CS 261: Data Structures

Week 9: Time travel

Lecture 9b: Persistent search trees and retroactivity

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Review of persistence

Persistence

Classical data structures

Handle a sequence of update and query operations

Each update changes the data structure

Once changed, old information may no longer be accessible

Persistent data structures

Each update creates a new version of the data structure

All old versions can be queried and may also be updated

Types of persistence

Partial persistence

Updates operate only on latest version of structure

Queries can examine old versions

History is linear (sequence of operations forms a single timeline)

Full persistence

Updates can be applied to any version

History forms a tree
(updating an old version creates a new branch)

Confluent persistence

Updates can combine multiple versions (like git merge)

History forms a directed acyclic graph

Path copying

Requires a tree-like data structure in which each node can be reached by a unique path from a root node

Represent each version as a pointer to its root node

Query: same as non-persistent, starting from the version's root

Update: make new copies of all nodes on paths from root to changed nodes, with links to old unchanged nodes

Fat nodes

General technique for making any data structure persistent

- Divide the structure into pieces with a constant number of words (nodes of a node-pointer structure, cells of an array, individual words of memory)
- Each piece stores the history of what has been stored there
- To access a version of the data structure, simulate a non-persistent operation, replacing each read or write of a piece of data by a query or update to its local history

Analysis

Path-copying

Limited to top-down tree-based structures

Query and update time: Same as non-persistent

Space: Same as total update time

May be significantly bigger than non-persistent space

Fat nodes

Works for any data structure

Query and update time: slowed by history queries in each fat node

Space = O(changed data in update), not O(total update time)

Persistent binary search trees

First we need a good non-persistent tree!

Splay trees won't work
because of general incompatibility of persistence with amortization

We need $O(1)$ structural changes per update
to get good space complexity out of fat nodes

Changing extra information used to maintain balance is free
(at least for partial persistence)
keep only for latest version, non-persistently

Red–black trees

Each node stores one bit (its color, red or black)

Constraints: Root and children of red nodes are black; all root-leaf paths have equally many black nodes $\Rightarrow$ height $\leq 2 \log_2 n$

Messy case analysis: $O(\log n)$ time and $O(1)$ rotations per update

[Guibas and Sedgewick 1978]

WAVL trees

(WAVL = "weak AVL", also called rank-balanced trees)

Each node stores a number, its rank

Constraints:

- External nodes have rank 0
- Internal nodes with two external children have rank 1
- Rank of parent is rank of child + 1 or + 2

Simpler case analysis: $O(\log n)$ time and $O(1)$ rotations per update

[Haeupler et al. 2015]

Application of persistence techniques

For a partially persistent WAVL search tree after n updates:

- Path copying uses $O(\log n)$ time per operation but takes a total of $O(n \log n)$ space
- Fat nodes use $O(\log n \log \log n)$ time per operation and are complicated (flat trees) but use only $O(n)$ space
- Hybrid structure (detailed on the following slides) combining both path copying and fat nodes has $O(\log n)$ time per operation, $O(n)$ space, the best of both worlds. And it's much simpler than fat nodes because it doesn't need flat trees.

Hybrid persistent search trees

Versions are numbered (as in the fat node technique)

Pieces of memory = nodes in a WAVL tree

- We store the node ranks non-persistently, because we only need them to update the latest version of the structure
- Each update causes $O(1)$ rotations changing the tree structure

Each node stores its local structure (left and right children) for up to three versions

When we want to update the structure of a node and its local history is full (already stores three versions), we make a copy of the node and add a new local version at its parent pointing to the copy

Hybrid tree analysis

Because this is only partially persistent, we can use amortized analysis

Potential function Φ = sum over nodes of most recent version of the tree of how many versions are stored at each node

Making a local change in structure and adding a new version increases Φ by one

Making a new node to replace a full node decreases Φ by two (at that node) and increases it by one (at the parent node)

So decrease in Φ cancels extra space used to create new node and the amortized space (not amortized time) per update is $O(1)$

Time per operation = non-persistent WAVL tree operation $\times$ time to find correct version at each node = $O(\log n) \times O(1) = O(\log n)$

An application of persistent search trees

The locus method

Method for building data structures for problems where:

Data does not change

Queries are points in the plane

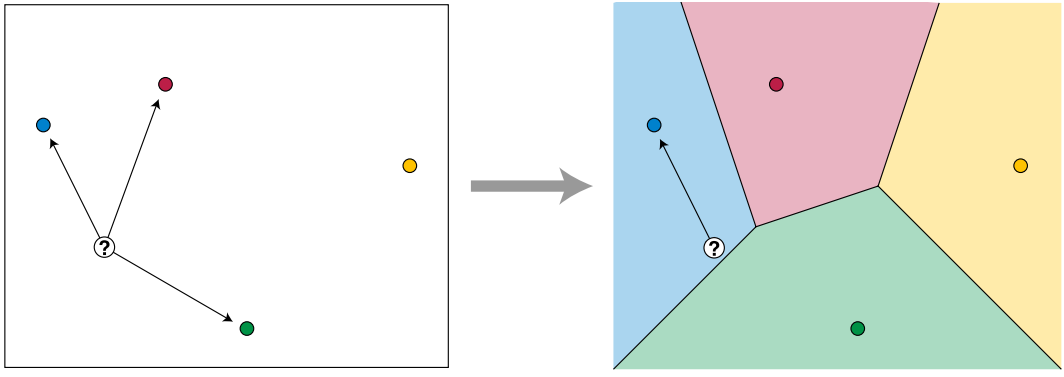
Answers are constant over regions of the plane
(rather than varying continuously)

Partition plane into regions within which answer is constant

Build "point location" data structure
that can find the region containing each query

Post offices and Voronoi diagrams

The post office problem: given a set S of points in plane
answer queries asking: which point in S is closest to query point q?



Voronoi diagram: partition plane into regions surrounding each point of S, within which that point is closer than any of the others

Point location in Voronoi diagram = answer to post office problem

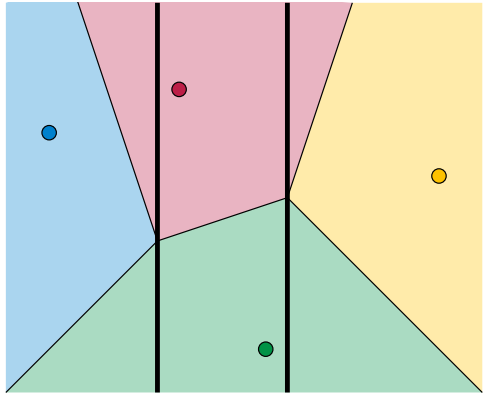
Point location by slabs

Simplifying assumptions: regions are polygons, at most three meet at any vertex, no vertical boundaries

Partition plane by vertical lines through vertices

Point location: binary search among x-coordinates of vertical lines, then binary search in vertical ordering of regions in slab between two vertical lines

Query $O(\log n)$, space $O(n^2)$

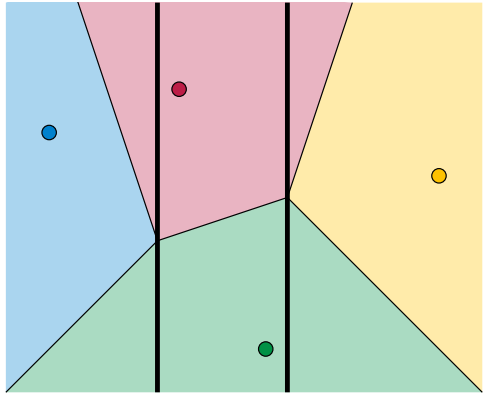


Adjacent slabs are not very different

When a vertical slab boundary passes through the rightmost vertex of a Voronoi region, the slabs to the left and right differ by removing that region

When a vertical slab boundary passes through the leftmost vertex of a Voronoi region, the slabs to the left and right differ by adding that region

Otherwise the vertical ordering of regions stays the same!



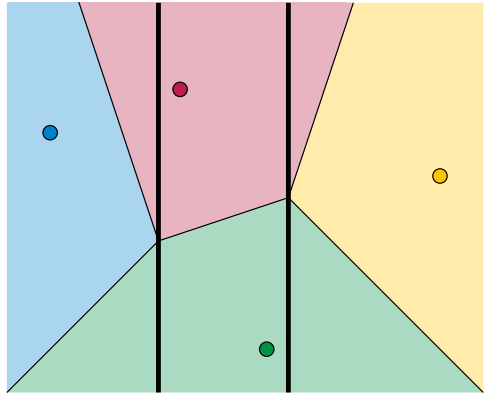
Point location by persistent search trees

