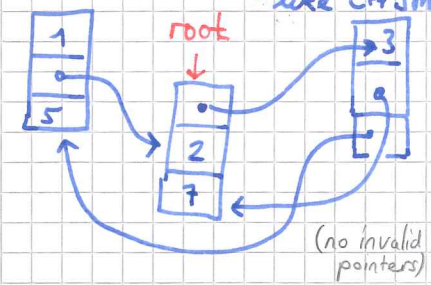
Lecture 4: Persistence / Temporal Data Structure

Computational Model: Pointer Machine

deal with nodes, each node has several fields, each field is a value or a pointer
↑ like C++ structs ↑ O(1) many



operations: → [not possible to have as array with O(1) access]

- x = new node
- x = y.field; x.field = y
- x = y + 2 (manipulate data, but not pointers)
- destroy x (if no pointers to x)

Temporal Data Structures

persistence
- never destroy old versions
- on update: makes new version
- many levels:

partial persistence ← full persistence ← confluent persistence ← functional persistence
only update latest version | update any version | merging 2 versions | never modify versions
list of versions (SVN) | tree of versions (GIT) | DAG of versions | (Haskell)

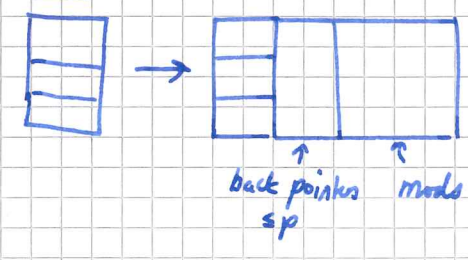
representations might change, behavior is preserved

retroactivity
- allow for „time travel", i.e. to perform updates in the past → next week

For partial persistence (Driscoll et al. 1989)

[Any pointer machine data structure with indegree ≤ p = O(1) (# pointers pointing to a node) can be made partially persistent with
- O(1) amortized multiplicative overhead (time & space)
- O(1) space per update]

Proof



1. ↑ node stores p back pointers latest version only
2. allows ≤ 2p modifications = (version, field, value)
   ↳ history of O(1) versions

Update:

- not full: add MOD (if switching pointers, also update backpointers)
- full: • create new node (node' = node with all MODs applied)

  - change backpointers (node → node') by following node' pointers
  - change pointers to node by following node' backpointers
    - ↳ append appropriate changes recursively (has to be persistent)

Analysis

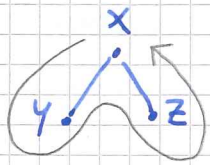
- termination: in the worst case, all nodes overflow, so $\leq p$ mods per node which causes only one new version (as there is space for $2p$ mods)
- amortization with potential: $\Phi = c \cdot \Sigma$ #mods in nodes in latest version

$$\text{amortized cost} = \underbrace{O(1)}_{\leq c} + \underbrace{O(1)}_{\leq c} + \underbrace{p \cdot \text{recursion} - 2 \cdot c \cdot p}_{\text{or this is } 0} \leq 2c \qquad \square$$

$O(1)$: compute + modify; $\leq c$: pay for potential change at node

For full persistence (Driscoll et al. 1987)

tree of versions → linearize with in-order traversal



$b_x\ b_y\ e_y\ b_z\ e_z\ e_x$ (↑ begin, ↑ end)

$b_v < b_w < e_w < e_v \Leftrightarrow v$ is ancestor of w

we want ancestry tests in $O(1)$

↳ maintaining ordered list

↳ insert, delete, order query in amortized $O(1)$ (but use RAM model for indirection)

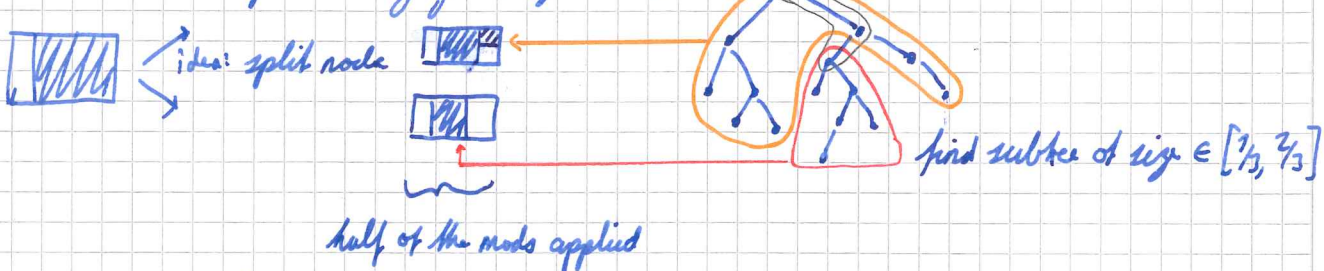
1. versions: root version, subtree of versions

→ queries: traverse version tree and use ancestry queries

2. also do backpointers persistently

p: max in/out degree, d: # fields, $2(d+p+1)$ mods

problem: constantly discharging (huge cost)



→ amortization still works out → everything $O(1)$ amortized