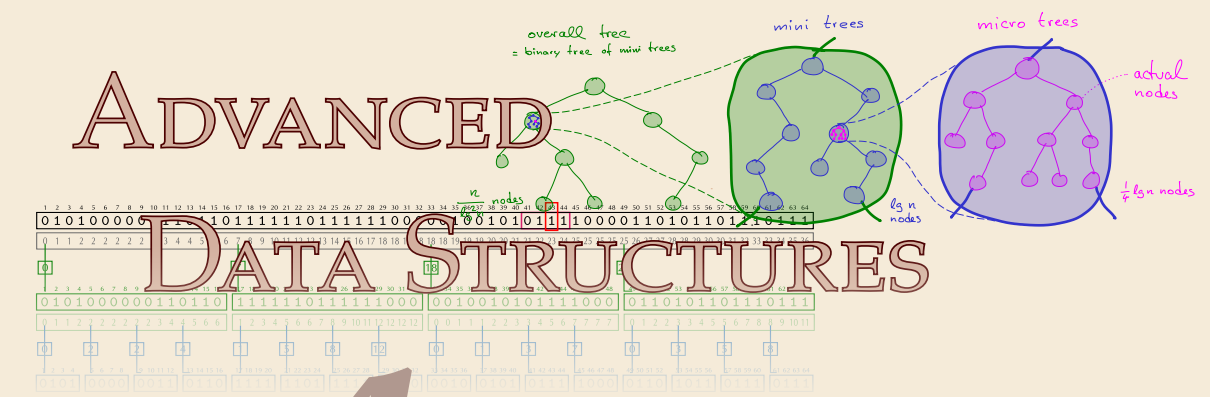

Advanced Data Structures

4 Persistence

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version `2026-06-09 22:55 H`

Outline

4 Persistence

4.1 Time Travel!

Ephemeral Data Structures

Standard mode for data structures: destructive updates.

- Changes to the data structure destroy previous version

$\rightsquigarrow$ queries and updates are always w.r.t. latest version

Persistent data structures have non-destructive updates

$\rightsquigarrow$ Can still access past versions

- useful for auditing
- allows clever algorithms (see below)
- natural mode of operation for functional programming

- **Partial persistence:** queries to any version, updates only to most recent
  - version history still a path $\rightsquigarrow$ number versions
- **Full persistence:** both queries and updates to any version
  - versions form a tree
- more general versions thinkable, but unclear whether efficiently supportable (not here)
  - confluent persistence, functional persistence
  - retroactive data structures (time travel with single timeline . . .)

4.2 Application: Planar Point Location

Planar Point Location

Before diving into implementing persistence, let's see an application

4.3 Elementary ideas

Simple tricks

Full and Partial Persistence is trivial . . . if you don't care about time and space

- every update copies entire data structure
- keep a dictionary of versions

How (much) can we do better?

To make discussion concrete, let's try to build
Partially Persistent Doubly Linked Lists and **Partially Persistent BSTs**

- Upon one update *operation*, we may have a sequence of update *steps*:
  - update the value stored in a node
  - update pred/next pointer resp. left/right child pointer of a node
  - allocate a new node

⇝ DLL and BST have few update steps per update operation

- copying entire data structure very wasteful

Attempt 1: Copy-On-Write Updates

*Treat all fields in a node as **immutable**.*

Upon update step at node x:

1. Create a copy x' of x with the new field value
2. For every other node y with a pointer to x, $y.p == x$, recursively trigger an atomic update $y.p := x'$
   - We assume here that these predecessors y of x are known
   - For DLLs trivial as always pointer in both directions exist
   - For BSTs, we can remember the predecessors during traversals

For header/root nodes, maintain sorted dictionary of *versions*

This effectively copies path from root to changed node $\rightsquigarrow$ *path copying*

Copy-on-Write – Discussion

- 👍 CoW directly supports full persistence
- 👍 For queries, only slowdown comes from locating the root of the desired version $O(\log m)$ in addition to original time after m updates.
- 👍 For low-depth, acyclic data structures, space overhead moderate
- 👎 DLLs copies all
  - ⚡ worst case $\Omega(nm)$ for ephemeral data structure with n nodes after m updates

Attempt 2: Log all Updates

Full replay log of all updates

- always store original data structure and full log of updates
- for every read and update, replay updates from beginning

$O(n + m)$ space (assuming we can encode update in $O(1)$ space)

$\Omega(i \cdot T_u + T)$ time for operation in version i
T_u time per ephemeral update, T time for last operation

Attempt 3: Hybrid of CoW and Log

- store snapshot for every kth version
- update log in between

$\rightsquigarrow$ recompute version i from previous snapshot + log entries


tune-able trade-off between space and time


worst case still $\Omega(\sqrt{m})$ blow-up in either space or query time

4.4 Partial Success: Fat Nodes

Pointer-Based Data Structures

We make some assumption about the data structure (generic version)

- Upon one update *operation*, we may have a sequence of update *steps*:
  - allocate a new nodem
  - update an *information field* of a node, or
  - update a *pointer field* of a node

$\rightsquigarrow$ encompasses most data structures without arrays

Fat Nodes

Within each node and for each field: keep a log of writes, sorted by version stamp

- log stored as BBST
- when reading a field: find latest update before current version $\rightsquigarrow$ $O(\log m)$ time
- writing field: add log entry $\rightsquigarrow$ $O(\log m)$ time


$O(1)$ extra space per update step


$O(\log m)$-*factor* slow-down for operations

Can be better if guaranteed to see few updates to fixed field! (e.g., unbalanced BST)

4.5 Bounded Degree Pointer Machine

Terminology

Bounded In-Degree

4.6 Partial Persistence

General Transformation: Node Copying

Theorem 4.1 (Node copying transformation)

For any (ephemeral) bounded-indegree linked data structure, there is a **partially persistent** data structure that has amortized space overhead of $O(1)$ per update step and a $O(1)$ factor slowdown for all operations. ◀

Driscoll, Sarnak, Sleator, Tarjan: *Making data structures persistent,* JCSS **1989**

Full Persistence

Technique can be extended do full persistence at same costs(!)

Driscoll, Sarnak, Sleator, Tarjan: *Making data structures persistent,* JCSS **1989**

Construction becomes more complicated

- Versions are now nodes in a version tree, not just numbers
- ⇝ Linearize the tree using an Euler tour around the tree
- any persistent node need to be kept live
- ⇝ cannot pack old nodes full of update logs
- ⇝ more complicated *node splitting* instead of node copying.

End of History for Partial Persistence?

Despite good properties of general transformation,
room for improvement on specific data structures.

- For balanced binary search trees, overhead can be reduced
- For external-memory data structures, space usage can be improved

Becker, Gschwind, Ohler, Seeger, Widmayer: *An asymptotically optimal multiversion B-tree,* VLDB J **1996**

Brodal, Sioutas, Tsakalidis, Tsichlas: *Fully persistent B-trees,* TCS **2020**

Time Travel Kills Amortization

Note that (partial) persistence transformation works fine for amortized data structures however, guarantees may not be strong enough!

- Suppose we are at a state in the data structure, where there is a certain query
- In a partially persistent data structure, such a bad state is preserved

$\rightsquigarrow$ Could repeatedly query this version and exhibit high running time