6.851	Lecture 1	Feb. 7, 2012

Prof. Erik Demaine
TAs: Tom Morgan & Justin Zhang

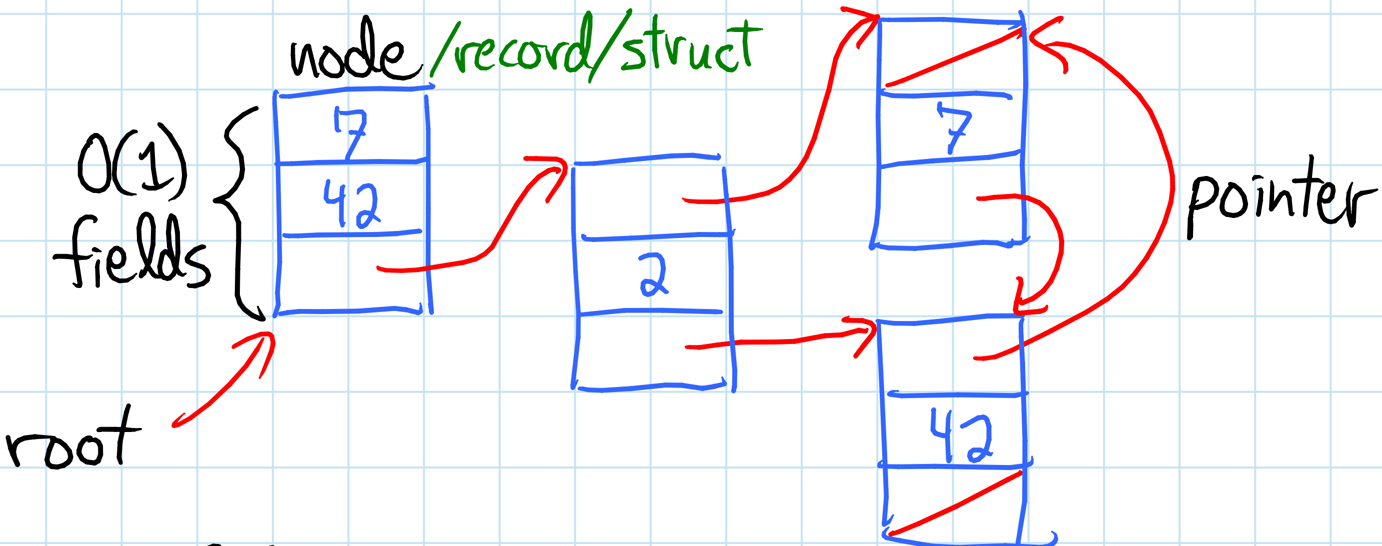
TOPICS:
- time travel: remembering/changing the past [THIS WEEK]
- geometry: >1 dimension (maps, DB tables)
- dynamic optimality: is there one best BST?
- memory hierarchy: minimize cache misses
- hashing: most used DS in CS
- integers: beat lg n time/op, or prove impossible
- dynamic graphs: changing computer/social network
- strings: search for phrase in text (DNA, web)
- succinct: reduce space to ≈ bare minimum

Administration:
- video recording of lectures
- requirements: attending lecture, ≈ weekly psets, scribing, project
- signup sheet
- listeners welcome
- problem session (starting ~ week 3)

[- scribe for today]

Theme in this class: THE MODEL MATTERS

Pointer machine: model of computation



- field = data item OR pointer to node
- operations: O(1) time each
  - x = new node
  - x = y.field
  - x.field = y
  - x = y + z etc. (data operations)
  - [– destroy x (if no pointers to it)]

  where x, y, z are fields of root (or root)

  ⇒ constant working space

e.g. linked list, binary search tree (BST), most object-oriented programs

Temporal data structures:
- persistence [L1]
- retroactivity [L2]

think: time travel

Persistence:

branching-universe model

- keep all versions of DS
- DS operations relative to specified version
- update creates (& returns) new version (never modify a version)
- 4 levels:

most of Terminator / Sarah Conner Chron.

