CS 261: Graduate Data Structures

Week 9: Time travel

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Persistent data structures

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

Driscoll, Sarnak, Sleator, and Tarjan,
"Making data structures persistent", STOC 1986

Analogy to version control

In a classical file system, each file exists only in its current form;
older versions of the file and previously deleted files
are no longer available

In a version-controlled file system (for example NILFS) or
version-control software (for example git, mercurial) each change
adds a new *version* to a file, but older versions can still be accessed

May be useful to have multiple versions of the same files
(development branch, release branch)

May be useful to go back into the history and look at old versions
(assign blame for new bugs; restore old pre-bug versions of code;
assist clients with the old versions they are still using)

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

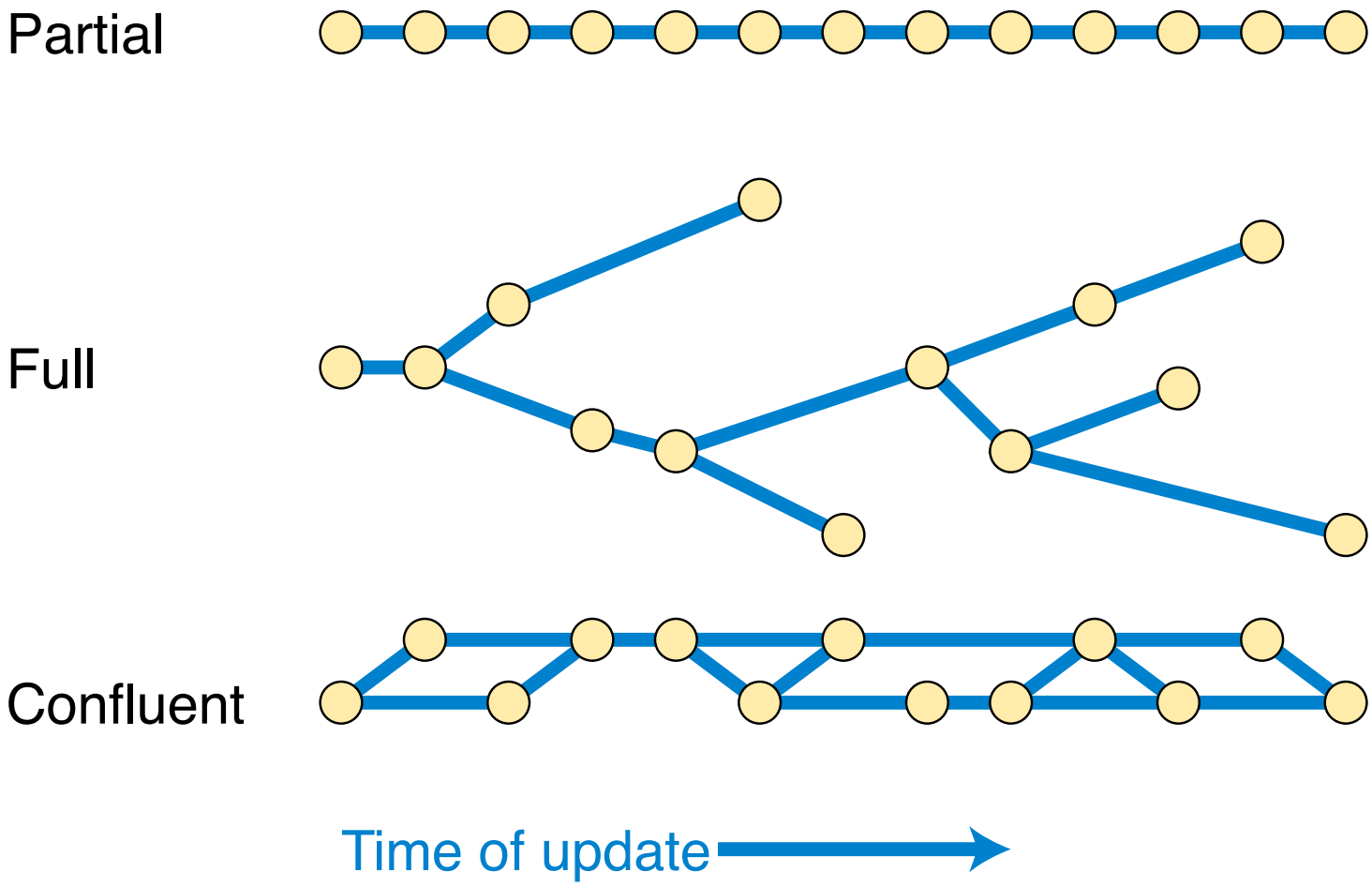
History forms a tree
(updating an old version creates a new branch)

