

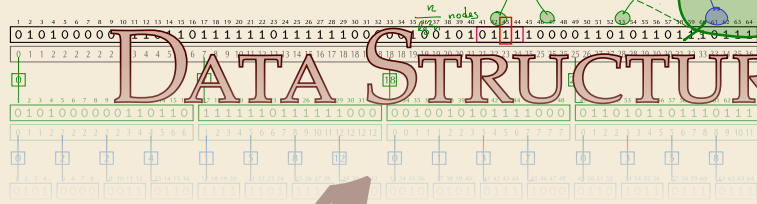
ADVANCED

overall tree
= binary tree of mini trees

mini trees

micro trees

actual nodes



DATA STRUCTURES

4

Persistence

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

4 Persistence

- 4.1 Time Travel!
- 4.2 Application: Planar Point Location
- 4.3 Elementary ideas
- 4.4 Partial Success: Fat Nodes
- 4.5 Bounded Degree Pointer Machine
- 4.6 Partial 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
- ↪ 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

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

} sit

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

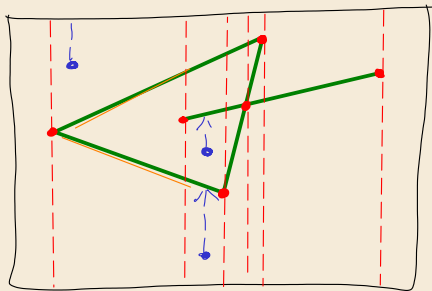
- ↪ Can still access past versions
 - ▶ useful for auditing
 - ▶ allows clever algorithms (see below)
 - ▶ natural mode of operation for functional programming
- ▶ **Partial persistence:** queries to any version, updates only to most recent
 - ▶ version history still a path ↪ number versions
- ▶ **Full persistence:** both queries and updates to any version
 - ▶ versions form a tree
- ▶ not here: more general versions thinkable, but unclear whether efficiently supportable
 - ▶ 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

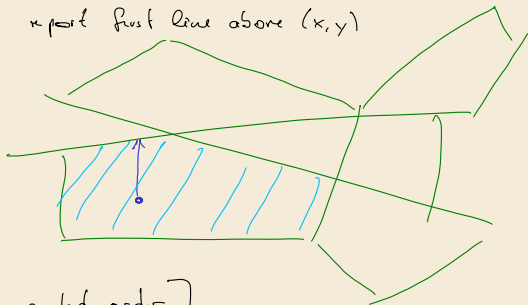


set of n line segments, only intersect at endpoints

prepare that into data structure

at query given point (x, y)

report first line above (x, y)



within one vertical slab

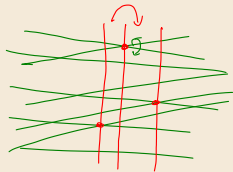
lines are sorted

⇒ 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

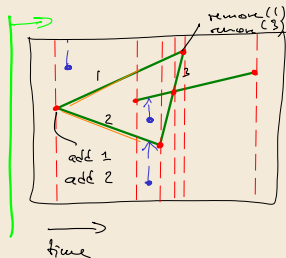
Maintain BST sorted by y -direction

↑
persistent

with inserts/deletes at \bullet

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

$O(n \log n)$
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:

- 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 k th version
 - ▶ update log in between
- ↪ recompute version i from previous snapshot + log


Attempt 3: Hybrid of CoW and Log


- ▶ store snapshot for every k th version

- ▶ update log in between

↪ recompute version i from previous snapshot + log

$$m = \# \text{update operations}$$

 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

Recall assumption on data structure (generic version)


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

↪ 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!

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

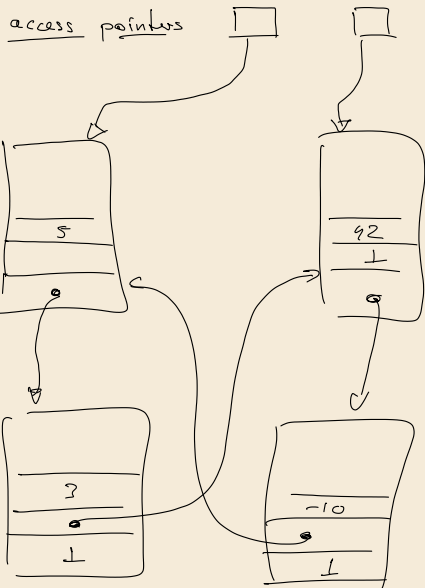
access step:

read info field in known node
pointer field ---

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

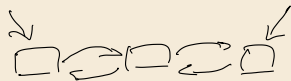
Bounded In-Degree

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

\bar{x} persistent node

$F(x)$ persistent nodes
for x

4.6 Partial Persistence

General Transformation: Node Copying

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

Theorem: Given an ephemeral bounded-indegree linked data structure, we can construct partially persistent data structure with $O(1)$ extra space per update step and $O(1)$ slowdown for all operations.

Intuition: between CoW & fat nodes

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

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

ordinary pointers
suffix

t.b.d.

"extra pointers"

$(e \leq p)$

\langle version stamp, field name, value \rangle

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

upon update step, try to use update box

$F(x)$: "family" of persistent nodes for x

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

last/most recent \bar{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 \rightarrow 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
 \bar{x} reached from entry for version v has entry for pointer to \bar{y}

Update Operations

any update step is resolved via update boxes if possible, otherwise needs to copy \bar{x}
copies can cascade as they change predecessors;

\Rightarrow maintain set copied nodes S (initially empty) until end of update operation