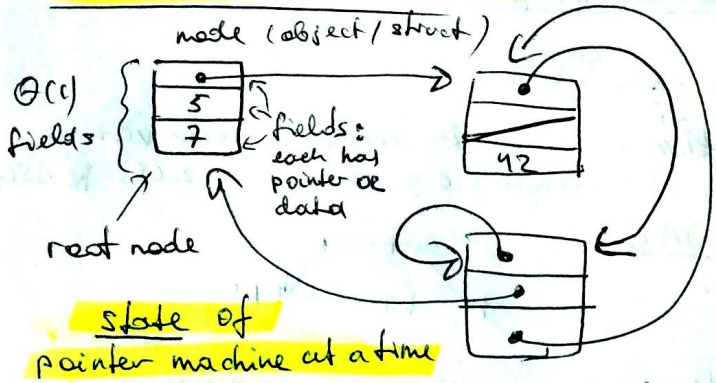
Lecture 1 Theme: Models matter.

Model of computation:

Pointer machine (1980s)



State of pointer machine at a time

"memory part" of model (state)

"computation part" of model ← my comment: state transitions

Operations, O(1)

- destroy (not used)
- x = new node // create new node
- x = y.field // look at field
- x.field = y // set field
- (x = y + z etc.) // arithmetic allowed (not used)

[redacted] nodes can be reached from root

(working relative to root) following pointers

BSTs, linked lists etc. — follow the pointer machine model.

Goal: given a pointer machine DS, make new pointer machine DS that provides "time travel" (travel through time)

Temporal DSs: make a change in the past → branch to a new universe, without destroying old universes.

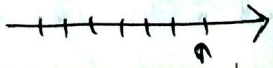
- persistence: (don't forget anything)
- retroactivity: go to the past, make change, return to the present, see what happened

Persistence:

- keep all versions of DS
- DS ops are relative to specified version
- update makes and returns new version

4 levels of persistence:

1) partial persistence
   - update latest version only
   - ⇒ versions linearly ordered

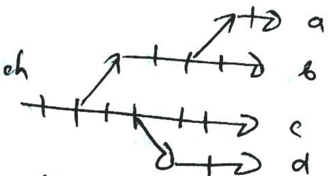


can query older versions, but cannot change them

2) full persistence

- update any version & branch

⇒ versions form tree



a does not know about b's future, cannot ~~merge~~ combine universes

3) confluent persistence (can ~~merge~~ combine)

- can "combine" >1 versions to create a new version (not destroying old versions) e.g. concatenate 2 LLs, or [redacted] BSTs

⇒ versions form DAG

4) functional:

- never modify nodes
- only make new nodes
  → can't modify nodes ⇒ keep all versions

confluent vs. full persistence: → same representation of (old) versions whereas in non-functional can use another representation

functional ⇒ confluent ⇒ full ⇒ partial

Partial persistence

any pointer-machine DS with $\leq p = O(1)$ (const max) pointers to any node (const. indegree) can be made partially persistent with O(1) amortized factor overhead + O(1) space/change

always have const max pointer out of a node

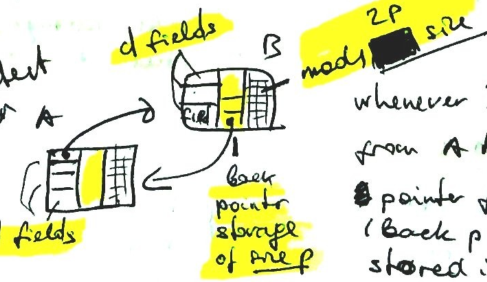
in a pointer machine outdegree is const.

Proof (construction + analysis)

① store back pointers for latest version

② store $\leq 2p$ mods to fields of DS

mod. = (version, field, value)



whenever ∃ pointer from A to B, make pointer from B to A (back pointer stored in the back pointer space of DS)

nodes are still constant size

back pointers only for nodes in latest version

- to read node.field at version v: look at field and then at mods with version $\leq v$ ⇒ O(1)

use time counter for versioning

- to modify node.field = x:
  - if node not full, add mod. (now, field, x)
  - else: make new node' with all mods applied to fields

my comment: downstream nodes from [redacted]

- update back-pointers to node → node'
- recursively update pointers (field updates in upstream nodes), may need to copy parent, grandparent .... root.

