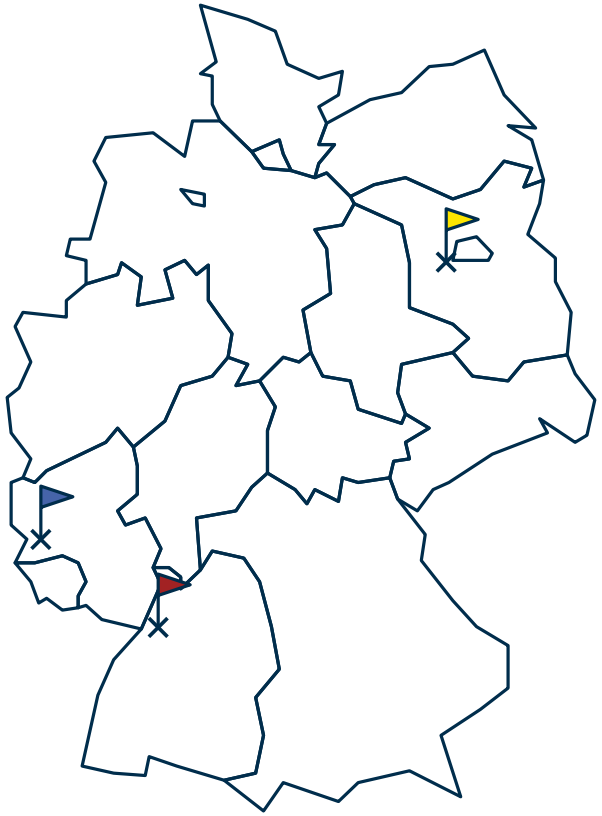
Computational Geometry

Point Location & Persistence – Where am I? And When?

Thomas Bläsius

Where am I?

Problem: Point Location
In which face of a geometric graph does a given point p lie?



Static Variant

- the graph G does not change
- answer queries for many points $p \in \mathbb{R}^2$
- develop data structure for G, such that
  - every query is fast
  - data structure can be build efficiently
  - data structure requires little space

What are possible applications?

Data Structures And Time Travel

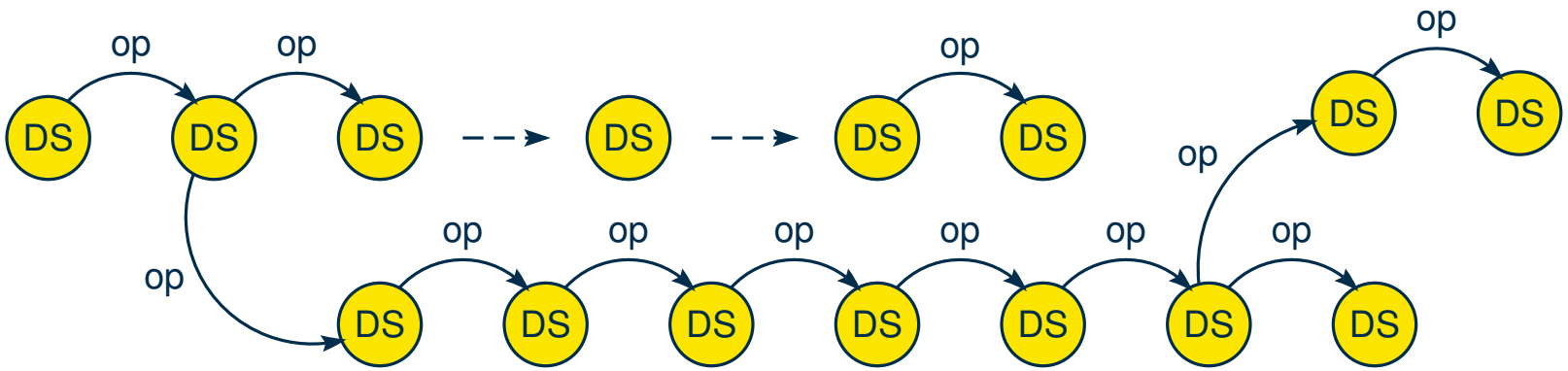
What Can We Typically Do With A Data Structure?

