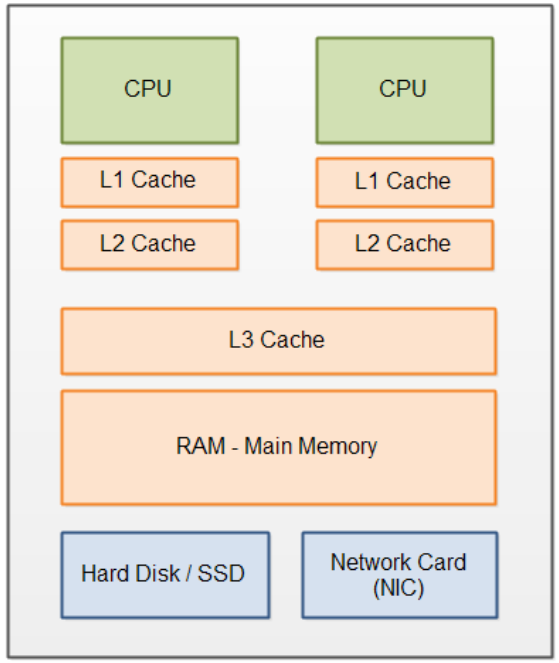
Datové struktury 1

4. přednáška

Pavel Veselý

Plán

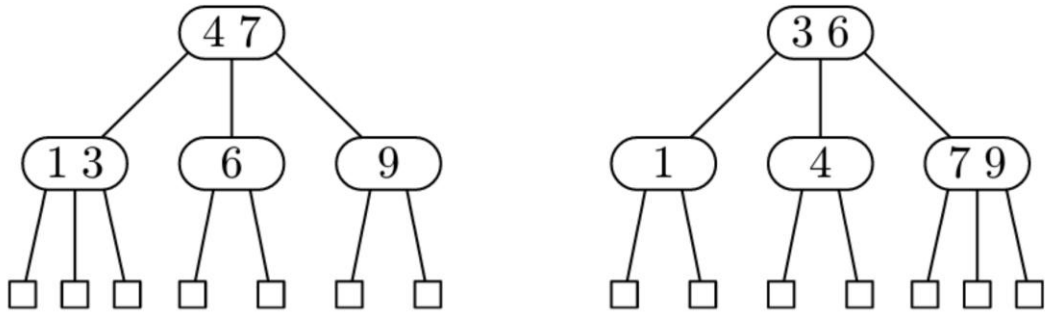
- (a, b)-stromy stromy – dokončení
- DS přátelské k CPU cache
  - A obecně paměťovým hierarchiím



Zdroj: https://flog.pravda.sk/loxodonta.flog?foto=679353

Vícecestný vyhledávací stromy: (a, b)-stromy

- Ve vnitřním vrcholu $\ell \geq 1$ klíčů (seřazených), $\ell + 1$ synů
  - Listy prázdné (`None` / `nullptr`)
- (a, b)-stromy pro $a \geq 2$ a $b \geq 2a - 1$
  - Vnitřní vrchol má a až b synů, kořen má 2 až b synů
  - Listy na stejné hladině
  - Hloubka = # hran od kořene do listů



Obrázek 8.7: Dva (2, 3)-stromy pro tutéž množinu klíčů

Lemma: Hloubka (a, b)-stromu je mezi $\log_b(n+1)$ a $1 + \log_a \frac{(n+1)}{2}$

(a, b)-stromy: amortizovaná složitost

- Změny stromu jsou dražší
- Vyhledávání někdy není potřeba
  - Např.: máme odkaz na mazaný prvek nebo list pro Insert

Pozorování: m operací `Insert` na prázdném (a,b)-stromě provede $O(m)$ změn uzlů.