Analysis (amortized)

- potential Φ = c · Σ # nodes in latest-version nodes

"bad" state is when all the upstream nodes are full and each needs to be copied (recursively)

if recurse only in bold case

incl. in c for upper bound

$$\Rightarrow \text{amortized cost} \le c + C + [-2cp + p \text{ recursions}]$$

- c: [redacted] ops etc. time → c_i
- C: 1 node added in worst case → $\Delta\Phi$ increase each step
- $-2cp$: emptying nodes → $\Delta\Phi$
- p recursions: at most p pointers to the node; if recurse

when recursion happens $-2cp + p2c$ ..... cancel.

$= O(1)$ amortized ✓

even if there are cycles, recursion ends because Φ is non-negative.

1989 paper, early days of amortization. very cool!!!

My comment:

better $\Phi = \frac{c}{2} \sum$ # nodes in latest-version nodes

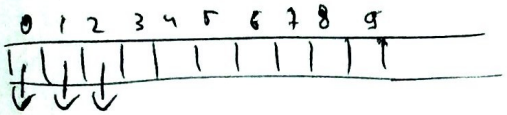
① case no recursion

$\hat{c} = c + \frac{c}{2} = \frac{3}{2}c$

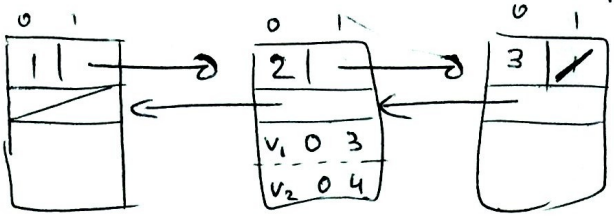
② case recursion

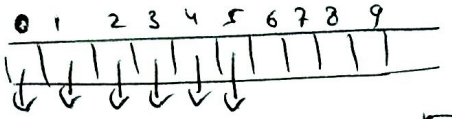
$\hat{c} = c + [-cp + p[c[-cp + p \dots]]] = c$