- queries to the data structure in its current state
- applying an operation $\rightarrow$ new state

Example: Priority Queue

get-min

insert, delete-min



Partial Persistence

- remember previous states of the data structure
- allow queries to arbitrary points in time

Full Persistence

- also allow operations in the past
- time is then no longer linear but branches

Pointer Machine Data Structures

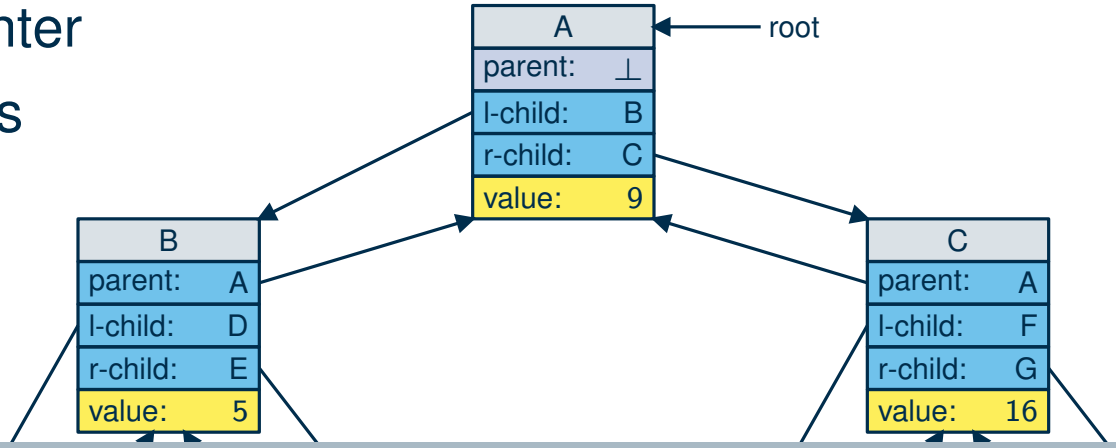
The Pointer Machine Model

- constant number of different node types, each with a constant number of fields
- each field contains one of the following
  - a data element (e.g., a number)
  - a pointer to another node or the NULL pointer
- a constant number of pointers to entry nodes

So Essentially

- a directed graph with constant out-degree
- and constant memory per node

Example: Binary Search Tree



Theorem **(partial persistence for everyone)**
Every data structure in the pointer machine model with constant in-degree can be made partially persistent with (amortized) constant overhead.

Queries, Operations, and Timestamps

Observation

- we don't know what queries and operations the DS has
- but: each is a sequence of atomic queries/operations
- $O(1)$ slow-down for each atomic query/operation $\Rightarrow$ $O(1)$ slow-down overall

Atomic Query: read field of the current node or follow a pointer to another node

Atomic Operation: change a field of the current node (data or pointer)

Timestamps

- is incremented after each complete operation (not after queries)
- entry nodes before: constant number of pointers to entry nodes
- now: for every timestamp a constant number of pointers to entry nodes

(e.g., constant number of arrays with one pointer per timestamp)

Nodes With A History

Atomic Operation

- idea: every node stores its own diff in an additional “mod” field
- instead of applying atomic operation: store (timestamp, field, new value) in mod

Atomic Query: take mod field into account

- example query at time t: C.value
  - yields 16, if $t < 28$
  - yields 1, if $t \geq 28$

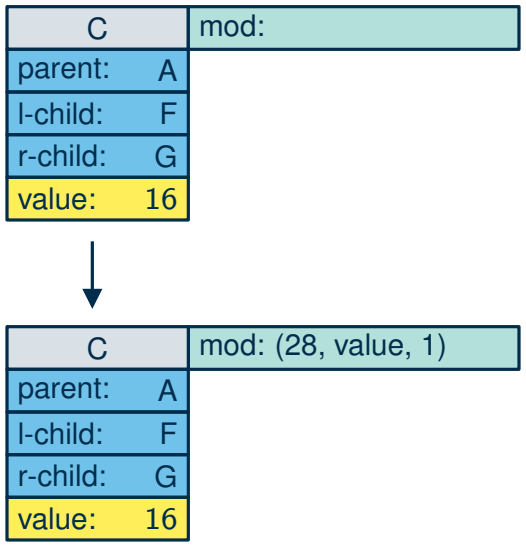
Problem: one modification per node is not enough