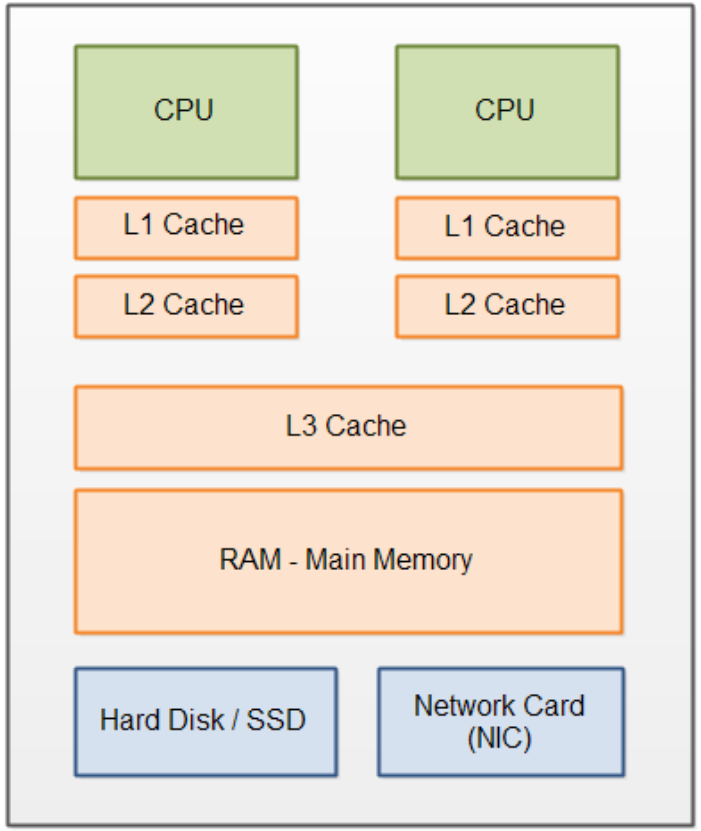
Věta: m operací `Insert` a `Delete` na $(a, 2a)$-stromě provede $O(n + m)$ změn uzlů.

(a, b)-stromy: top-down verze

- Pouze pro $b \geq 2a$
- `Insert` jedním průchodem z kořene dolů
  - Pokud má aktuální uzel $b - 1$ klíčů, preventivně rozdělíme
    - Výsledné uzly mají $\frac{b-2}{2} \geq a - 1$ klíčů + jeden přidáme do rodiče
- `Delete` jedním průchodem z kořene dolů
  - Pokud má aktuální uzel $a - 1$ klíčů
    - Provedeme půjčení od bratra nebo spojení s bratrem

