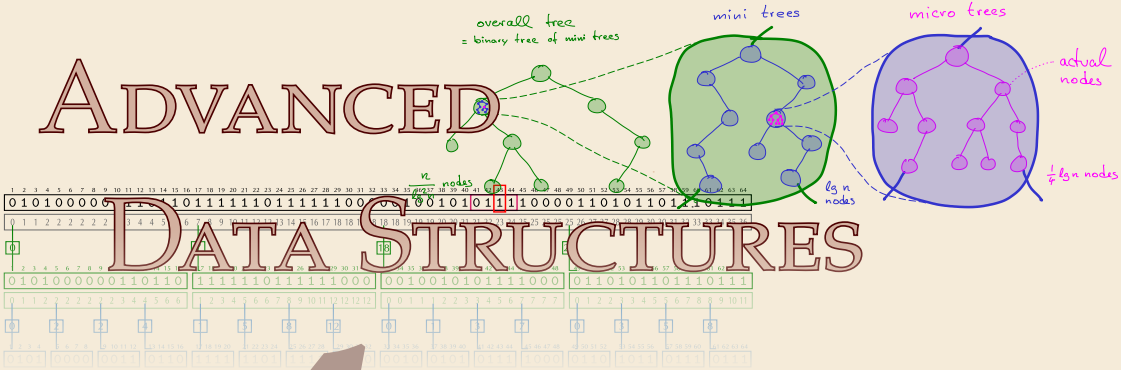


Advanced Data Structures

4 Persistence

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version **2026-06-08 15:47**

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

⇝ queries and updates are always w.r.t. latest version

Ephemeral Data Structures

Standard mode for data structures: destructive updates.

- Changes to the data structure destroy previous version
- ⇝ queries and updates are always w.r.t. latest version

Persistent data structures have non-destructive updates

- ⇝ Can still access past versions
  - useful for auditing
  - allows clever algorithms (see below)
  - natural mode of operation for functional programming

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

Ephemeral Data Structures

Standard mode for data structures: destructive updates.

- Changes to the data structure destroy previous version
- ⇝ queries and updates are always w.r.t. latest version

Persistent data structures have non-destructive updates

- ⇝ 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 ⇝ number versions
- **Full persistence:** both queries and updates to any version
  - versions form a tree

Ephemeral Data Structures

Standard mode for data structures: destructive updates.

- Changes to the data structure destroy previous version
- ⇝ queries and updates are always w.r.t. latest version

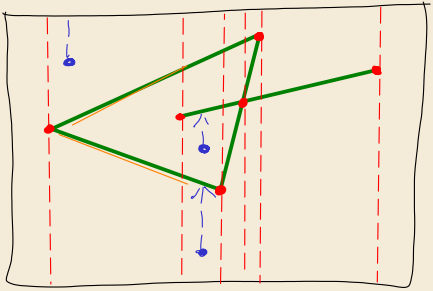
Persistent data structures have non-destructive updates

- ⇝ 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 ⇝ 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

2D Planar Point Location

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

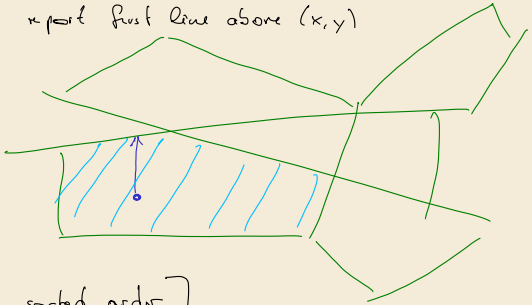


set of n line segments, only intersect at endpoints

preprocess that into data structure

at query given point (x,y)

report first line above (x,y)



within one vertical slab

lines are sorted

$\Rightarrow$ for every vertical prepare line segments in sorted order (by y-coordinate)

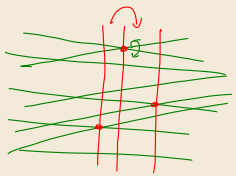
sorted array of slabs by x-boundaries

finding point: ① use x to find slab $O(\log n)$

② within slab, do binary search for y $O(\log n)$

/

comparing line at x with y



space usage

$\Theta(n)$ per slab

$\Theta(n)$ slabs

$\Theta(n^2)$ overall

preprocessing time $\Theta(n^2 \log n)$

neighboring slabs don't differ much!

actually: total number of changes between slabs $= O(n)$ (# endpoints!)

---

① Preprocessing step: Sweepline algorithm

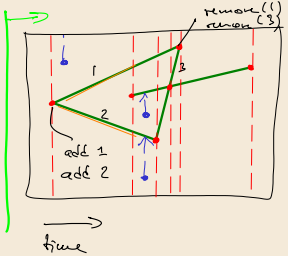
Maintain BST sorted by y-direction

persistent

with inserts / deletes at •

$\Rightarrow$ also store sorted list of x-values w/ pointers

$O(n \log n)$ total time



(2) Query: ∘ v = find version w/ x $O(\log n)$

∘ query successor of y at version v $O(\log n)$

4.3 Elementary ideas

Terminology

version for partial persistence = #past updates $\in \mathbb{N}$

ephemeral data structure = "normal" data structure

persistent data structure = partial persistent d.s.

↪ wish: general transformation

access operation
update operation
] methods of ephemeral d.s.

consist of sequences of access steps

sequence of (access steps, one update step)

total time = # steps

update time = # update steps

Simple tricks

Full and Partial Persistence is trivial

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

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?