Idea

- store multiple modifications (we’ll think later about how many exactly)
- all mod fields occupied $\rightarrow$ create new node in the current state
- adjust pointers to point to new node $\rightarrow$ we need back-pointers

Example Operation

- operation: C.value = 1
- current timestamp: 28

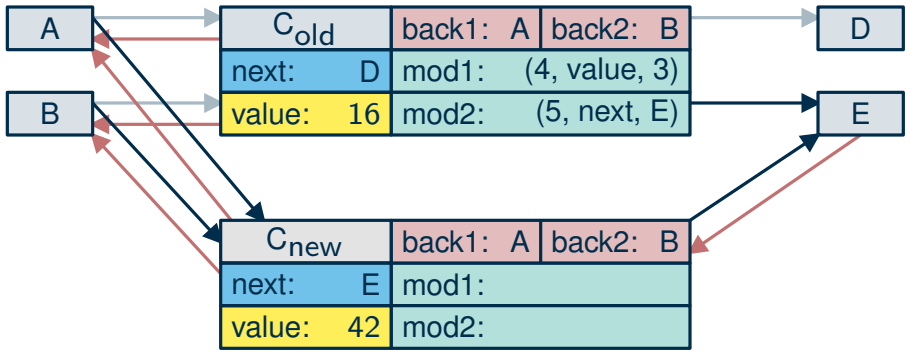


An Atomic Operation

Case 1: There Is An Empty Mod Field

example: C.value $=$ 3 | C.next $=$ E | C.value $=$ 42

- insert change into a mod field
- if pointer is changed: update back-pointer



Case 2: All Mod Fields Occupied