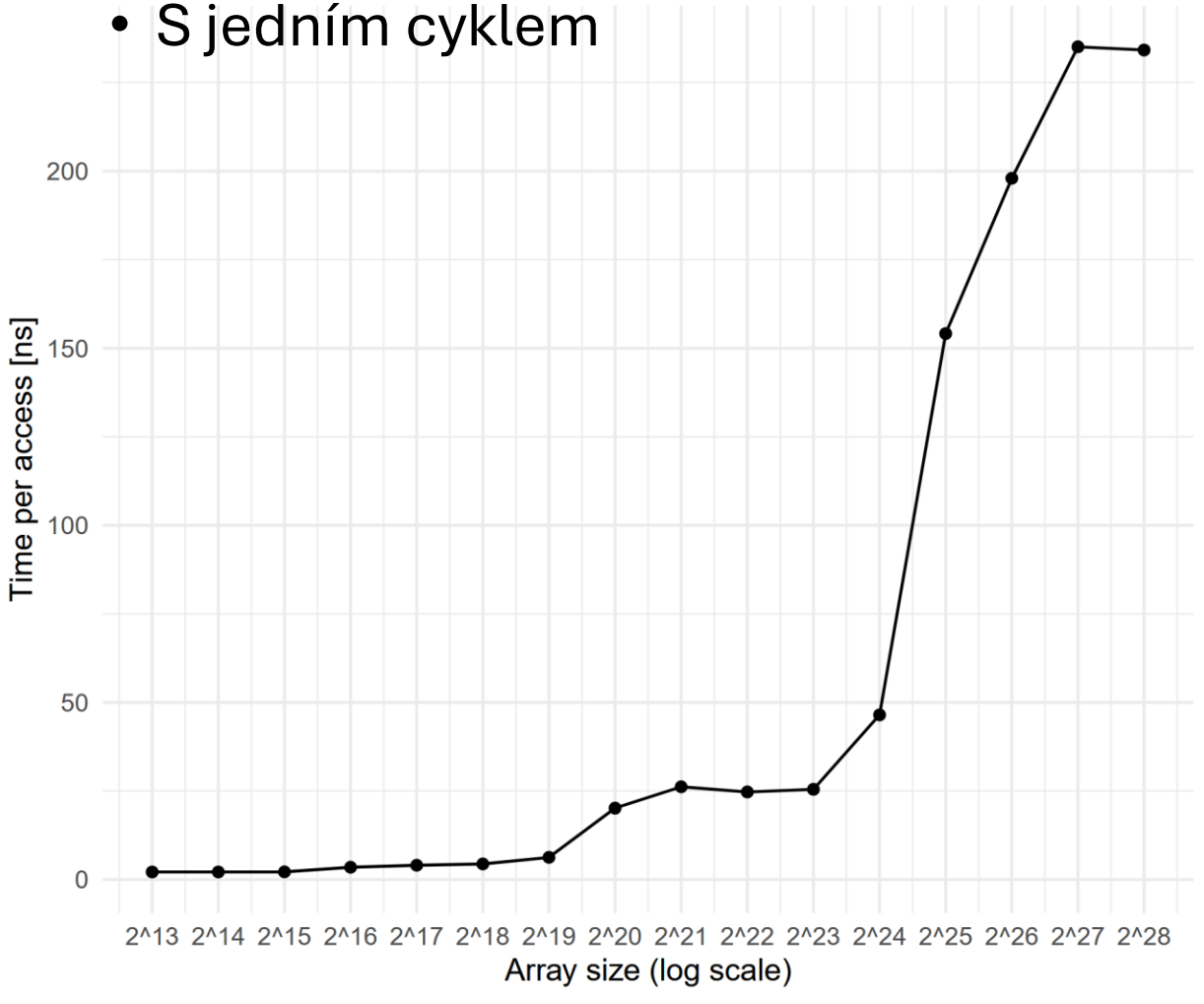
Paměťové hierarchie

- Př. AMD EPYC 7302 16-Core Processor
  - L1 cache (data): 16 KiB
    (16 ×, 8-way)
  - L2 cache: 512 KiB
    (16 ×, 8-way)
  - L3 cache: 16 MiB
    (8 ×, 16-way)
  - RAM: 256 GB



Příklad vlivu cache na čas běhu

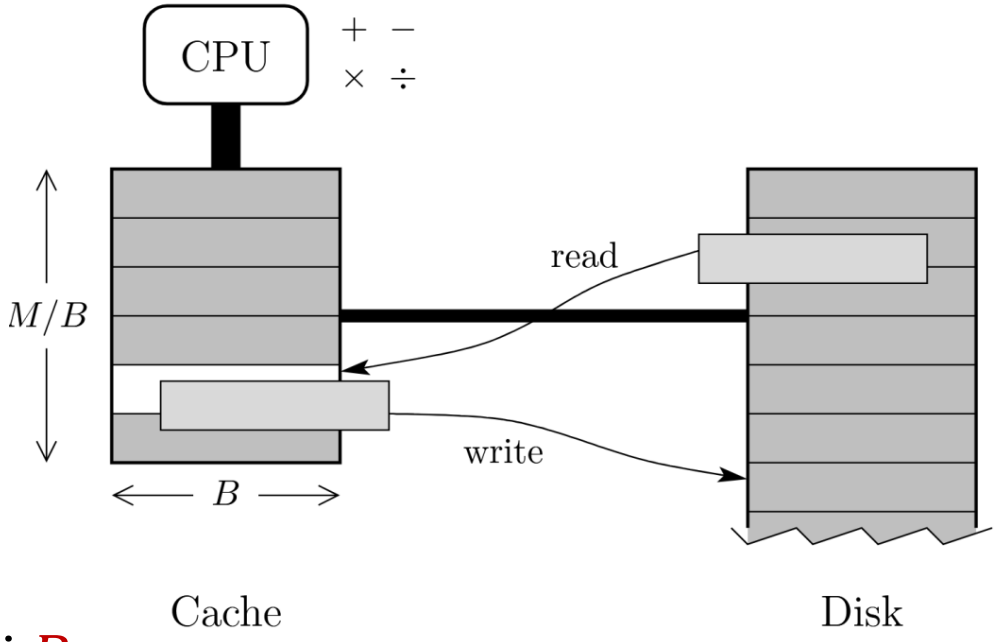
- Průchod polem v náhodné permutaci
  - S jedním cyklem




```
allocate array NEXT[]
randomize all indices to form one big cycle
cur = 0
start timer
repeat N times: cur = NEXT[cur] // pointer
chase
stop timer
print (elapsed_time / N) as ns per access
```