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

   movie Déjà Vu part 1
2. full persistence:
   - update any version
   - ⇒ versions form a tree

   Déjà Vu part 2

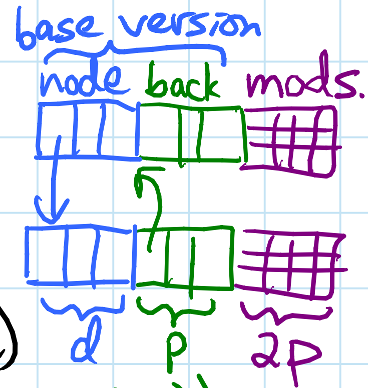
   Pullman's book Subtle Knife?
3. confluent persistence:
   - can combine >1 given version into new v.
   - ⇒ versions form a DAG

   TV show Sliders

   movie Primer?
4. functional:
   - never modify nodes; only create new
   - version of DS represented by pointer

Partial persistence: [Driscoll, Sarnak, Sleator, Tarjan — JCSS 1989]

any pointer-machine DS with $\le p = O(1)$ pointers to any node (in any version) can be made partially persistent with $O(1)$ amortized multiplicative overhead & $O(1)$ space per change



Proof:

- store reverse pointers for nodes in latest version (only)
- allow $\le 2p$ (version, field, value) mods. in a node (using that $p = O(1)$)
- to read node.field at version v, check for mods with time $\le v$
- when update changes node.field $= x$:
  - if node not full: add mod. (now, field, x)
  - else: - create $\underset{\text{empty mods.}}{\text{node}'} = \underset{\text{now old}}{\text{node}}$ with mods applied
    - change $\underbrace{\text{back pointers}}_{\text{found by following pointers}}$ to node $\to$ node$'$
    - recursively change $\underbrace{\text{pointers}}_{\text{found via back pointers}}$ to node $\to$ $'$

root node part of returned version

(- add back pointer from x to node)

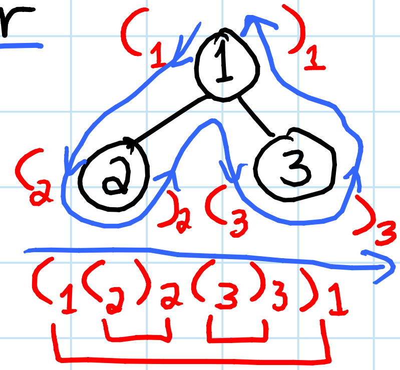
- potential $\Phi = c \cdot \sum$ #mods. in nodes in latest version

$\Rightarrow$ amortized cost $\le \underbrace{c}_{\text{compute}} + \underbrace{c}_{\text{mod.}} \underbrace{- 2cp + p \text{ recursions}}_{\text{if recurse}}$

$\le 2c$. □

Full persistence: ditto [Driscoll et al. 1989]

- linearize tree of versions via Euler tour, marking begin & end of each subtree
  ( )

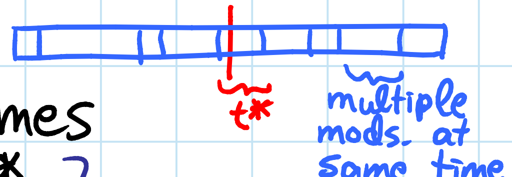


- paren sequence = linearized times
  - time $(_i$ makes change i
  - time $)_i$ unmakes change i

  $\Rightarrow$ version i accumulates changes at times $\le (_i$
- store times in order-maintenance DS:
  [L8: Dietz & Sleator - STOC 1987]
  - insert item before/after specified item (like linked list)
  - relative order of 2 items? < or >

  in $O(1)$ time/op.
- update to version i represented by 2 mods at times $(_{i'}$ & $)_{i'}$ inserted after $(_i$ or before $)_i$
  update ↲ ↳ undo update

- allow $\le 2(d+p+1)$ mods. per node
  - forward & backward pointers now symmetric
- when node overflows → $2(d+p+1)+1$ mods.
  split into 2 nodes, each ≈ half full (like B-tree)
  - find median time t^* in sorted order of mod. times