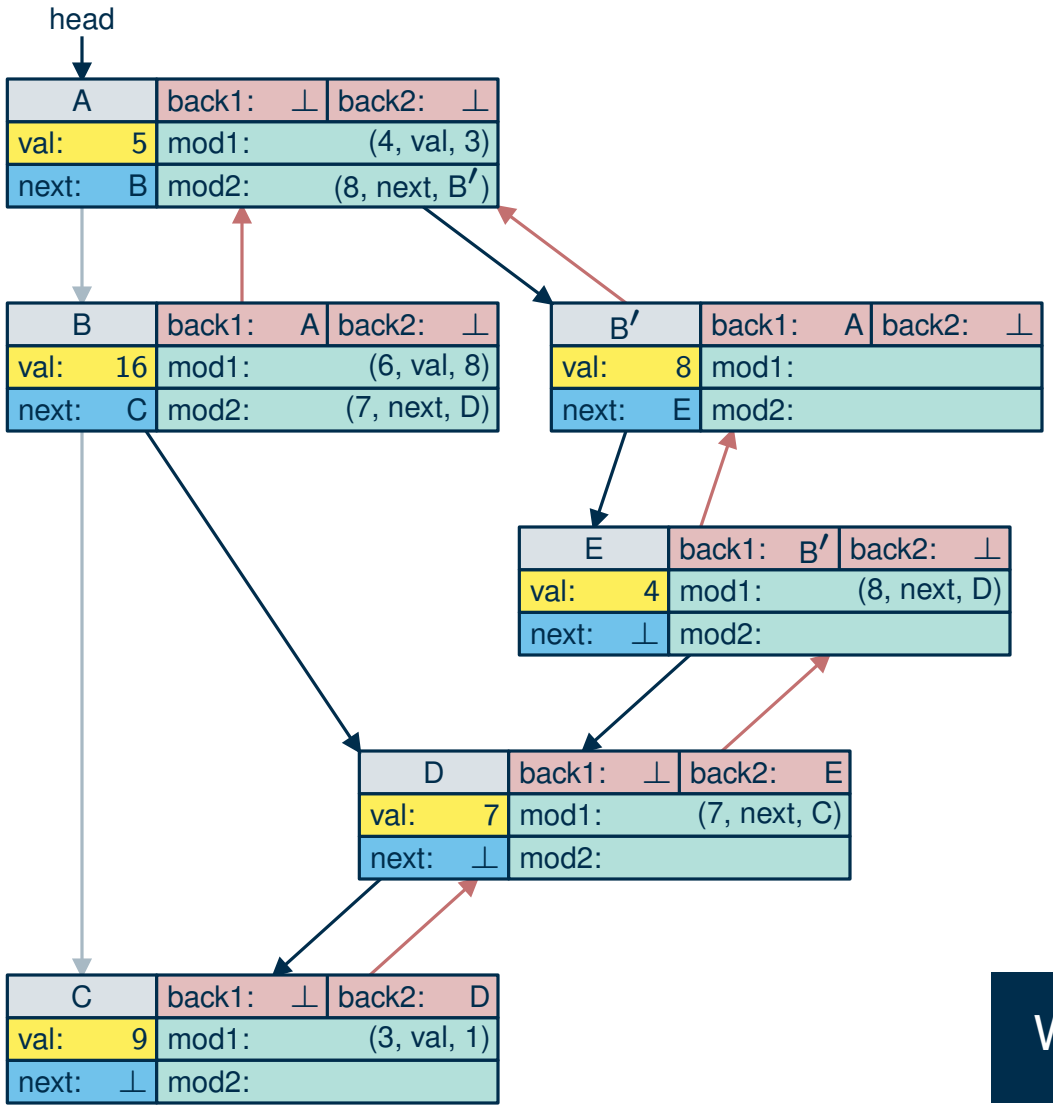
- copy node: old Node $v_{old} \rightarrow$ new Node v_{new}
- apply all modifications to v_{new} (including current operation)
- update nodes pointing to v_{old} to instead point to v_{new}
  - this is why we need the back-pointers
  - we have to make this change persistent
- remove back-pointers to v_{old}, add back-pointers to v_{new}
  - note: back-pointers are only needed for the present time

this creates new atomic operations:

$A.next = C_{new}$ | $B.next = C_{new}$

this recursion may cascade!