I/O složitost pro dvě úrovně

- Pomalá paměť: disk, externí paměť
  - Složena z bloků velikosti B
- Rychlá paměť: cache, interní paměť
  - Velikost M, také rozdělena na bloky velikosti B
- I/O složitost: # přesených bloků
  - Příklad sekvenční průchod polem délky N
    - I/O složitost $O(N/B + 1)$
    - Lze se zbavit té +1?
  - Většinou stačí omezit počet čtení

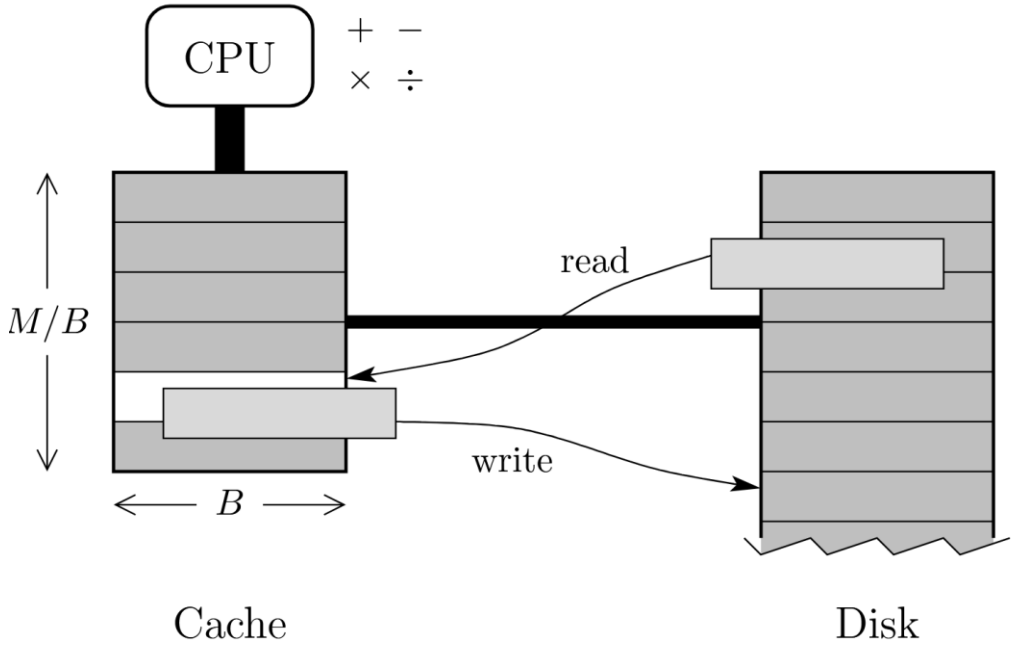


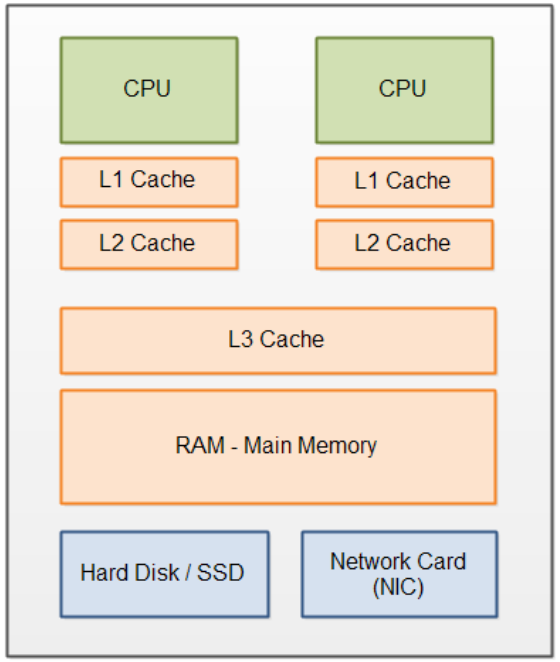
Zdroj obrázku: Demaine: Cache-Oblivious Algorithms and Data Structures