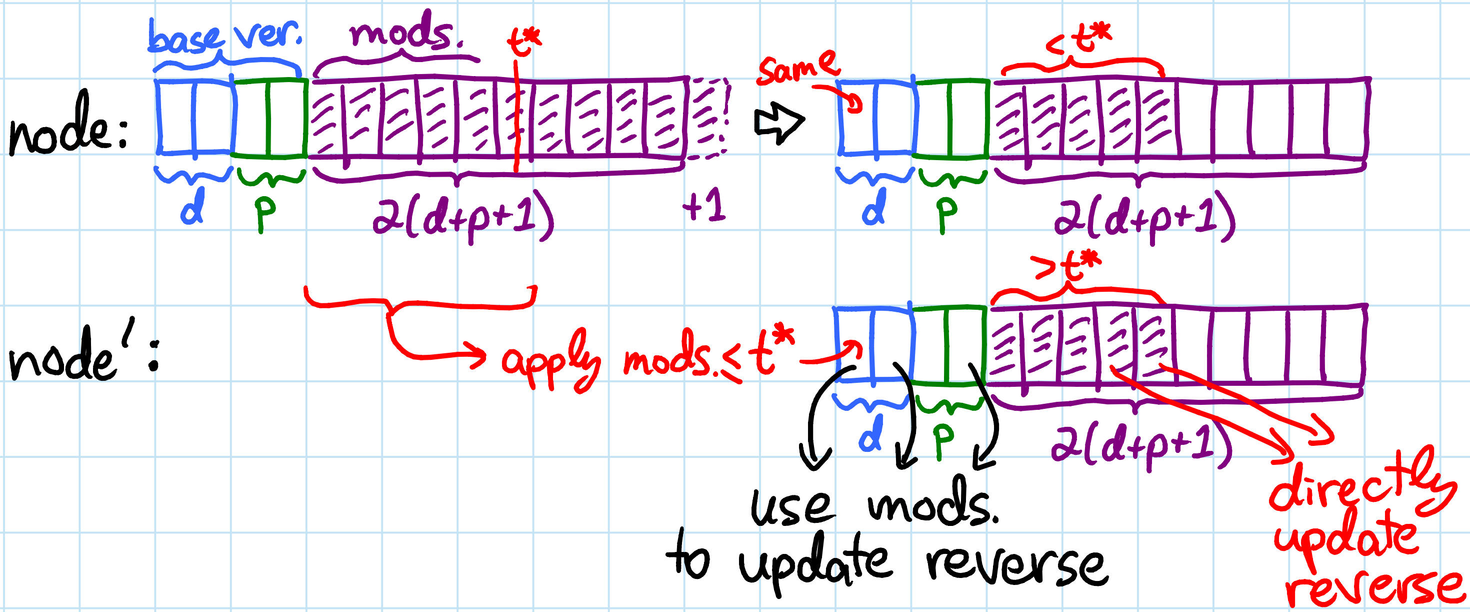


  - old node keeps mods. at $< t^*$
  - new node' keeps mods. at $> t^*$

  } exclude t^* ⇒ each $\le d+p+1$

  & base version = old base + all mods. at $\le t^*$ including t^*

- use mods. to update reverses of $\le d+p$ pointers in base version of node′ (node → node′)
- directly update reverses of pointers in mods. of node′ (their time $> t'$ ⇒ don't care about node)

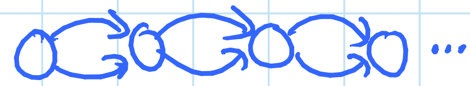


- potential $\Phi = c \cdot \sum_{\text{node}}$ (# mods. in second half of node)
  i.e. don't count first $d+p+1$ mods.
  - update creates 2 mods. ⇒ $\Phi \nearrow 2c$
    ⇒ $O(1)$ amortized cost ignoring splits
  - split: $\Phi \searrow c(d+p+1)$ in node (empty second half)
    & $\Phi \nearrow c(d+p)$ to update reverses of base version of node′
    ⇒ net $\Phi \searrow c$
    - set c to the work of one split (excluding any recursive splits)
    ⇒ amortized cost $= 0$