Example: Singly Linked List



List Operations

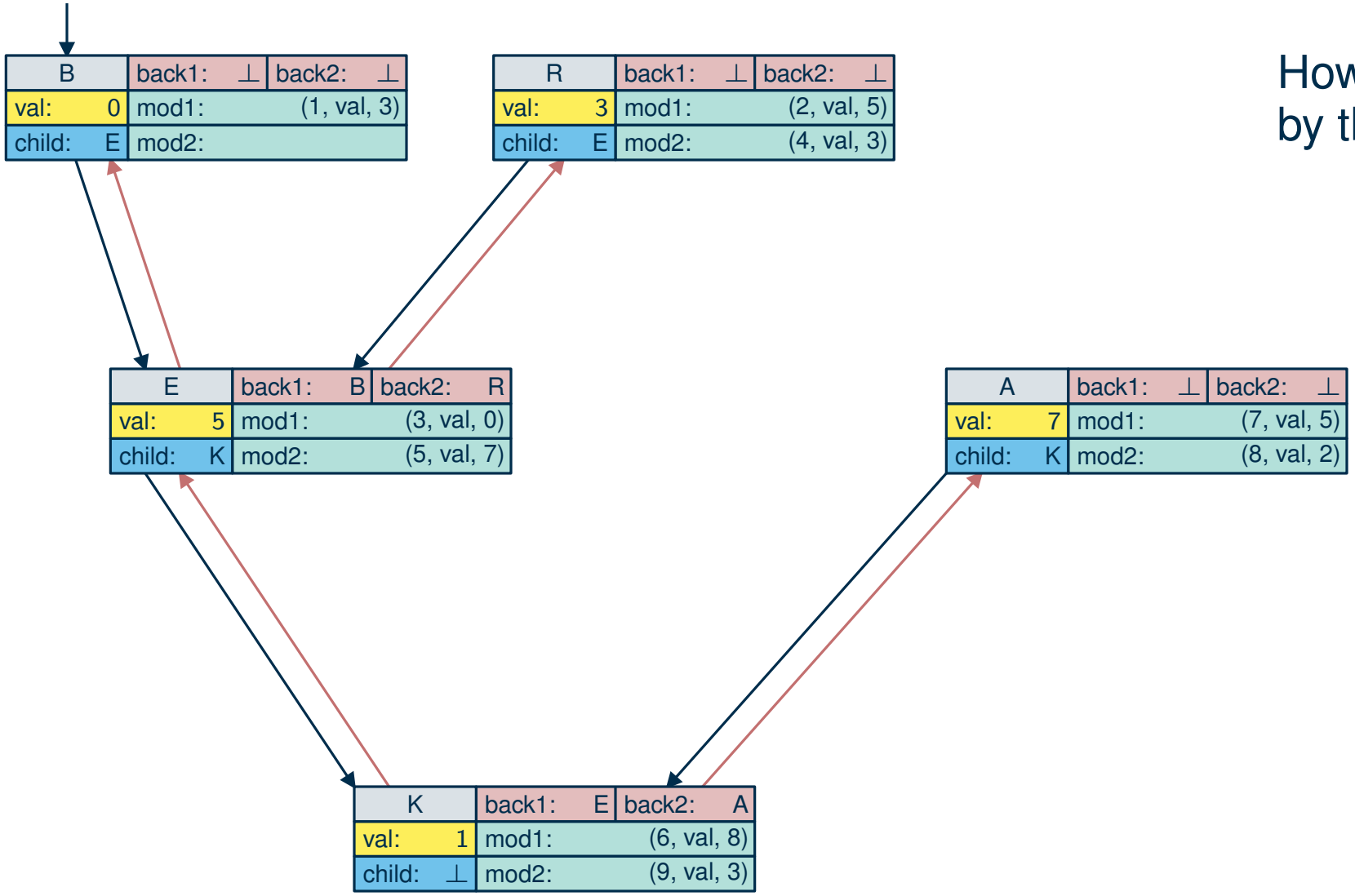
- set value of node X to x: set(X, x)
- insert new node with value y after X: ins(X, y)

time	operation	atomic operation
6	set(B, 8)	B.val = 8
7	ins(B, 7)	D = newNode(7)
		D.next = B.next
		B.next = D
8	ins(B, 4)	E = newNode(4)
		E.next = B.next
		B.next = E
		B.nextBack.next = B′

recursive call!