Confluent persistence

Updates can combine multiple versions (like git merge)

History forms a directed acyclic graph

Visualization of version histories



Persistence versus amortization

We will design persistent data structures by building them on top of classical (non-persistent) data structures for the same problem

But, it generally does not work to build fully-persistent data structures out of classical structures that use amortized analysis

The reason: If there is a classical operation that is slow in actual time but fast in amortized time, then a fully persistent structure could be made to repeat that operation many times, by repeatedly going back to the version before it happened.

Its total actual time over the sequence of repetitions would be high, so its amortized time would also be high.

Persistence in API versus in representation

In a correctly designed persistent data structure, the *behavior* of old versions should never change: if a query on a given version has a given answer, it should always have that answer

The easiest way to achieve this is to make sure that the *data* stored in the old version, and accessed when we make a query, never changes

But it is also possible for some structures to change what they store, as long as the results of operations are unchanged.

Persistent stacks

Why?

In programming language implementation, stacks are used to represent information local to subroutine calls (local variables, return address, etc).

- Call a subroutine: push a block of local information onto the stack
- Return from a subroutine: pop its block from the stack
- Access local variables of surrounding scope: look on the stack

Sometimes, that information needs to persist even after the subroutine returns!

```
def outer(arguments):
    localvariable = something
    def inner(arguments):
        do something to localvariable
    return inner
```

Design principles for fully persistent API

Each operation needs to specify the version it applies to

The return value from each operation that changes the structure should be the new version that it creates

Therefore, we need to distinguish between two kinds of operations:

Updates change the structure, do not return information about it (instead, they return the version)

Queries return information about the structure, do not change it

API for fully persistent stacks

Updates:

- Create new empty stack and return its version
- Push new item x onto given version of stack and return the version of the updated stack
- Pop a non-empty version of stack and return the version of the updated stack, *not* the item that was popped

Queries:

- Return top element of given stack version
- Test whether given stack version is empty