Teoretické modely

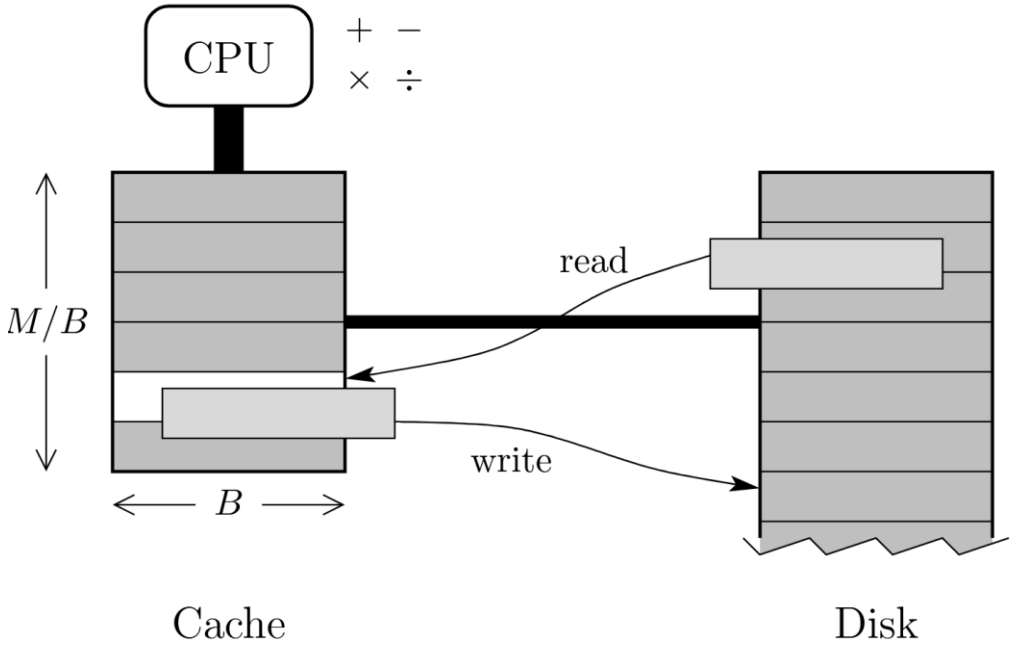
- I/O model
  a cache-aware algoritmy a DS
  - Známe parametry M a B
  - Rozdíly: kdo spravuje cache?
    - Tedy který blok vyhodit z cache při načtení nového?
    - Algoritmus vs. systém / hardware
- Cache-oblivious algoritmy a DS
  [Frigo, Leiserson, Prokop, Ramachandran, 1999]
  - Nepotřebují znát M a B

    → fungují na různých architekturách
    a efektivně využívají celou paměťovou hierarchii





Předpoklady modelu
(které lze odstranit)

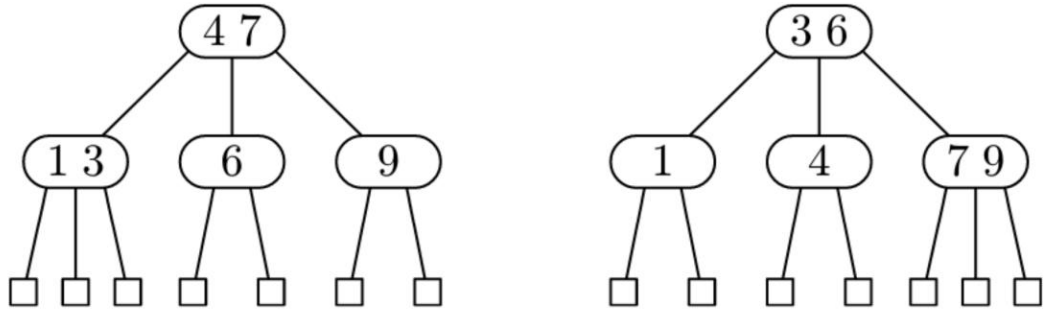


1. Cache řízena algoritmem (optimálně)
   - Algoritmus rozhoduje, která stránka je vyhozena z cache
   - Optimální algoritmus na správu cache:
     - *Longest Forward Distance (LFD)*: vyhodí blok, který bude potřeba nejpozději v budoucnu
   - Odstraníme příště pomocí *Least Recently Used* (LRU)
2. *Tall cache assumption:* $\frac{M}{B} \geq \Omega(B)$ … tedy # bloků je $\Omega(B)$
   - Není potřeba vždy
3. Plná asociativita: blok může být kdekoliv v cache
   - V reálu mají CPU cache omezenou asociativitu – např. 8-cestné (8-*way*)
     - Disk rozdělen do M clusterů, cache může obsahovat třeba jen 8 bloků z jednoho clusteru