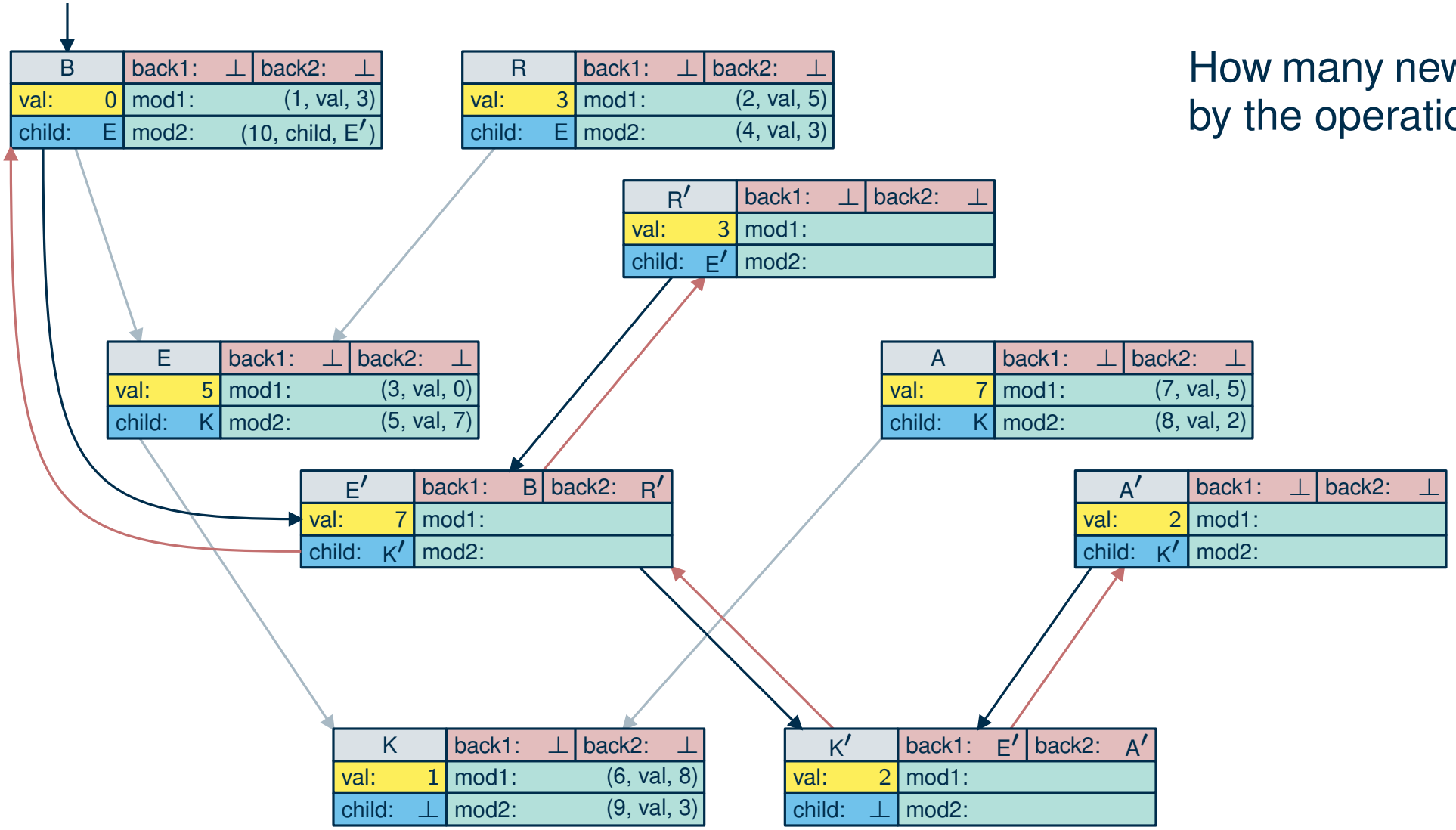
What would have happened, if A.mod2 was already occupied?

What Is Happening Here?



How many new nodes are created by the operation K.val $= 2$?

What Is Happening Here?



How many new nodes are created by the operation K.val $= 2$?

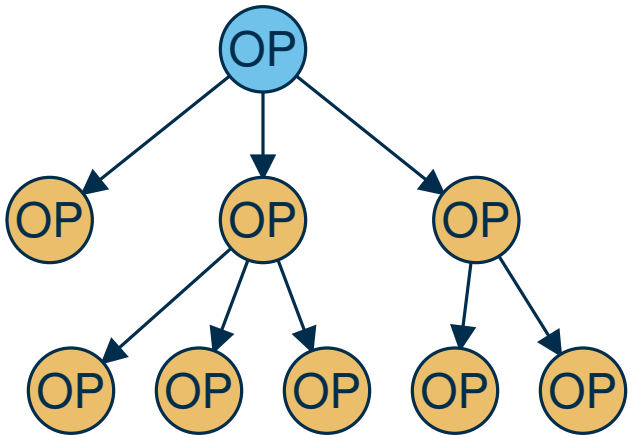
Running Time Of An Atomic Operation

(Almost) Everything Is Constant

- case 1: mod field empty $\rightarrow$ obviously $O(1)$ time
- case 2: copy node of size $O(1)$, apply $O(1)$ modifications, update $O(1)$ back pointers

Problem

- case 2 sometimes spawns in-degree many new atomic operations
- this can cascade $\rightarrow$ unbounded tree of additional atomic operations



But

- case 1: cheap but fills a mod field
- case 2: expensive but frees up mod fields
  - before: all mod fields of v_{old} occupied
  - after: all mod fields of v_{new} free

