2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163
- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

External memory model, I/O model

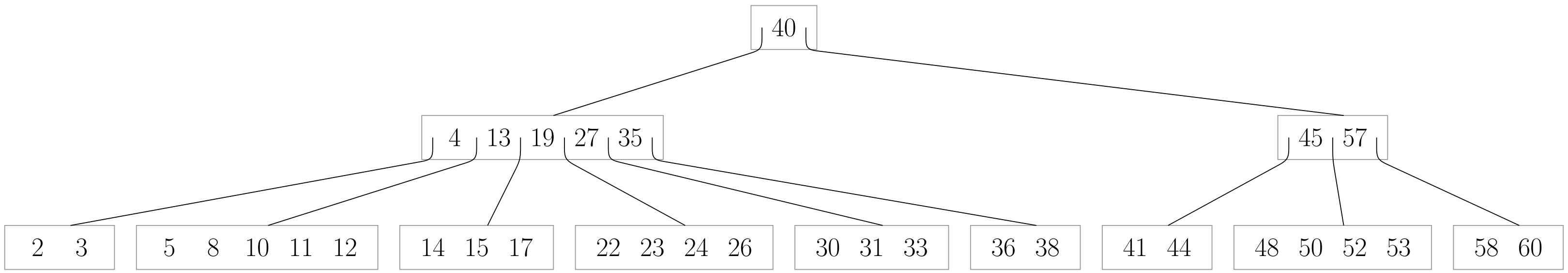
- big and slow disk, fast memory of a limited size M words
- disk reads/writes in blocks of size B words
- when analyzing algorithms, count how many blocks are read or written (**memory transfers**)

Example:

scanning through n elements: $O(\lceil n/B \rceil)$ transfers

B-tree

- parameter T
- node v has $v.n$ keys and 0 or $v.n + 1$ children
- keys in a node are sorted, each subtree contains only values between two successive keys in the parent
- all leaves are in the same depth
- for each node v except root satisfies $T - 1 \leq v.n \leq 2T - 1$
- in root, $1 \leq v.n \leq 2T - 1$



B-tree insert

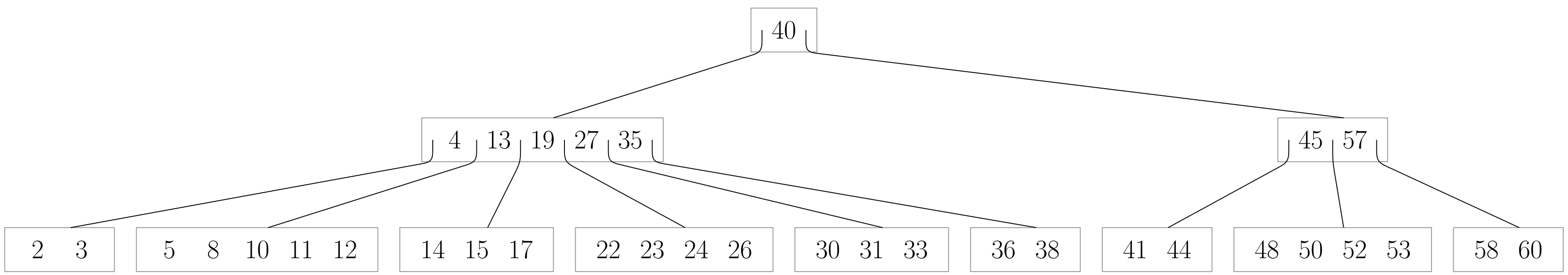
- Find the leaf where the new key belongs
- If leaf has $2T - 1$ keys:

  Split it into two leaves, each with $T - 1$ keys

  Insert original median to the parent (recursively)

  If the recursion reaches the root and root is full,

  create a new root with 2 children
- New element can be now inserted into its leaf



External MergeSort

- Create sorted runs of size M
- Repeatedly merge $M/B-1$ runs into one:

  read one block from each run, use one block for output

  when output block gets full, write it to disk

  when some input gets exhausted, read next block from the run (if any)
- Overall $\log_{M/B-1}(n/M)$ merging passes through data

External memory model: summary

- B-trees can do search tree operations (insert, delete, search/predecessor) in $O(\log_{B+1} n) = O(\log n / \log(B+1))$ memory transfers
- Sorting $O((n/B) \log_{M/B}(n/M))$ memory transfers

