CS 261: Data Structures

Week 9: Time travel

Lecture 9a: Persistence

David Eppstein
University of California, Irvine

Spring Quarter, 2024

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 et al. 1989]

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

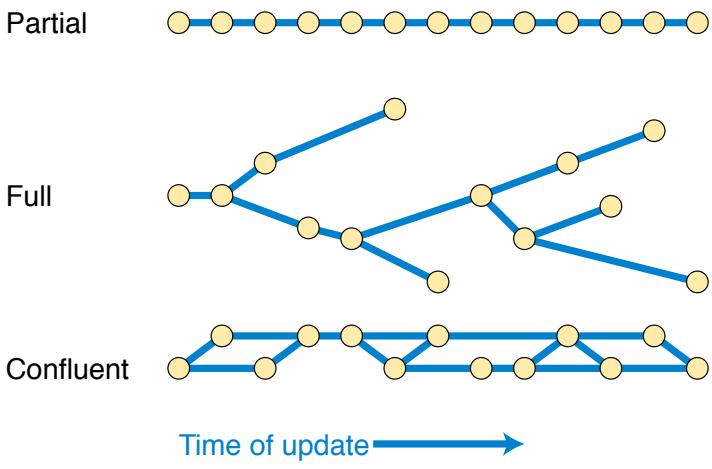
History forms a tree
(updating an old version creates a new branch)

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

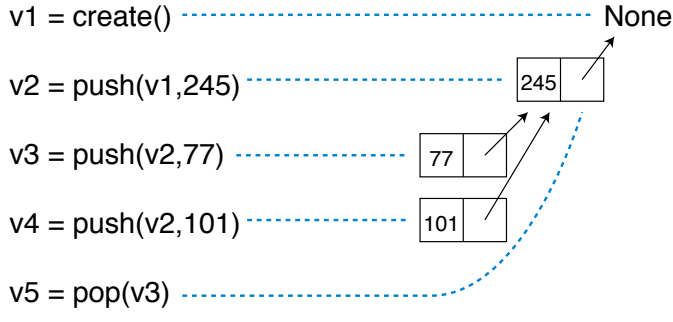
Empty version of stack = None

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)

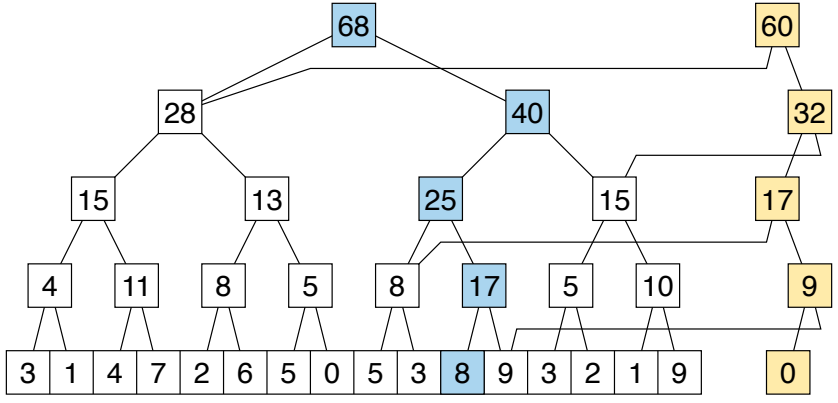
Perform each query exactly the same as you would in a non-persistent data structure, starting from the given root node or nodes

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:

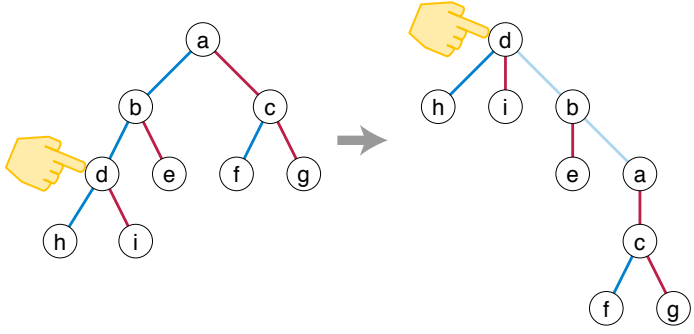
- Binary tree with data at leaves; each non-leaf stores sum of its two child values
- To update a given data value (here: 8 becomes 0), follow path down to it and make new copies of all nodes on the path



Finger trees / zippers

A representation of (binary) trees + a pointer to one specified vertex

Reverse edges between root and pointer (marked differently from forward edges)



Internal (non-persistent) representation): Only store links for downward edges

Each node has three pointers (left, right, downward parent)
+ one bit (if parent is non-empty, am I a left or right child

When a parent or child link goes upwards, leave its pointer empty!

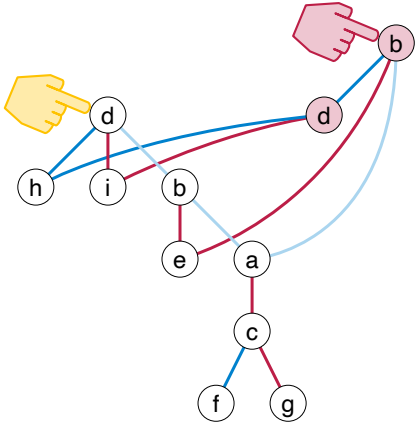
Path-copying navigation using zippers

Each time we move the finger, make new copies of changed nodes

To move across edge, create new versions of start and end vertex

Fully persistent!

Allows persistent movement of finger and local changes (like rotations) at finger in $O(1)$ time and space per 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 structures in a functional (and fully persistent) way

[Huet 1997]

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

[Pluquet et al. 2008]

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

References

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi: 10.1016/0022-0000(89)90034-2.

Gérard P. Huet. The Zipper. *The Journal of Functional Programming*, 7(5):549–554, 1997. doi: 10.1017/S0956796897002864.

Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In J. Ian Munro and Dorothea Wagner, editors, *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*, pages 37–48. SIAM, 2008. doi: 10.1137/1.9781611972887.4.