Linked-list implementation

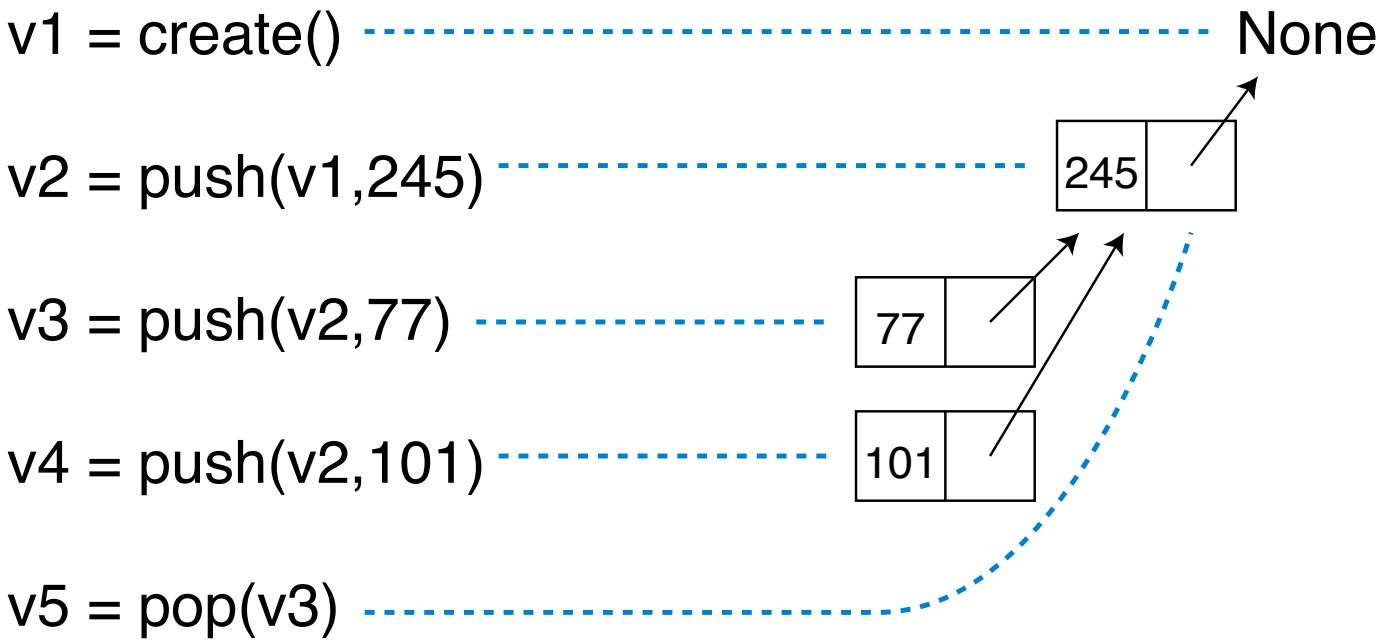
Empty version of stack = None

Non-empty version = pair (top element, version after popping it)

Operations:

- `def create(): return None`
- `def push(version,x): return (x,version)`
- `def pop((x,poppedversion)): return poppedversion`
- `def top((x,poppedversion)): return x`
- `def empty(version): return version == None`

Example



(Pop re-uses an old version rather than creating a new version of the data structure, but because we never change old versions, this re-use is safe and does not affect the results of the operations.)

Path copying

What is path copying?

A general technique for making some structures fully persistent

Works when:

- The data structure is built out of nodes of constant size
- Each node has pointers to a constant number of other nodes
- The main data structure is accessed through a constant number of pointers to root nodes
- Each node reachable by only one path of pointers from roots
- All operations access the data by following paths (no arrays!)

Examples:

- Linked-list based stacks (as we already saw)
- Tree-based structures with child pointers (but no parent pointers)

How to do path-copying

Represent each version as a pointers to its root node or tuple of pointers to its root nodes (as we did for stacks)