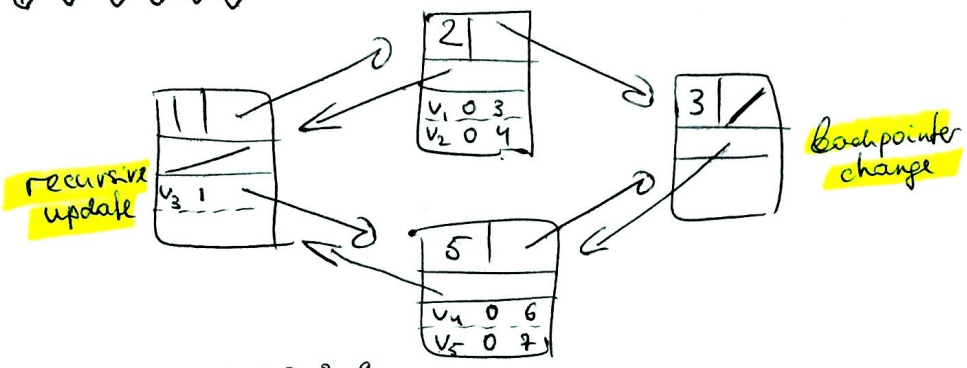
$\Rightarrow \hat{c} \le \frac{3}{2}c$

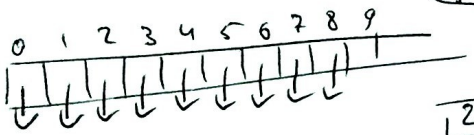


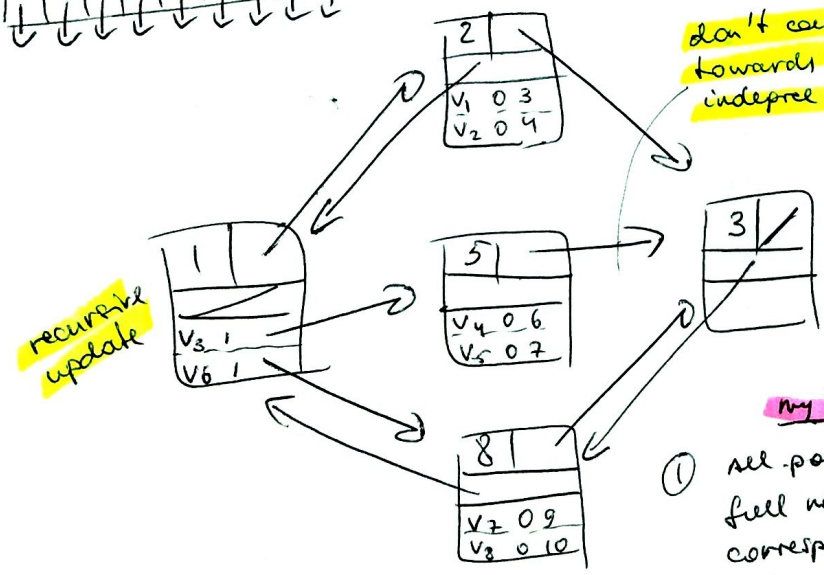
} array of access pointers to roots; root may be copied recursively during update (array technically not allowed in pointer machine)











① old full node is unchanged

my correctness comment

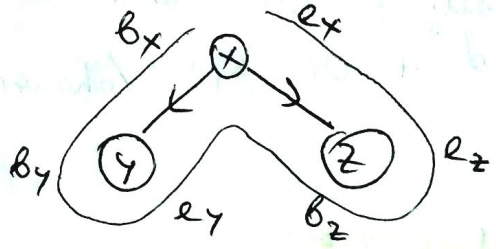
① all pointers leading to old full node from access pointers corresponding to versions in old full node, are preserved

② old full node is locked from recursive updates in the future

Diagram of construction is based on lecture notes and Driscoll et al.

Full Persistence (versions form a tree)

- linearize tree of versions (linearized representation)



b_x : beginning of X (first visited)

b_y : beginning of Y (first visited)

e_x end ...

e_y end ...

$b_x(b_y e_y)(b_z e_z) e_x$

my comment:

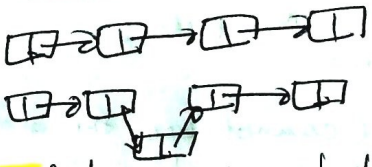
[redacted] DFS, preorder, postorder #s.

linear time

⇐ maintain begin & end of each subtree of versions (source version)

- use order-maintenance DS

update (1) - insert item before/after given item like in a linked list, can do in O(1)



insert new version ⇒ 2 insert (update operations) ⇒ O(1)

e.g. 2 version created from X

$b_x b_y e_y e_x$

↓

$b_x b_y e_y (b_z e_z) e_x$

or

$b_x (b_z e_z) b_y e_y e_x$

query (2) relative order of two items x & y

given x, y, which is to the left?

in O(1)

[redacted] CS version

V an ancestor of w?

$b_v < b_w < e_w < e_v$?

"in-node" recent ancestors upto the need closest ancestor for reading

⇒ O(1) to figure out which mod[redacted] applies to a given version to read [redacted] a field in a node

⇒ still getting O(1) to read a node with a const # of fields and mods.

Full persistence

any pointer-machine DS with $\leq p = O(1)$ pointers to any node can be made fully persistent with $O(1)$ amortized factor overhead + $O(1)$ space/change.

Proof:

- store back-pointers for all versions
- store $2(d+p+1)$ mods to fields of DS
  - ↑ # fields (out-degree max)
- read node.field at version v
  - look at all ancestor mods upto from nearest version ancestor of v (all that are in the node) $\Rightarrow O(1)$ per node
- if node not full, add mod

else: (cannot keep the old node full)

- split node into 2 halves
- split tree of mods
- apply mods to new node
- recursively update $\leq 2d+2p+1$ inverse pointers to node from nodes previously pointing to old node.