} note: v_{old} will never again be relevant for operations
$\rightarrow$ we call the newest copy of a node **active**

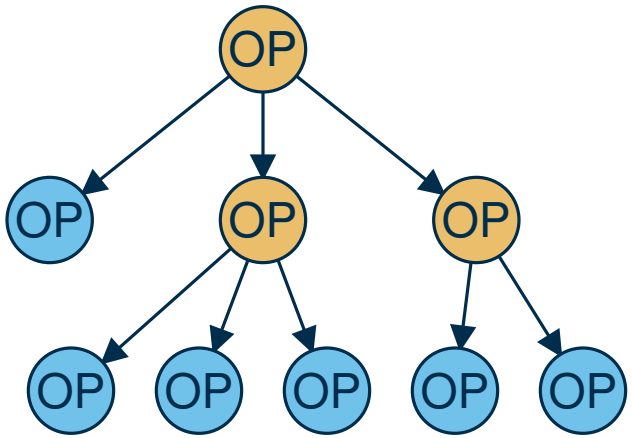
- amortized analysis: let case-1 operations pay for case-2 operations

Amortized Analysis

Some Variables
n_ℓ: #leaves in the tree
n_i: #inner nodes
k: #mod fields
d: in-degree in the DS

Accounting/Potential Method

- case-1 operations pay into account $\rightarrow$ deposit 1
- case-2 operations withdraw to cover cost $\rightarrow$ withdraw x
- plan: account balance $=$ #(occupied mod fields in active nodes)



How Much Can A Case-2 Operation Withdraw?

- n_i nodes with k occupied mod fields now empty $\rightarrow$ withdraw kn_i
- occupy one mod filed in each leaf $\rightarrow$ deposit n_ℓ

Bounding The Amortized Cost Of A Case-2 Operation

$$\underbrace{n_\ell + n_i}_{\text{actual cost}} + \underbrace{n_\ell - kn_i}_{\text{account balance change}} = 2n_\ell + n_i - kn_i \leq 2dn_i + n_i - kn_i = 0$$

as $n_\ell \leq dn_i$

using $k = 2d + 1$ mod fields

Wrap-Up: Persistence

Cost Of An Atomic Operation

- amortized constant time overhead
- amortized constant additional memory

Cost Of An Atomic Query

- constant time overhead

Theorem **(partial persistence for everyone)**
Every data structure in the pointer machine model with constant in-degree can be made partially persistent, such that every atomic operation has amortized constant time and space overhead. Every query has constant overhead.

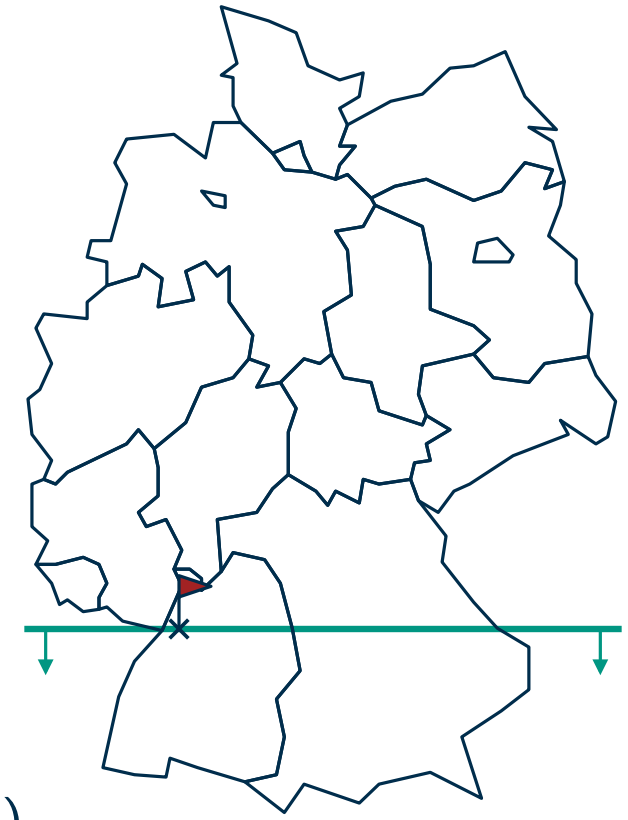
Remarks

- operation consists of x atomic operations $\rightarrow \Theta(x)$ additional space
- binary search tree: improvement to $O(1)$ memory per operation is possible
- theorem can be generalized to full persistence

Where does our approach fail for full persistence?