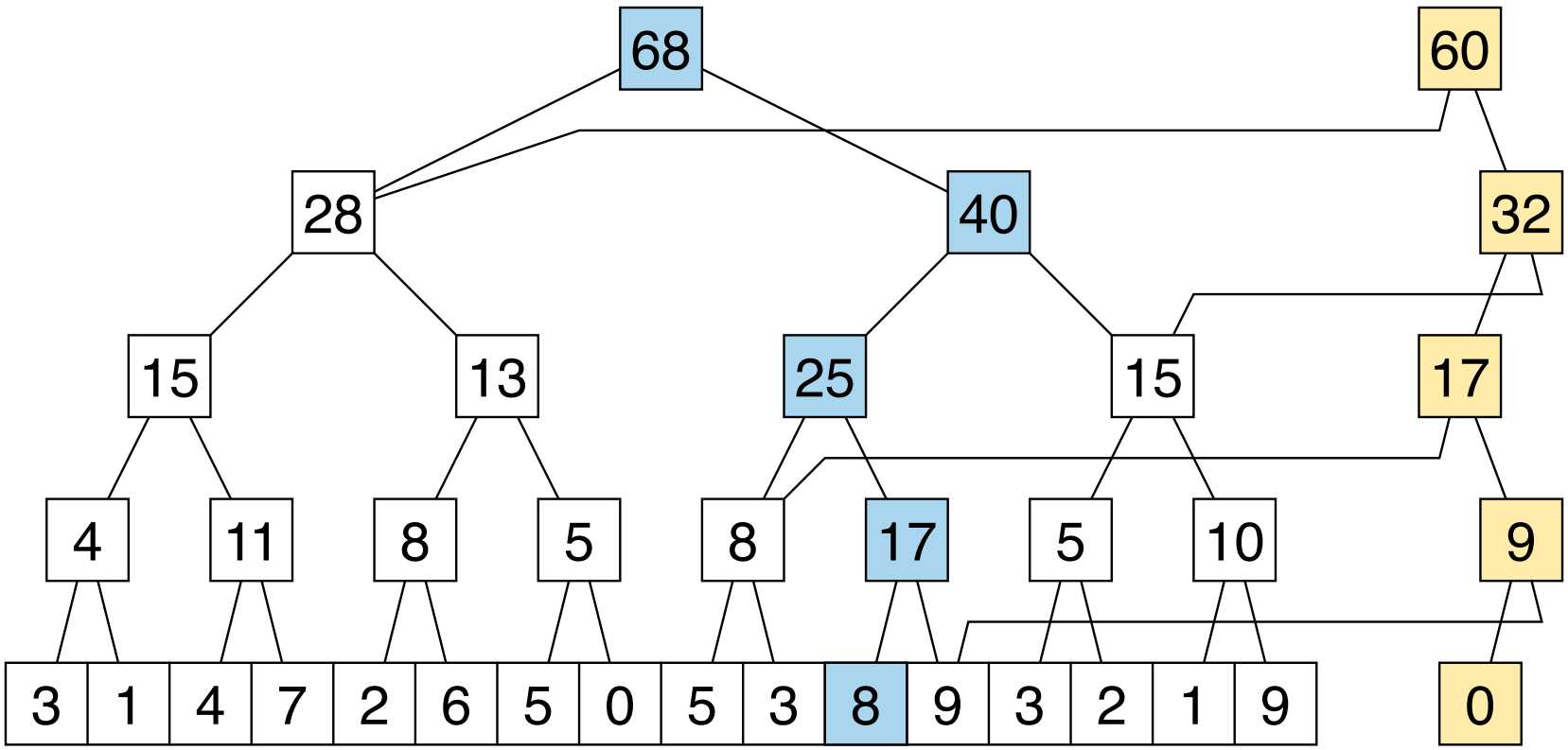
Perform each query exactly the same as you would in a non-persistent data structure, starting from the given root node or nodes

When a non-persistent update would change some nodes, make new copies of both the changed nodes and all of the other nodes on the paths that reached them