De-amortization:

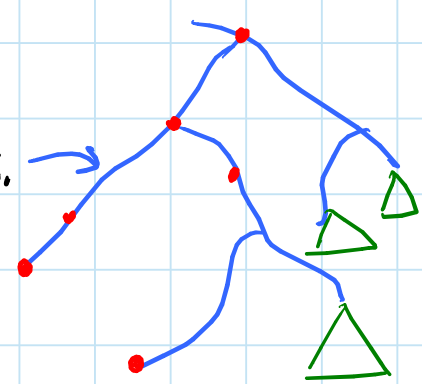
- partial: $O(1)$ worst case [Brodal – NJC 1996]
- full: $\boxed{\text{OPEN}}$: $O(1)$ worst case?

<u>Confluent persistence</u>:



- after u confluent updates, can get size 2^u
- general transformation: [Fiat & Kaplan – J.Alg 2003]
  - $d(v)$ = depth of version v in version DAG
  - $e(v) = 1 + \lg(\#\text{ paths from root to } v)$
  - <u>overhead</u>: $\lg(\#\text{updates}) + \underbrace{\max_v e(v)}_{\text{can be up to } u\ldots}$ time & space
    - still exponentially better than complete copy...
- <u>lower bound</u>: $\sum_v e(v)$ bits of space [Fiat & Kaplan]
  - $\Rightarrow \Omega(e(v))$ for update *if* queries are free
  - construction makes $\approx e(v)$ queries per update
- $O(\lg^3 \max_v e(v))$ update, $O(\lg^2 \max_v e(v))$ query [Fiat & Kaplan]
- $\boxed{\text{OPEN}}$: $O(1)$ or even $O(\lg n)$ overhead per op.?

- disjoint transformation: [Collette, Iacono, Langerman – SODA 2012]
  - assume confluent ops. performed only on versions with no shared nodes
  - then $O(\lg n)$ overhead possible
  - <u>Idea</u>: each node in <u>subtree</u> of version DAG
    - only <u>some</u> of those versions <u>modify node</u>
    - 3 types of versions:
      - <u>node modified</u> ~ easy
      - along <u>path</u> between mods.
      - <u>below</u> a leaf ~ hard



  - fractional cascading [L3] & link-cut trees [L19]

Functional: [Okasaki - book 2003]

path copying

- simple example: balanced BSTs
  - work top-down ⇒ no parent pointers
  - duplicate all $\underbrace{\text{changed nodes \& ancestors}}_{O(\lg n)}$ before changing

  ⇒ $O(\lg n)$/op.

⇒ link-cut trees too [Demaine, Langerman, Price]

- e.g. deques with concat. in $O(1)$/op.
  double-ended queues [Kaplan, Okasaki, Tarjan - SICOMP 2000]
  + update & search in $O(\lg n)$/op.
  [Brodal, Makris, Tsichlas - ESA 2006]

- tries with local navigation & subtree copy/delete & $O(1)$ fingers maintained to "present"
  [Demaine, Langerman, Price - Algorithmica 2010]

think: Subversion

method	finger move time	finger move space	modification (time = space)	
path copying	$\lg \Delta$	$\emptyset$	depth	
1. functional	$\lg \Delta$	$\lg \Delta$	$\lg \Delta$	local mods. cheap
1. confluent	$\lg\lg \Delta$	$\lg\lg \Delta$	$\lg\lg \Delta$	
2. functional	$\lg \Delta$	$\emptyset$	$\lg n$	globally balanced
2. confluent	$\lg\lg \Delta$	$\emptyset$	$\lg n$	

Beyond:

- functional: $\geq \log$ separation from pointer machine [Pippinger – TPLS 1997]
- $\boxed{\text{OPEN}}$: bigger separation? general transformations? } functional & confluent
- $\boxed{\text{OPEN}}$: lists with split & concatenate?

→ SOLVED by Blame Trees in 6.851 Spring '12 [Demaine, Panchekha, Wilson, Yang – WADS 2013]