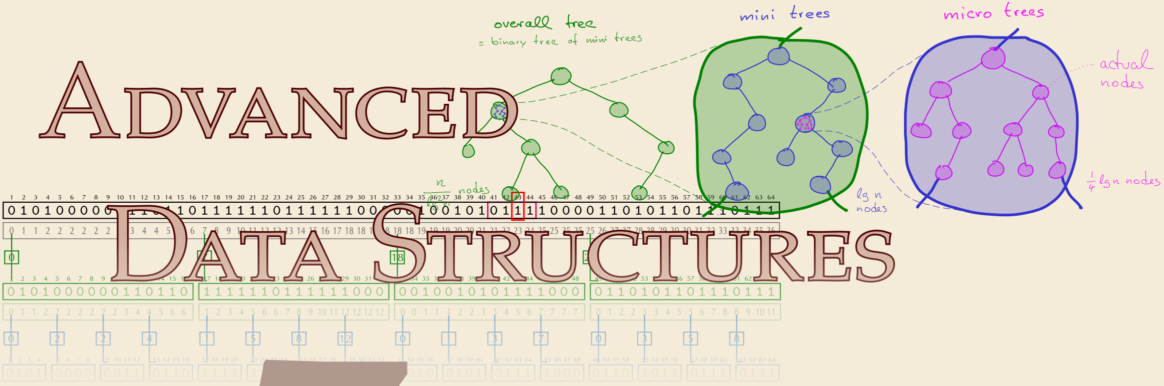
Advanced Data Structures



7 External Memory

Advanced Data Structures · *Summer 2026*

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg

version `2026-07-08 20:38 H`

Outline

7 External Memory

7.1 External Memory Basics

The External-Memory Model (EMM)

`https://tiny.cc/latency`

Notation

Two-level I/O model of Aggarwal and Vitter

Aggarwal, Vitter: *The Input/Output Complexity of Sorting and Related Problems*, CACM **1988**

- Memory is split into a fast **internal memory** (cache) and a slow, unbounded **external memory** (disk).
- Data is transferred between them in fixed-size **blocks**
- N = #elements in the problem instance
- M = #elements that fit in internal memory
- B = #elements per block, with $1 \le B \le M$.
- **Abbreviations:** $n = \frac{N}{B}$ (input size in blocks) $m = \frac{M}{B}$ (cache size in blocks)

Cost model

- An **I/O** (block transfer) moves one block between disk and internal memory.
- **cost** of algorithm = number of I/Os (computation in internal memory is free)

EMM – Complexities

External-memory algorithms are a well-studied area.

For most standard problems, the I/O complexity is known, and practical algorithms exist.

Problem	**I/O complexity**
Scanning N contiguous elements	$\mathrm{scan}(N) \;=\; \Theta\left(\dfrac{N}{B}\right) \;=\; \Theta(n)$
Sorting N elements	$\mathrm{sort}(N) \;=\; \Theta\left(\dfrac{N}{B}\log_{M/B}\dfrac{N}{B}\right) \;=\; \Theta(n\log_m n)$
Searching among N keys	$\Theta(\log_B N)$ per query
Permuting N elements	$\Theta(\min\{N,\ \mathrm{sort}(N)\})$

External Sorting – Upper Bound

Sorting in $O(n \log_m n)$ I/Os can be done with multiway mergesort.

External Mergesort

1. **Run formation:** Read M elements, sorting internally, write back.
2. **Paged Merging:** Merge $k = \frac{M}{B} - 1 = \Theta(m)$ runs at a time.
   - Hold one block per input run and one output block in internal memory
   - Repeatedly move smallest element from a run to output
   - Flush output buffer when block full; refill input buffer when run's block exhausted

Analysis

- Run formation uses $O(n)$ I/Os

$\rightsquigarrow$ Start with N/M runs

- Number of passes: $\log_k\left(\frac{N}{M}\right) = \log_{M/B}\left(\frac{N}{B}\frac{B}{M}\right) = \log_{M/B}\left(\frac{N}{B}\right) - 1 = \Theta(\log_m(n))$
- Each pass costs $O(n)$ I/Os