Example

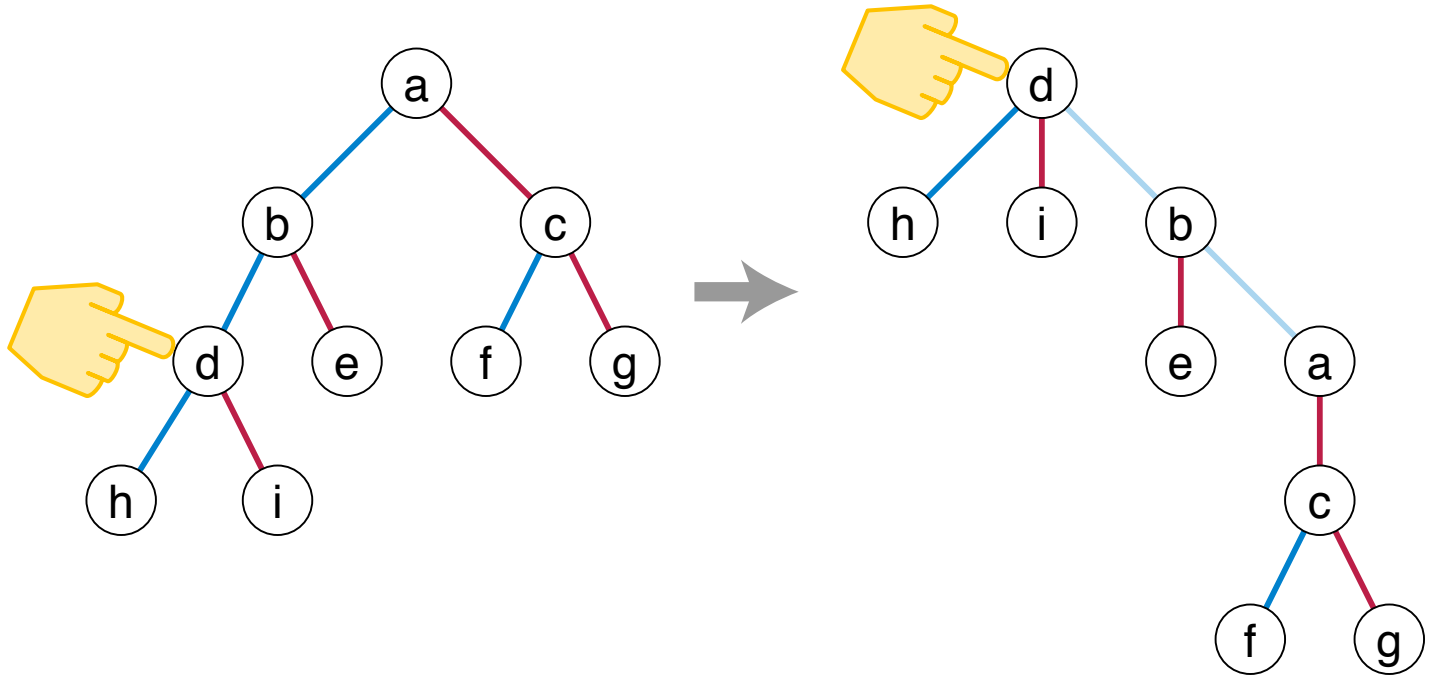
Prefix sum structure from week 7:

- Binary tree with data at leaves; each non-leaf stores sum of its two child values
- To update a given data value (here: 8 becomes 0), follow path down to it and make new copies of all nodes on the path



Finger trees / zippers

To represent a binary tree + a pointer to a vertex v, reverse all the edges between the tree root and v (but mark them differently from non-reversed edges so you can tell which ones were reversed)



Allows persistent movement of finger up/left/right and local changes to tree at finger, by path copying, in $O(1)$ time/update

In functional programming

In purely functional programming styles (e.g. in Haskell) data is immutable: once a piece of data has been created, it can be forgotten, but not changed

Path-copying is automatic (each change to data is really the creation of a new copy of an object), and all structures of linked data are automatically fully persistent

Zippers provide a convenient way to traverse and modify tree-based data srtuctures in a functional (and fully persistent) way

Path-copying analysis

Query time: Same as non-persistent structure

(because query is same as non-persistent structure)

Update time: Same as non-persistent structure

(extra work creating new nodes is proportional to the amount of time for non-persistent structure to reach the same set of nodes)

Space: Same as total update time

May be significantly bigger than non-persistent space

E.g. prefix-sum with n data values and n operations:
non-persistent $O(n)$, persistent $O(n \log n)$

Fat nodes

What are 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

Typically slower than path-copying because each access to a piece of memory turns into a more complicated data structure operation

More general (doesn't require nodes linked into paths from roots)

Optimal space (only enough to store all of the changes to the non-persistent structure)

Partially persistent fat nodes

In a partially persistent data structure, there is only one sequence of update operations

We can represent a version by a number, its position in the sequence

Each piece of memory stores a collection of key-value pairs

- Key = the number of an update operation
- Value = the value of that piece after that update