Build partially persistent binary search tree of vertically ordered regions, with one version/slab

Point location: binary search in x-coordinates of vertical lines to find slab and its version, then search in that version of the persistent search tree

Each binary search tree comparison: test whether query point is above/below region boundary

Query $O(\log n)$, space $O(n)$



Retroactive data structures

Persistence / retroactivity in time travel movies

Partial persistence: You can visit the past but you can't change it; there is always a single unchangeable timeline

Example: *Harry Potter and the Prisoner of Azkaban*

Persistence / retroactivity in time travel movies

Full persistence: Each change in the past creates a separate branch in the timeline

Example: *Groundhog Day*

Persistence / retroactivity in time travel movies

Retroactive: There is only one timeline; changing the past also changes the present

Example: *Back to the Future*

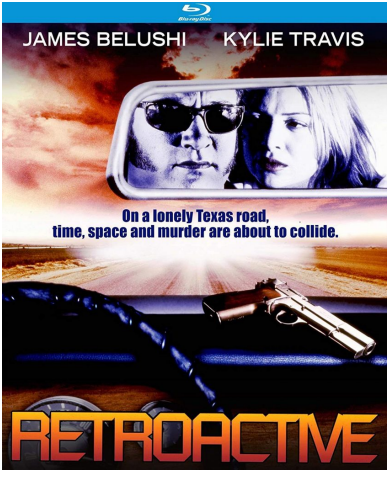
The main idea

There is a single linear timeline (times might as well be numbers)

Each operation has the time it should have been performed as one of its arguments

Updates change the timeline by adding or removing operations

Queries are performed at the given point in the current version of the timeline



Partial versus full retroactivity

Partially retroactive

Updates can happen in the past

Updates can be removed from the timeline as well as inserted

All queries must happen in the present
(that is, their timestamp must be $\geq$ all updates)

Fully retroactive

All operations can have any timestamp (past or present)

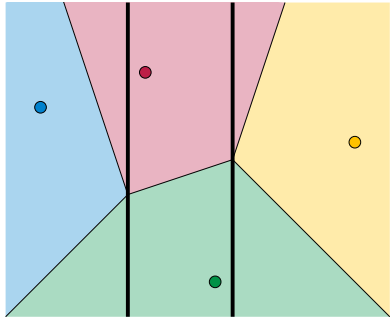
Point location revisited

Time: x-coordinate

Structure: Binary search tree of line segments, ordered by y

Timeline: At x-coordinate of left segment endpoint, insert it into search tree; at coordinate of right endpoint, delete it

To locate point (x, y), search for y in vertical sequence of segments @ time x

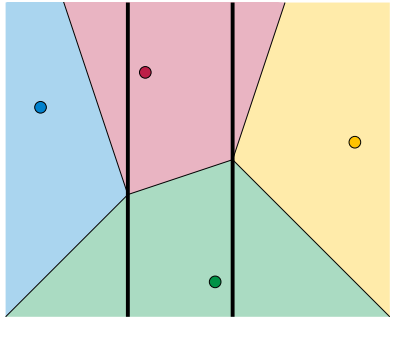


Dynamic point location by retroactivity

To change the subdivision that we are doing point location in:

Add or remove a pair of insert/delete operations in the timeline

Equivalent to adding or removing a line segment from the subdivision



Some history

Demaine et al. [2007]: Introduce retroactivity

Blelloch [2008]; Giyora and Kaplan [2009]:
Optimal retroactive binary search trees: $O(\log n)$ time per operation, $O(n)$ space; application to dynamic point location

Requires that search tree elements come from a single totally ordered sequence. Every two elements can be compared, even when they are not both in the tree at the same time as each other

Dickerson et al. [2010]:
Retroactive binary search trees, $O(\log n)$ time per update, $O(\log^2 n)$ time per query, $O(n)$ space, only comparing pairs of elements active at the same time

Application to information security: copy a Voronoi diagram given only access to it through post office queries

Example: Retroactive stack API

Push(t, x): Add a push(x) stack operation to the timeline at time t; return an identifier for the added operation