7.2 Basic EMM Data Structures

B-Trees

- idea of B-trees long pre-dates EMM
- but of course, the goal is the same: *make better use of entire loaded block!*

- choose parameter so that node of B-tree fits into one block size B
  - ⇝ fanout of all nodes in $\Theta(B)$

⇝ tree search costs $O(1 + \log_B(N))$ I/Os (= height of B-tree)

⇝ by linking in order (and maybe using B^+-trees) can retrieve k elements in sorted order with $O(1 + k/B)$ I/Os (after search)

- search cost $O(1 + \log_B(N/M))$ for a **single search** is I/Os optimal for data structures with $O(N/B)$ blocks (even without updates)
- B-trees support updates at the same cost

If that's basically optimal, what to hope for?

Lists

For simple queries, we may have ***less than 1*** *single I/O per query (amortized)!*

- For very simple data structures easy to do!
  - **Queue:** Store one block at beginning and end in internal memory
  - read/write next block when exhausted $\rightsquigarrow$ only 1 I/O for every B enqueue/dequeue ops
  - $\rightsquigarrow$ $O(1/B)$ I/Os amortized cost
  - **Stack:** Almost same, but need to cache 2 blocks at top
- *How about general linked lists?*
  - inserting in the middle when block not in memory must cost 1 I/O
  - but if we have block already in memory (e. g., for batch inserts), easy! $\rightsquigarrow$ $O(1/B)$ I/Os amortized?
  - . . . unless the block is full; then we need to insert a new block (update pointers in pred/succ block)
  - **Problem:** Can't leave new block with single element!
  - ϟ Deleting same element again would break amortization

Blocked Linked List

Blocked Linked List

- Invariant: Block has $\geq B/4$ elements.
- When inserting into a full block, we evenly split elements among two blocks

$\rightsquigarrow$ both blocks roughly $B/2$ elements

$\rightsquigarrow$ $\Theta(B)$ further insertions before we split here again.

- Delete needs to handle too empty blocks
  - otherwise can't scan list in $O(N/B)$ I/Os and waste space

$\rightsquigarrow$ if block has less than $B/4$ elements, fuse with neighbor block

- if together with neighbor more than $3B/4$ elements, instead evenly distribute elements

$\rightsquigarrow$ $\Theta(B)$ operations before blocks reach capacity constraints again.

$\rightsquigarrow$ Restructuring cost is $O(1/B)$ I/Os amortized.

Pointers & Handles

- typical use case of linked lists in internal memory maintains ***pointers*** to nodes
  - *referential integrity / stable pointers often main selling point of linked lists!*
  - ↯ with rearrangement for compact space usage not possible in external memory!
- **Option 1:** *Backpointers*
  - when outside has pointer to element in a list node, list node stores location of pointer
  - ⇝ can update outside pointer when element gets moved
  - uopn node split/fuse, may need to change $O(B)$ pointers
  - ⇝ only get $O(1)$ I/Os amortized for updates
    (cannot guarantee amortized $O(1/B)$ I/Os even if list node in cache)
  - If we have many pointers to same element, expensive to update all!
- **Option 2:** *Handles*
  - External pointers point to *handle* object
  - Handle forwards to location of element inside data structure
  - When element gets moved, only handle needs to be updated
  - ⇝ Same complexity as if we had only $O(1)$ pointers

7.3 Buffer Trees

Search Trees vs Sorting in EMM

- Note: in internal memory, for comparison-based algorithms and up to constant factors
  Dynamic sorted ***dictionaries*** *are equally efficient as offline* ***sorting***
  (Assuming dynamic optimality, maybe even in an instance-optimal sense!)
- *This is very much not true in external memory:*

  Sorting $\Theta\left(\frac{N}{\boldsymbol{B}} \log_{\boldsymbol{M/B}}\left(\frac{N}{B}\right)\right)$ $\ll$ B-Tree inserts $\Theta\left(N \log_{\boldsymbol{B}}\left(\frac{N}{B}\right)\right)$
- B is usually not extremely large, but M may be (not an artifact of specific data structures; lower bounds!)
- *It turns out, it's not the operations themselves, but not knowing them in advance that costs*

Buffer Trees

- conceptually a B^+-tree with "supernodes" with fanout $\Theta(m)$

$\leadsto$ holding $\Theta(M/B)$ keys and child pointers

- in addition, each node stores a **buffer** for $\Theta(M)$ pending operations
  - each is a triple of $\langle key, operation\ type, timestamp\rangle$
  - operations can be any sorted dictionary operations.