In an operation that wants to read or write version i of piece x, we need to find the predecessor of i in the pairs stored for x, or add a new pair (i, x)

Partially persistent fat node analysis

Space = total number of times a non-persistent data structure would write a piece of memory

(May be significantly smaller than total update time if most of the work in an update is reading not writing)

Time per operation =
(non-persistent time) $\times$ (time per predecessor operation)

- If local version-value pairs are stored in a binary search tree, then the time to access a piece of memory in a data structure with n updates is slowed down by $O(\log n)$ compared to non-persistent structure
- If they are stored in a flat tree (in a version of flat trees extended for predecessor queries) then the time to access a piece of memory is $O(\log \log n)$

Fully persistent fat nodes

In a fully persistent data structure, the history forms a tree

- Tree nodes = versions
- Parent of a version = the version it was updated from

Each update adds a new leaf to the tree

Each piece of memory stores a subset of versions (set of tree nodes); reading the piece of memory requires finding the nearest ancestor in this subset

The details of this nearest ancestor problem involve both maintaining order in lists and flat trees, but can be done with the same $O(\log \log n)$ slowdown as partial persistence.

Implementation

Because it does not depend on the details of the structure, fat-node persistence can be implemented generically, as a wrapper:

Implement your data structure normally, in a non-persistent way

Call a wrapper function to make a persistent version of it

Several Python implementation projects exist

But be careful: "persistence" can also mean that your data survives on disk from one execution of a program to another

Persistent binary search trees

Comparison of persistence techniques

For a partially persistent binary 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

- This is a kind of search tree where each node has an integer rank; constraints on ranks of neighboring nodes $\Rightarrow$ balanced
- 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 = $O(\log n) \times O(1) = O(\log n)$

The locus method

Method for building data structures for problems where:

Data does not change

Queries are points in the plane

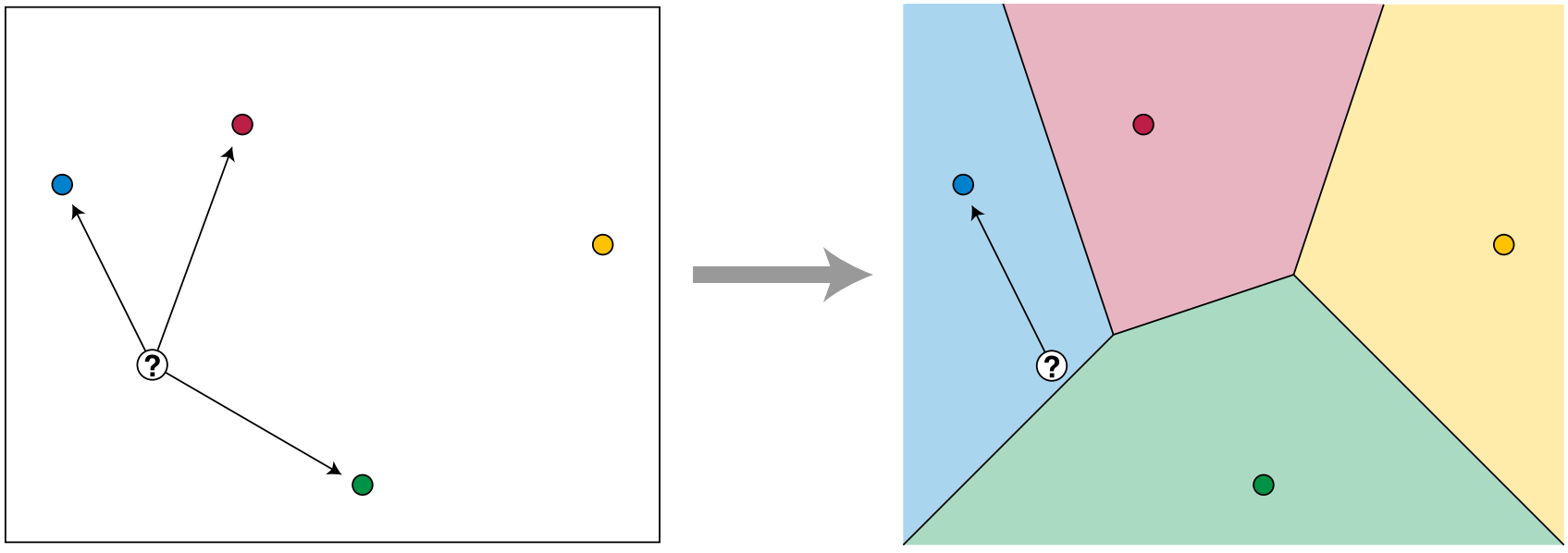
Answers are constant over regions of the plane
(rather than varying continuously)