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*

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

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

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

Attempt 3: Hybrid of CoW and Log

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

⇝ recompute version i from previous snapshot + log entries

m = # update operations

👍 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)

unbalanced BSTs w/ insert only

4.5 Bounded Degree Pointer Machine

Terminology

linked data structures only

- nodes (single type)
- pointers between them
- O(1) access pointers point to entry nodes

---

access step:

read info field in known node

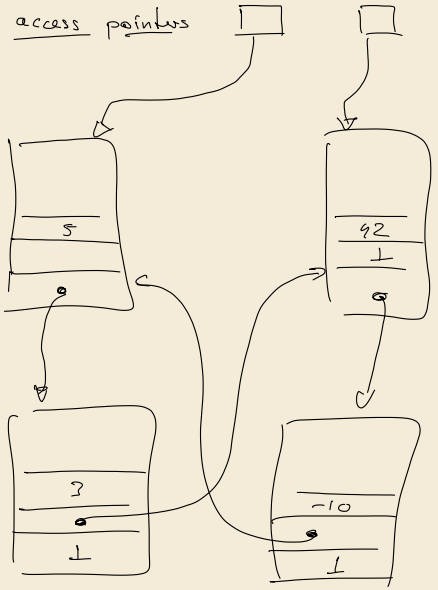
pointer field — 1 —

follow pointer

maintain set of accessed nodes

initially only entry nodes

add more via following pointers



update step

- new node
- write info field
- with pointer field to accessed node

everything allowed except arrays

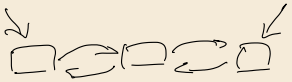
Bounded In-Degree

linked data structure w/ d pointer fields per node

Every node is target of $\leq p$ pointers in d.s.

BST: $p = 1$

DLL: $p = 2$



Notation

n = #nodes in current version

v version number

m = #update oper. = #versions

T, T_q, T_u time for query/update

x ephemeral node

$\overline{x}$ persistent node

$\mathcal{F}(x)$ persistent nodes for x

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**

Examples: BBST
but guarantees not so good
if many update steps (Splay! ?)

DLL ✓ Fib/Quake heaps ↯ array

LP Heaps ✓

Intuition: between CoW & fat nodes

Concretely: ephemeral node x w/ in-degree $\leq p$ and out-degree $= d$

$\Rightarrow$ persistent node $\bar{x}$ w/ $p + d + e + 1$ update box

ordinary pointers suffice

f.b.d.

"extra pointers"

$(e \leq p)$

< version stamp, field name, value >

upon access step to x, read through update boxes to check for new values

upon update step: try to use update box

$\mathcal{F}(x)$: "family" of persistent nodes for x

+1 above: store $\mathcal{F}(x)$ as linked list using next pointer

last/most recent $\overline{x}$ called live, others dead

live pointer = pointer in newest version

+p from above: store back pointers (not needed for dead pointers)

access operation: for each access pointer, maintain an array of version $\mapsto$ node mappings $O(1)$ from version to entry node

Invariant: If x points to y in a version v, either in field or update box

$\overline{x}$ reached from entry for version v has entry for pointer to $\overline{y}$

Update Operations

v = current version

any update step is resolved via update boxes if possible, otherwise needs to copy $\bar{x}$

copies can cascade as their change predecessors!

=> maintain set copied nodes S (initially empty) until end of update operation

① Update step:

- new node => new persistent node w/ time stamp v
- $x.f := \alpha$ (info field) do not use update boxes!

  if $\bar{x}.v == v$ update $\bar{x}.f := \alpha$

  otherwise create new persistent copy $c(\bar{x})$
  (w/ new field value and timestamp v)

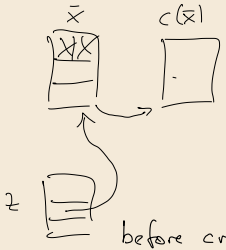
  add $c(\bar{x})$ to S

- $x.p := y$ (pointer field)

  if $\bar{x}.v == v$ update $\bar{x}.p := \bar{y}$

  otherwise if $\bar{x}$ has free update box

  also maintain backward pointers here



add $\langle v, p, \bar{y} \rangle$ into update bar

else allocate copy $c(\bar{x})$

w/ version v and new pointer value

add $c(\bar{x})$ to S

before creating copy, check if $c(\bar{x})$ already exists and if so, use that

(2) Fix predecessor pointers (after <u>all</u> update steps of operation done)

While $S \neq \emptyset$

pick $c(\bar{x}) \in S$

for $\bar{z} \in$ predecessors$(c(\bar{x}))$ // backward pointers

update step $\bar{z}.p := c(\bar{x})$ (if previously $\bar{z}.p == \bar{x}$) as above

(might create new copies in S)

Analysis

We will focus on extra space for update operations

$e \geq p$

amortized space cost = #new nodes + $\Delta\Phi$

Φ = #used/filled update boxes in live nodes

Recall that operation may have, say, n update steps

Recall we keep new copies S

during n update steps $t \leq n$ new copies ($= |S_0|$)

In postprocessing (②), we may get k extra copies

$\Rightarrow$ actual cost $t+k$

For each copy in ②, full node dies, and a new node is created, but empty ($\Delta\Phi = -e$)

$\Delta\Phi \leq -k \cdot e$

Overall remove $t+k$ nodes from S triggers $\leq p$ update steps (update pred. pointers)

$\Delta\Phi \leq (t+k) \cdot p - k$

$$\begin{aligned}
\Delta\Phi &\leq -e\cdot k + p(t+k) - k \\
&= k\,(p - (e+1)) + p\cdot t \qquad (e \geq p) \\
&\leq -k + p\cdot t
\end{aligned}$$

total amortized space $\quad t+k + -k + p\cdot t$

$$= O(n)$$

□

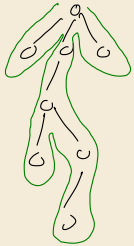
Full Persistence

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

⇝ Could repeatedly query this version and exhibit high running time