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

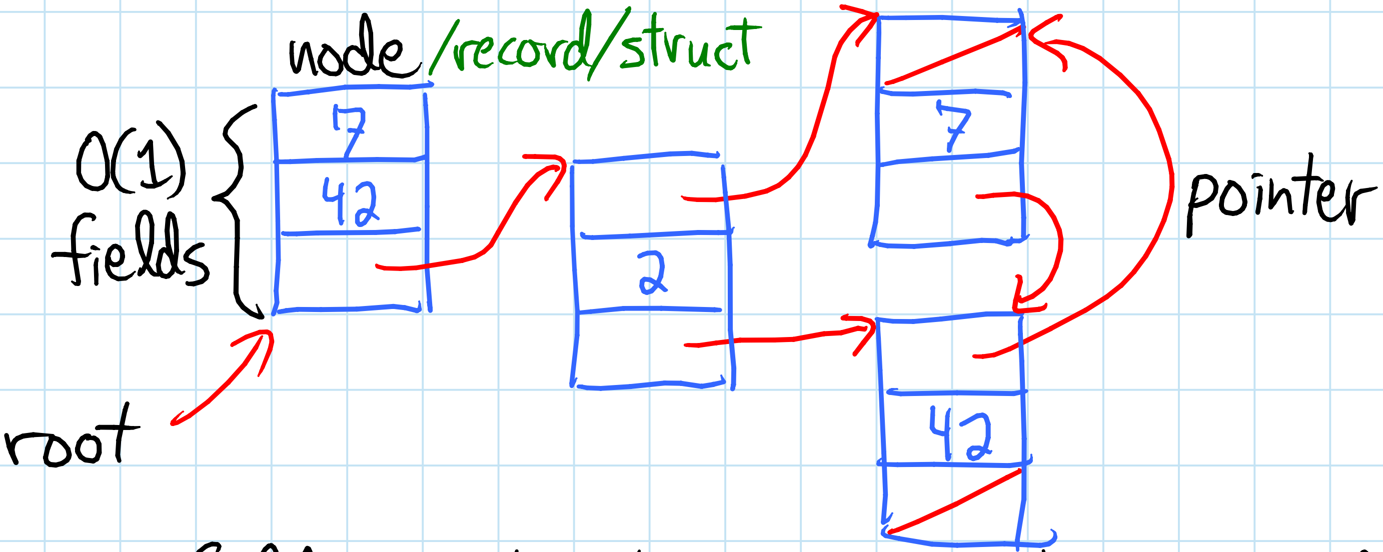
Administration:

- 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

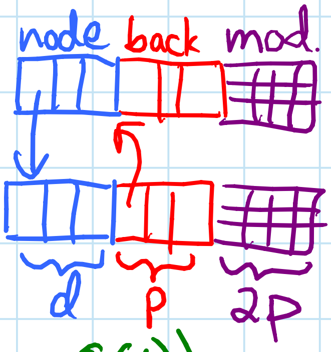
movie Primer?

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 node′ = node with mods applied

      empty mods. ↑ ↑ now old
    - change back pointers to node → node′

      ↳ found by following pointers
    - recursively change pointers to node → ′

      found via back pointers

(− add back pointer from x to node)

root node part of returned version

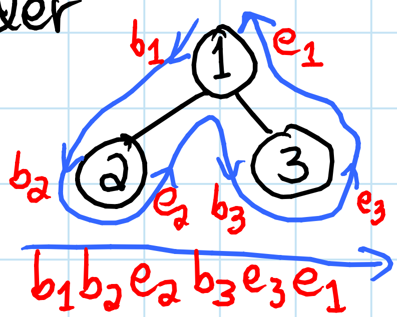
- potential $\Phi = c \cdot \Sigma$ # mods. in nodes in latest version

$\Rightarrow$ 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 in-order traversal, marking $\underbrace{\text{begin}}_{b}$ & $\underbrace{\text{end}}_{e}$ of subtree



- store sequence of b's & e's in order-maintenance DS:
  [L8: Dietz & Sleator – STOC 1987]
  - insert item before/after specified item (like linked list)
  - relative order of 2 items?

  in O(1) time/op.
- version v ancestor of w ⟺ $b_v < b_w < e_w < e_v$ — order in list

⇒ O(1) time/op.

⇒ can tell which mods apply to specified version

- create child version of v via 2 inserts after b_v
- allow ≤ $2(d+p+1)$ mods. per node
- when changed node is full:
  - split into two nodes, each half full (like B-tree node) by making copy with half mods. applied, half left
  - recursively update pointers & back pointers to copy
- potential $\Phi = -c \sum \#$ empty mod. slots (all nodes live)

⇒ charge ≤ $\underbrace{d+p}_{\text{from rest}} + \underbrace{(d+p+1)}_{\text{from mods.}}$ recursions to $\Phi \searrow c\,2(d+p+1)$

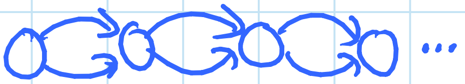
⇒ O(1) amortized

actually splitting mod. version tree 1/3 : 2/3

De-amortization: (see L10)

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

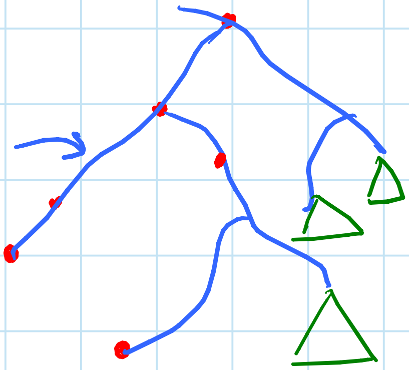
Confluent persistence:



- 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
  - $e(v) = 1 + \lg(\#\text{ paths from root to } v)$
  - overhead: $\lg(\#\text{updates}) + \underbrace{\max_v e(v)}_{\text{can be up to } u\ldots}$ time & space
    - still exponentially better than complete copy...
- lower bound: $\sum_v e(v)$ bits of space [Fiat & Kaplan]
  - $\Rightarrow \Omega(e(v))$ for update *if* queries are free
  - construction makes $\approx e(v)$ queries per update
- $\boxed{\text{OPEN}}$: $O(1)$ 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

Idea: each node in subtree of version DAG
- only some of those versions modify node
- 3 types of versions:
  - node modified ~ easy
  - along path between mods.
  - below 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

	finger move		modification	
method	time	space	(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]
- OPEN: bigger separation?
general transformations? } functional & confluent
- OPEN: lists with split & concatenate?
- OPEN: arrays with copy & paste?

6.851 Advanced Data Structures
Spring 2012