Příklad cache-aware DS: (a, b)-stromy

- Dáno B, nastavíme $b = B/2$ a $a = b/2$

→ Uzel se vejde do bloku



Obrázek 8.7: Dva (2, 3)-stromy pro tutéž množinu klíčů

- I/O složitost vyhledávání klíče:

$O(\log_a(n) + 1) = O(\log_B(n) + 1)$ přenosů

- Existuje cache-oblivious varianta [Bender, Demaine & Farach-Colton: Cache-oblivious B-trees. FOCS'00]
- Cache-oblivious binární stromy: van Emde Boasovo uspořádání
  - $O(\log_B(n) + 1)$ přenosů na cestě do listu

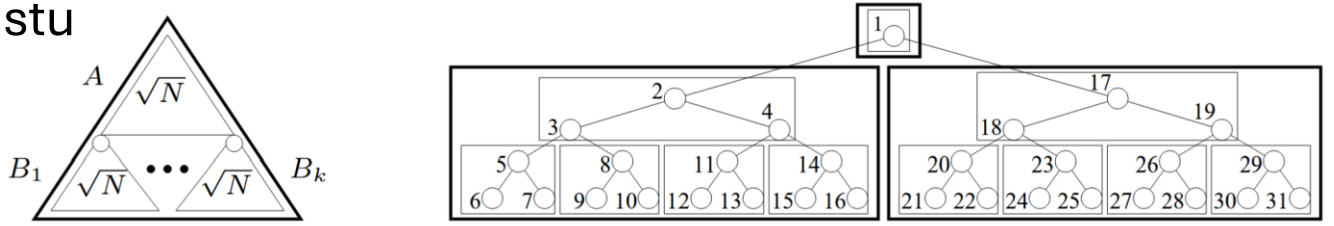


Figure 4. The van Emde Boas layout (left) in general and (right) of a tree of height 5.

Třídění: I/O analýza MergeSortu (třídění sléváním)

Algoritmus MERGESORT (rekurzivní třídění sléváním)
Vstup: Posloupnost $a_1, \ldots, a_n$ k setřídění

1. Pokud $n = 1$, vrátíme jako výsledek $b_1 = a_1$ a skončíme.
2. $x_1, \ldots, x_{\lfloor n/2 \rfloor} \leftarrow \text{MERGESORT}(a_1, \ldots, a_{\lfloor n/2 \rfloor})$
3. $y_1, \ldots, y_{\lceil n/2 \rceil} \leftarrow \text{MERGESORT}(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$
4. $b_1, \ldots, b_n \leftarrow \text{MERGE}(x_1, \ldots, x_{\lfloor n/2 \rfloor}; y_1, \ldots, y_{\lceil n/2 \rceil})$

Výstup: Setříděná posloupnost $b_1, \ldots, b_n$

- Časová složitost $O(N \cdot \log N + 1)$
- I/O složitost $O(\frac{N}{B} \cdot \log N + 1)$
- Zlepšení: k-cestný MergeSort pro $k = \frac{M}{B}$: $O(\frac{N}{B} \cdot \frac{\log N}{\log \frac{M}{B}} + 1)$

Zlepšení: k-cestný MergeSort

- pro $k = \frac{M}{B}$: I/O složitost $O\left(\frac{N}{B} \cdot \frac{\log N}{\log \frac{M}{B}} + 1\right)$
  - Optimální
  - … ale cache-aware
- Optimální cache-oblivious třídění: FunnelSort

Matice $N \times N$

- Uložena po řádcích

Příště

- Dokončení cache-oblivious transpozice
- Online algoritmy pro správu cache
  - Least Recently Used (LRU)
- Hešování ☺