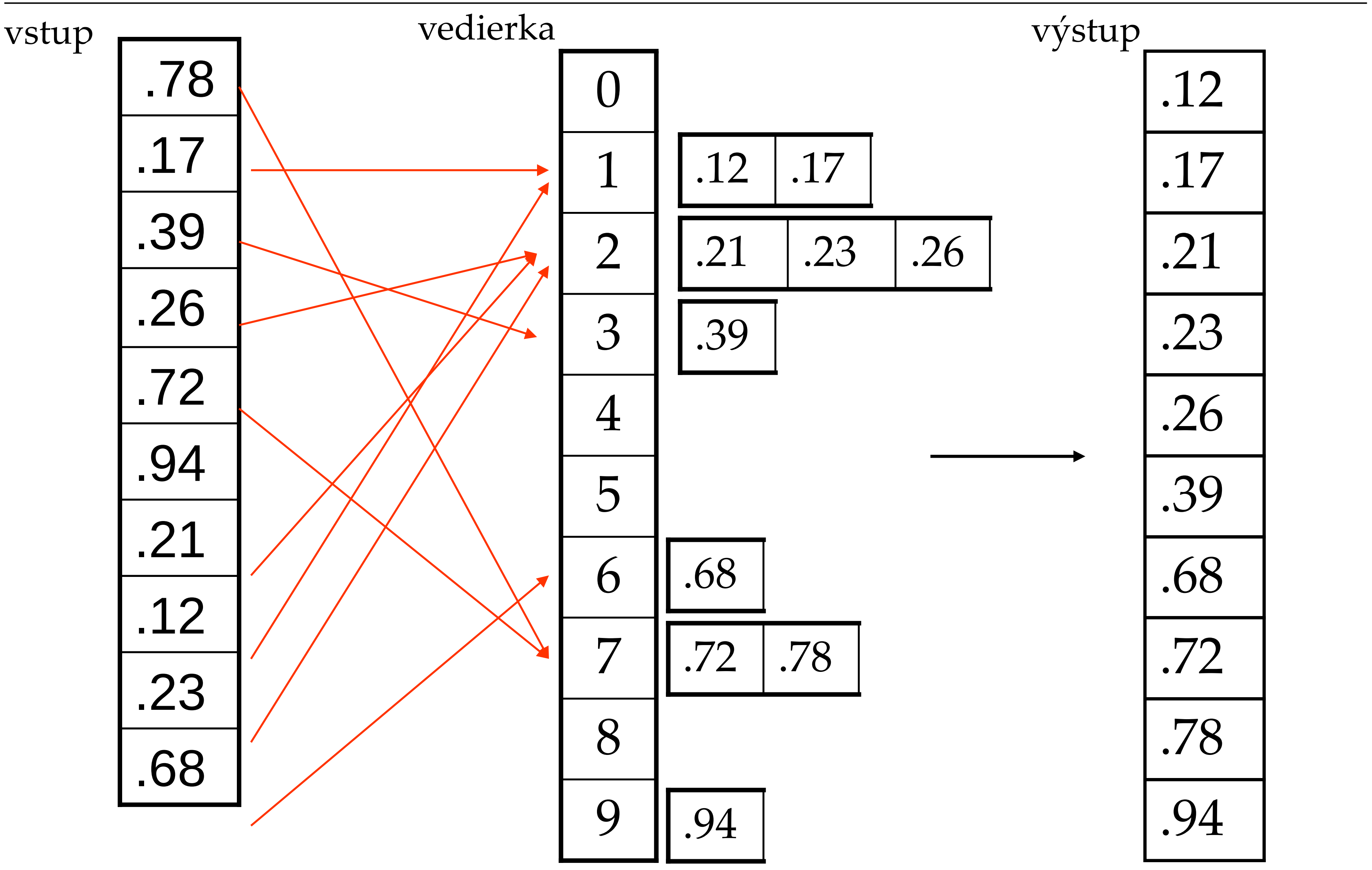


- Usporiadanie naplnených vedierok
- Zreťazenie vedierok – postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa



3 9 21 25 29 37 43 49

vedierko i obsahuje hodnoty z polouzavretého intervalu [i/10, (i + 1)/10).



Analýza zložitosti (Bucket sort)

- Jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)

- Činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zreťazenie vedierok do výslednej postupnosti trvajú **O(n)**

- Výpočtová zložitosť usporiadania prvkov vo vedierkach Insert sortom **O(n^2)**

Analýza zložitosti (Bucket sort)

- Výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvky rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, tak časová zložitosť Insert sortu $O(n^2)$ prevažuje nad lineárnou zložitosťou a predstavuje výslednú zložitosť celého usporadúvania
- Takýto stav sa môže vyskytnúť ak rozsah prvkov *m* je oveľa väčší ako ich počet
- Preto sa niekedy celková zložitosť značí podobne ako pri Counting sorte $O(n+m)$. Ak $m=O(n)$, tak výsledná časová zložitosť je $O(n)$
- Ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a preto sa za priemernú zložitosť berie $O(n)$

Niektoré ďalšie algoritmy

- Využitím haldy (Heap sort)
- Využitím vyhľadávacieho stromu (Tree sort)
- Knižničné usporadúvanie (Library sort)
- Varianty bublinkového (Coctail sort)
- Varianty rýchleho (Introspective sort)
- Optimálny počet zápisov (Cycle sort)
- Vhodný na paralelne architektúry (Odd-even sort)
- Pasiánsové usporiadanie (Patience sort)
- ...