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

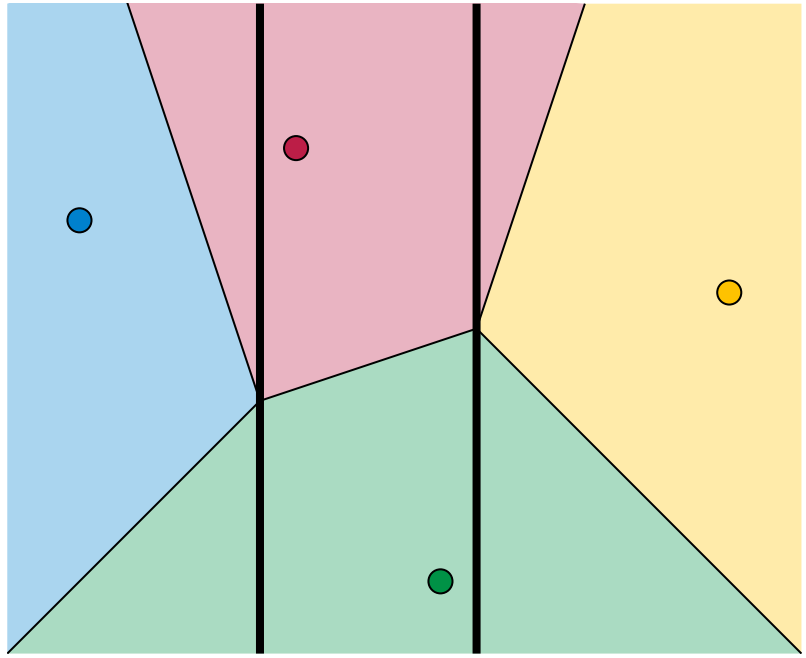
Point location by slabs

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 $O(n^2)$



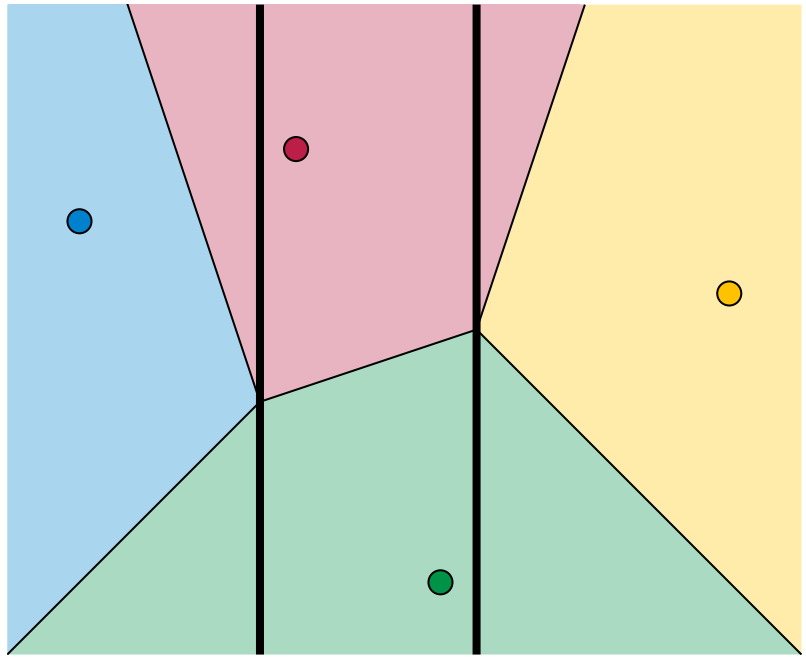
Point location by persistent search trees

