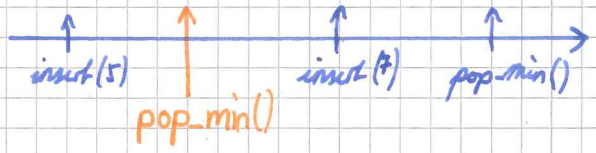
Lecture 5: Temporal Data Structures II: Retroactivity

retroactive: only a single timeline, but we can modify the past

data structure: sequence of updates



changes to this sequence:
- insert(t, "OP(...)") } everywhere
- delete(t)
- query(t, "OP(...)") → partial retroactivity: query „now"
  → full retroactivity: query anytime

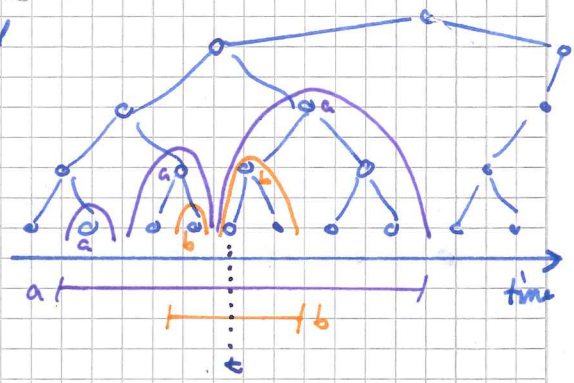
① commutative updates: $x \circ y = y \circ x$ then Insert(t, x) = Insert(now, x)
② invertible updates: Delete(t) = Insert(now, x^{-1})
If ① + ②, partial retroactivity is easy. E.g. - Hashing
- Array with $A[i] \mathrel{+}= \Delta$
- Ordered set

For full retroactivity?

want decomposable search queries (set + queries)

query$(X, A \cup B)$ = query(X, A) ∘ query(X, B) i.e. nearest neighbor, successor

→ full retroactivity with $O(\log m)$ overhead (m = overall #updates)



- insert, delete into a set corresponds to modifying the element existence interval
- query(t): walk up from t to root } is $O(\log m)$ steps

missing details: how to do this for non-static timeline? how to rebalance the tree?

General transformation?

- rollback method:

unwind r operations, perform update, redo r operations } $O(r)$ overhead

this overhead is unavoidable: (in some models: algebraic decision trees)

DS: maintain: X, Y

updates: $X = x$, $Y \mathrel{+}= \alpha$, $Y = Y \cdot X$

sequence: $X = x$, $Y \mathrel{+}= \alpha_n$, $Y = Y \cdot X$, $Y \mathrel{+}= \alpha_{n-1}$, $Y = Y \cdot X$, ..., $Y \mathrel{+}= \alpha_0$

update + query = evaluation of a polynomial (Horner scheme)

↳ $\Omega(r)$ lower bound in algebraic decision trees

- priority queue: insert(x), pop_min()
  Demaine et al. 2003: partially retroactive $O(\log n)$

- successor data structure: set with next(x) query
  partial: $O(\log n)$ (by ① + ②)
  full: $O(\log^2 n)$ (fully decomposable)
  $O(\log n)$ (hard, Giora & Kaplan 2009)

Aim: list order maintainance

in $O(1)$: insert, remove, query(x,y) is $x < y$ $\quad o \leftrightarrow o \Rightarrow o \nearrow^{\circ} \searrow o$

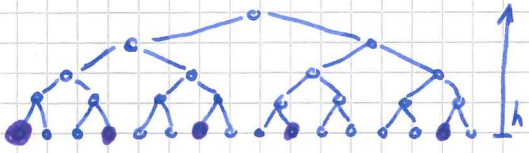
first: label space maintainance

linked list with monotone integer labels

5 7 9 ⇒ 5 6 7 9 ⇒ 4 5 6 8 10

	label space	label size	time	
1)	$\Theta(n) - \Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log^2 n)$	
2)	$\Theta(n^{1+\varepsilon}) - \Theta(\text{poly}(n))$	$\Theta(\log n)$	$\Theta(\log n)$	→ interesting to us
3)	$\Theta(2^n)$	$\Theta(n)$	$\Theta(1)$	→ easy, good for very small sets

2) maintain sparse tree of labels ≡ trie of labels



$\#\text{space} = 2^h$

$\#\text{usage}$ = number of used nodes below

$1 < \alpha < 2$

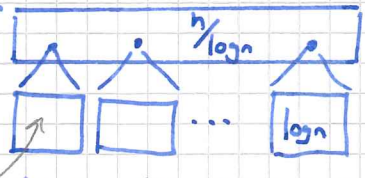
density of a node $= \frac{\#\text{usage}}{\#\text{space}} \leq \frac{1}{\alpha^h}$

update: ① try to squeeze in the element
② walk up the tree until you find a node x with small density
③ rebalance whole subtree below x (costs $O(\frac{2^h}{\alpha^h})$, i.e. linear in #labels)

Idea: single child of x: y is also balanced
and: child density − child threshold $\geq \frac{1}{\alpha^{h-1}} - \frac{1}{\alpha^h} = \Theta(\frac{1}{\alpha^h})$
thus: $\Omega(\frac{2^h}{\alpha^h})$ inserts into the child needed to make it unbalanced
→ $O(1)$ amortized per ancestor → $O(\log n)$ amortized overall

now for list order maintenance

two levels:



} 2) $\Theta(\log n)$ time, label size $\Theta(\log n)$
↳ worst case, it amortizes to $O(1)$
} 3) $\Theta(1)$ time, label size $\Theta(\log n)$

keep it balanced [¼ log n, ¾ log n]

label = (top label, bottom label) ⇒ $O(1)$ amortized update time