Theorem 7.1 (Amortized cost of buffer trees)

An (offline) sequence of N operations (insertions, deletions, and queries) on an initially empty buffer tree can be performed in a total of $O\big(n \log_m n\big)$ I/Os.
If the sequence includes range queries that report elements, the total cost is $O\big(n \log_m n + \frac{Z}{B}\big)$ I/Os, where Z is the total number of elements reported across the entire sequence. ◀

$\leadsto$ in particular, optimal sorting (insert all, read out in sequential order)

Buffer-Tree Priority Queues

Theorem 7.2 (Buffer-Tree Priority Queues Amortized Cost)

A Buffer-Tree Priority Queues answers a sequence of N `Insert` and `DeleteMin` operations with amortized cost of $O(\frac{1}{B} \log_m n)$ I/Os. ◀

7.4 Cache-Oblivious Data Structures

Model

- like EMM, but algorithm cannot know/use B and M
  - We use them in the *analysis*, but not to design/tune the algorithm
  - ⇝ forces algorithm to (simultaneously) adapt to any parameter choices(!)
- Goal: Match I/O lower bound **simultaneously** for **all** values of M and B
- *Why should even possible!?*
- Scanning N elements sequentially is CO-optimal!

- **Fine print**
  - assume an *ideal cache*:
  - fully associative cache
  - **optimal offline** block replacement policy
  - **Note:** given $2M$ memory, LRU is 2-competitive to offline OPT

    Sleator, Tarjan: *Amortized efficiency of list update and paging rules*, CACM **1985**
  - ⇝ Optimal replacement policy may be replaced by realistic choice

CO & Multi-level hierarchies

Informal theorem: If memory is inclusive and uses consistent paging policy across levels, a CO-optimal algorithm is an optimal algorithm also for multi-level memory hierarchy.

7.5 vEB-Layout

Static Searching

If B-trees are out of the question, how can we do (binary) search?

- even not obvious for a **static** sorted set of elements

⇝ Simplest nontrivial problem in cache-oblivious external memory

van Emde Boas (vEB) layout

- A clever recursive order for array entries entirely solves this problem(!)

vEB Analysis

7.6 File Maintenance

Dynamic vEB Layout?

- To allow insertions and deletions, cannot keep elements in rigid array

Cache-Oblivious B-Trees

1. store data in sorted order and separately a search tree as index on top of the data
2. **Idea 1:** Store data in sorted order in a ***Packed Memory Array (PMA)***
   - keeps N elements in array of size $O(N)$ and in sorted order (but with gaps)
   - insertions are allowed to move old elements
   - next section: can do this while moving $O(\log^2 N)$ elements amortized(!)
3. **Idea 2:** apply vEB layout to **nodes of a balanced search tree** index into PMA

$\leadsto$ B-tree style operations cache-obliviously in $O(\log_B(N) + \frac{1}{B}\log^2(N))$ I/Os

Packed-Memory Array

- Imagine range tree on top of array of size $c \cdot N$
- each range has a maximal density (stricter for smaller intervals)
- by uniformly spreading elements out at first non-over-dense range suffices
- (amortized analysis shows $O(\log^2(N))$ elements moved per insertion)

Advanced Data Structures in our group

You've seen several research results from our group!

- Jumplists
- Lazy Search Trees
- Lazy Partition Heaps
- Hypersuccinct Trees (Tree Covering)
- Graphs via *compressed* wavelet trees

Maybe your future paper enters that list some day 😉