Vertical ordering of regions changes only by removing a region (blue from left to middle slabs) or inserting a region (yellow from middle to right)

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

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



Retroactive data structures

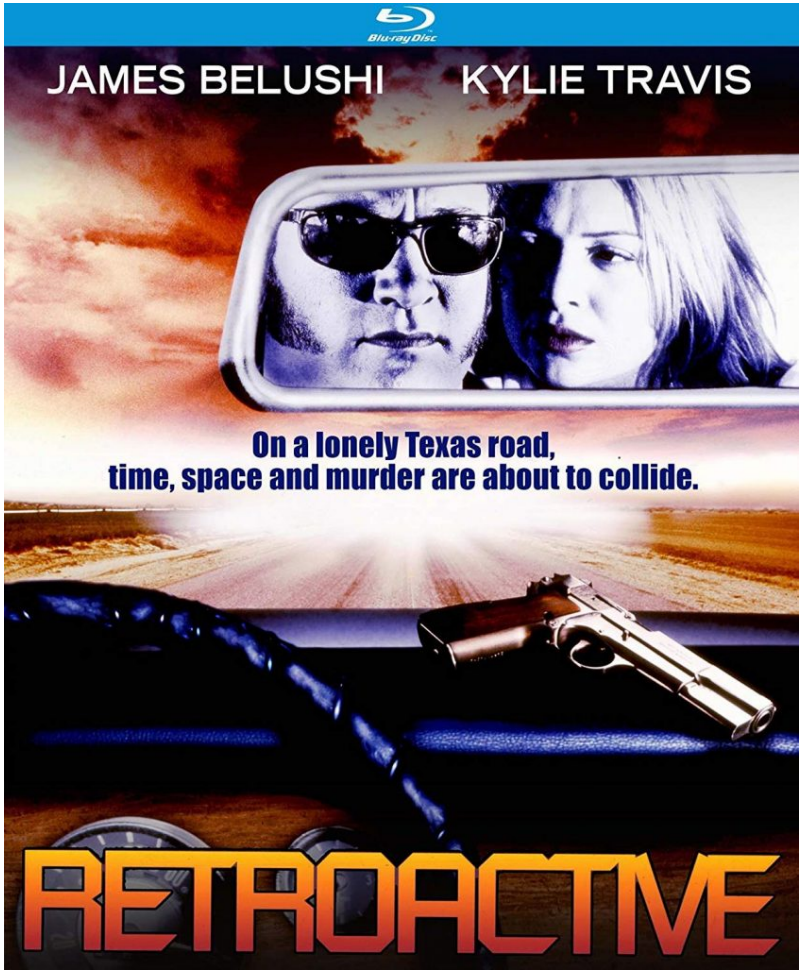
The main idea

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

Adding an operation to the timeline changes the history of the structure from its timestamp onwards

Queries are performed in the version of the data structure for their timestamp in current 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)

Some history

Demaine, Iacono, Langerman, SODA 2004: Introduce retroactivity

Blelloch, SODA 2008; Giora 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, Eppstein, and Goodrich, ESA 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

Fully 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

Retroactive stack implementation

Binary search tree of operations in current timeline

Augmented so that each tree node stores two numbers: $x - z$ and $y - z$ where x is starting stack size, y is minimum stack size, and z is ending stack size for its sequence of operations

Top(t):

- Search tree to find stack length at time t
- Search again 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

Updates: Use augmented tree to check that update does not cause more pops than pushes then add or remove operation from timeline and binary search tree

Summary

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