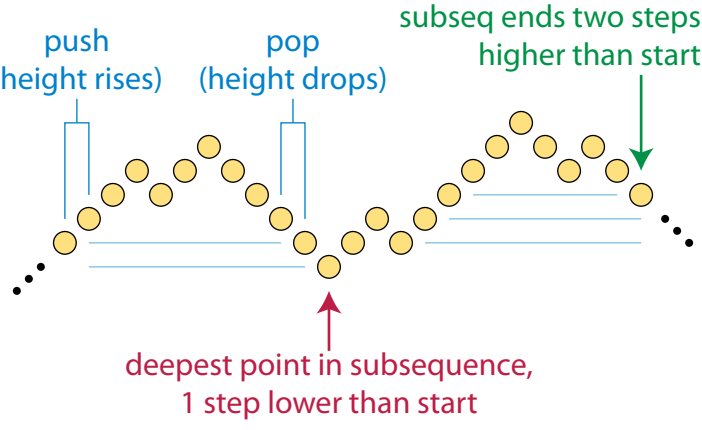
Pop(t): Add a pop stack operation to the timeline at time t and return its identifier

Undo(i): Remove the operation with identifier i from the timeline

Top(t): Return the item that, according to the current timeline, was at the top of the stack at time t

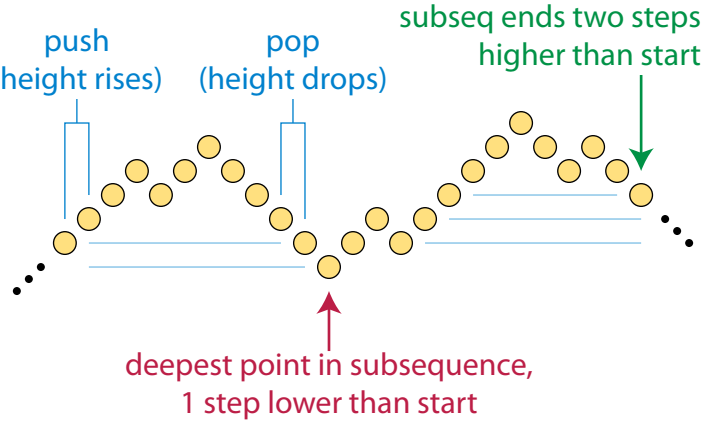
The time arguments can be numbers, or they can be positions in an ordered list (numbered using house numbers)

Visual analysis of a stack timeline



Update causes an earthquake! Landscape after update shifts up or down one unit

Visual analysis of a stack timeline



Yellow circle at time t shows the height of top(t)

It was pushed at the most recent earlier time the stack had the same height

Retroactive stack implementation

Binary search tree of operations in current timeline

Augmented so that each tree node i stores two numbers:

- g_i: How much the stack grows (positive) or shrinks (negative) during the sequence of operations in its subtree
  (total number of pushes minus total number of pops)
- d_i: Deepest point the stack is popped within the sequence of operations (relative to the starting level of the sequence)

Retroactive stack operations

Top(t):

- Search tree, adding g_i for subtrees to left of search path, to find stack size at time t
- Search again, using deepest points, to find the most recent earlier operation that started from a smaller stack size
- It must be a push and its argument is the element we want

Change timeline:

- Update g_i and d_i in augmented trees
  (in constant time at each ancestor of change, by combining information from its two children)
- Make sure that update does not cause deepest pop operation in whole tree to be at negative depth (this means that it is trying to pop more elements than were pushed)

Summary and references

Summary

- Definition of persistence and types of persistence
- Persistent stacks and application to programming language implementation
- Path-copying method for making treelike structures persistent
- Path-copying prefix sum and path-copying zippers
- Fat node method for making anything persistent
- Flat tree implementation of partially persistent fat nodes
- Hybrid persistence for efficient partially persistent binary search trees
- Locus method and point location using persistent search trees
- Retroactivity and retroactive stacks

References, I

Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20–22, 2008*, pages 894–903. SIAM, 2008.

Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(2):13, 2007. doi: 10.1145/1240233.1240236.

Matthew T. Dickerson, David Eppstein, and Michael T. Goodrich. Cloning Voronoi diagrams via retroactive data structures. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6–8, 2010, Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 362–373. Springer, 2010. doi: 10.1007/978-3-642-15775-2_31.

Yoav Giyora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3):28:1–28:51, 2009. doi: 10.1145/1541885.1541889.

References, II

Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16–18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi: 10.1109/SFCS.1978.3.

Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. Rank-balanced trees. *ACM Transactions on Algorithms*, 11(4):30:1–30:26, 2015. doi: 10.1145/2689412.