Cache oblivious model

- Algorithm in external memory model:

  – explicitly requests block transfers,

  – knows B,

  – controls memory allocation

- Algorithm in cache oblivious model:

  – does not know B or M,

  – algorithm requests reading/writing from disk,

  – automated caching,

  – memory M/B slots, each holding one block from disk

Cache operation

Algorithm requests reading/writing from disk

- cache M/B slots, each holding one block
- if the block containing request in cache, no transfer
- else replace one slot with block holding requested item, write original block if needed (1 or 2 transfers)
- which one to replace: classical on-line problem of paging

Paging

- Cache with k slots, each holds one page
- Sequence of page requests
- If requested page not in cache, bring it in and replace some other page (**page fault**)
- **Goal:** minimize the number of page faults
- **Offline optimum:**

  At a page fault remove the page that will not be used longest
- Example of an **on-line algorithm: FIFO**

  At a page fault remove the page that is longest time in cache

Paging

- On-line paging algorithm is **k-competitive**, if it always uses at most $k \cdot$ OPT page faults, where OPT is off-line optimum for the same input
- On-line algorithm is **conservative** if in a segment of requests containing at most k distinct pages, it needs $\leq k$ page faults
- Each conservative algorithm is k-competitive
- FIFO is conservative and thus also k-competitive
- No deterministic algorithm can be better than k-competitive
- Conservative paging algorithm on memory with k slots uses at most $k/(k-h)\mathrm{OPT}_h$ page faults where OPT_h is the off-line optimum for h slots ($h \leq k$)
- In fact it is possible to prove ratio $k/(k-h+1)$

Cache oblivious model

- Algorithm in cache oblivious model:

  – does not know B or M,

  – algorithm requests reading/writing from disk,

  – automated caching,

  – memory M/B slots, each holding one block from disk,

  – assumption: paging done by offline optimum,

  – usually asymptotially equivalent to FIFO on memory 2M

- Advantages of cache oblivious algorithms:

  – no need to know B

  – may adapt to changing M or B

  – also good for memory hierachy (multiple caches, disk, network)

  – often the same complexity as in external memory model

Cache oblivious model: results

- Scanning $O(\lceil N/B \rceil)$ as in I/O model
- Searching $O(\log_{B+1} N)$ as in I/O model
- Sorting $O((n/B) \log_{M/B}(n/M))$ as in I/O model but requires $M = \Omega(B^{1+\epsilon})$

Static cache-oblivious search trees, Prokop 1999

- Perfectly balanced binary search tree with nodes stored on disk in van Emde Boas order

- Search by the usual method, $O(\log_{B+1} n)$ block transfers (the same as B-trees for known B)

- For comparison: how many transfers needed for binary search?
  What about tree with nodes in pre-order or level-order?
  How to store the tree if we know B?

van Emde Boas order

- Split tree of height lg n into top and bottom, each of height $\frac{1}{2}$ lg n
- Top: a small tree with about $\sqrt{n}$ nodes
- Bottom: about $\sqrt{n}$ small trees, each about $\sqrt{n}$ nodes
- Print each of these small trees recursively, concatenate results

Example: A tree with 4 levels is split into 5 trees with 2 levels.

Resulting ordering:

```
              1
       2               3
   4       7       10      13
5   6   8   9  11  12  14  15
```

Ordered file maintenance

- Maintain n items in an array of size $O(n)$ with gaps of size $O(1)$
- Updates: delete item, insert item after a given item
  (similar to linked list)
- Update rewrites interval of size $O(\log^2 n)$ amortized, in $O(1)$ scans
- Done by keeping appropriate density in a hierarchy of intervals
- We will not cover details

Dynamic cache oblivious trees

- Keep elements in ordered file in a sorted order
- Build a full binary tree on top of array (segment tree)
- Each node stores maximum in its subtree
- Tree stored in vEB order
- When array gets too full, double the size, rebuild everything

Search: check max in left child and decide to move left or right.

Follows a path from root, uses $O(\log_B n)$ transfers

Update: search to find the leaf, update ordered file,

then update all ancestors of changed values by postorder traversal

Improvement of update time by bucketing