Where am I?

Problem: Point Location
In which face of a geometric graph does a given point p lie?



Static Variant

- graph G is fixed
- answer query for many points $p \in \mathbb{R}^2$

Run Sweep-Line Algorithm For Line Intersection

- time p_y: find edge e left of p in $O(\log n)$ (predecessor on sweep-line)
- output face next to e (to the right) in $O(1)$
- use persistent search tree for sweep-line status
  - "Where is $p = (p_x, p_y)$?" turns into
  - "Where was p_x at time p_y?"

Query: $O(\log n)$
Precomputation: $O(n \log n)$
Memory: $O(n)$

Is There A Solution Withouth Time Travel?

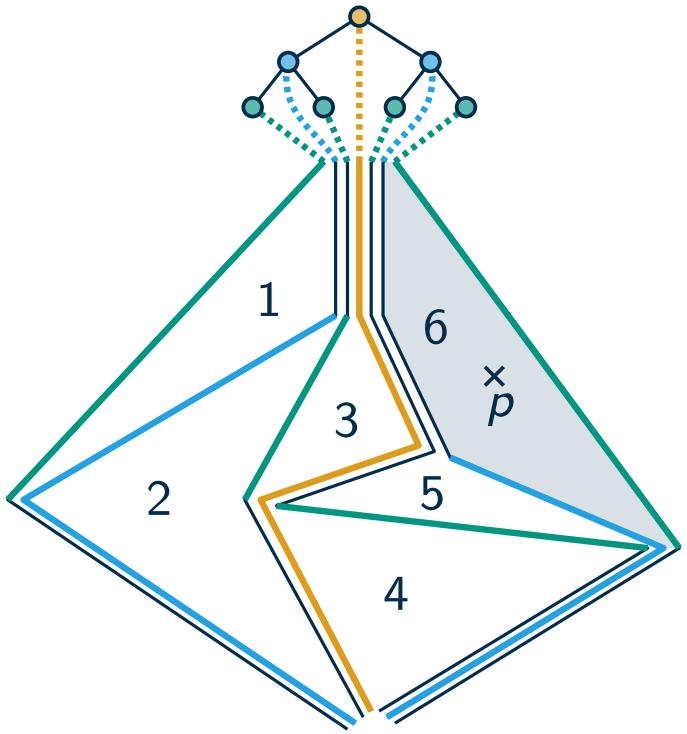
Bringing Order Into Chaos

- make faces y-monotone
- order faces from left to right
- split graph into paths using this order

} p in face $i \Leftrightarrow p$ between paths i and $i+1$
goal: find two consecutive paths with p in between

Finding The Position Of p With A Binary Search Tree On Paths

- decision at each tree node
  - find relevant y-position in the path $\rightarrow$ path edge e
  - walk left/right in the tree if p lies left/right of e
  - e non-existent $\rightarrow$ same decision as in step before
- query time: $O(\log n)$ binary searches $\rightarrow O(\log^2 n)$
- fractional cascading $\rightarrow$ **queries** in $O(\log n)$
- **precomputation** $O(n \log n)$ and **memory** $O(n)$



Wrap-Up

What Have We Learned Today?

- time travel is real and actually useful
- nice amortized analysis with accounting/potential
- different solutions for point-location
- using our toolbox: line intersection, triangulation, fractional cascading

What Else Is There?

- additional equally good approaches for point location: `www.csun.edu/~ctoth/Handbook/chap38.pdf`
  (including a randomized algorithm with similar analysis as for our 2D-LP)
- dynamic variants
- retroactive data structures: allow operations in the past affecting the present state