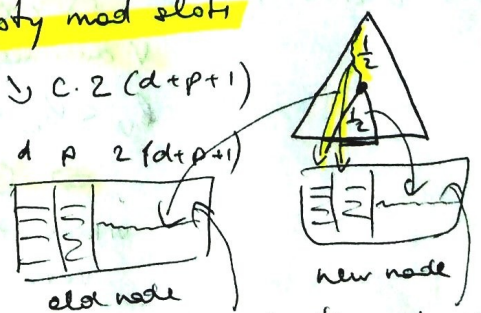
my comment: ↖ after copying node, the old node is no longer locked from updates, including recursive updates

- potential $\Phi = -c \sum$ # empty mod slots
- charge recursion to $\Phi \downarrow c \cdot 2(d+p+1)$

$\leq d+p+d+p+1$ recursions ✓

including inverse pointers to pointers in copied mods

size d p $2(d+p+1)$



De-amortize: $O(1)$ worst case overhead for partial persistence

for full persistence → OPEN Problem.

My comment

case 1 (normal): D_{i-1} has one more empty slot than D_i

$$\Rightarrow \Phi(D_i) - \Phi(D_{i-1}) = c$$

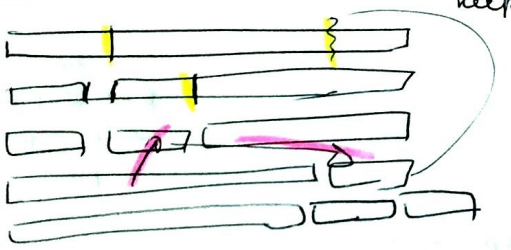
case 2 (Bad): D_{i-1} has $2(d+p+1)$ fewer empty slots than D_i

$$\Rightarrow \Phi(D_i) - \Phi(D_{i-1}) = -2(d+p+1)c$$

$$\hat{c} \leq c + c - 2(d+p+1)c + (2d+2p+1) \quad \text{recursion} \quad \checkmark$$

- c: compute
- c: normal case
- $-2(d+p+1)c$: Bad case
- $(2d+2p+1)$ recursion: recurse [redacted] with compute + Bad case

Confluent persistence



keeping all versions

confluence: can do cut/paste on data structures versions

DAG

split

concat

⇒ can double size of ds in a constant # of ops

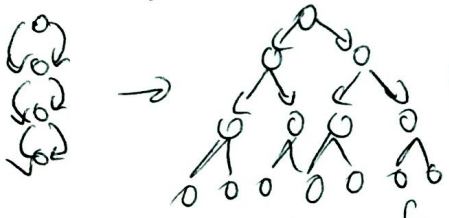
each time double size

in u updates → can get 2^u [redacted] and paths

General transform

- effective depth of a version in DAG
- $e(v) = 1 + \lg(\#\text{ paths from root to } v)$

2^u paths after u updates in this case



explicit representation of all paths in DAG

provides all paths to v from root

$\leq u$

- Overhead (multiplicative): $\lceil \lg u + \max_v e(v) \rceil$ · time of spree

$\#$ updates

max effective depth of all versions

pay effective depth of operation as a factor per operation

argument: linear slowdown is good

copying everything requires exponential slowdown per update

could be c linear slow down with $\#$ of updates if 2^u paths

Lower bound: $\sum_v \ell(v)$ bits of space in worst case

$\Rightarrow \Omega(\ell(v))$ overhead / update if queries are free

OPEN: Constant time, space overheads / operation
$o(\log(n))$

[redacted]

What if the size of ds does not grow exponentially?

↳ what if the size stays constant ← even in [redacted]
after update

↳ no strong reason for a linear overhead

Disjoint transform

assume confluent ops performed only on versions with no shared nodes

$\Rightarrow O(\lg n)$ overhead

look at all versions where node appears → tree

Functional persistence

- balanced BSTs (Path copying)
- deques (doubly ended queues ← stack and queue ops with concat. in $O(1)$
- tries (tree with fixed topology)
- log separation