



Universidad Católica  
**San Pablo**

# TEMPORAL DATA STRUCTURES



# The pointer machine model of data structures

In this model we think of data structures as collections of nodes of a bounded size with entries for data. Each piece of data in the node can be either actual data, or a pointer to a node.

The primitive operations allowed in this model are:

1. `x = new Node()`
2. `x = y.field`
3. `x.field = y`
4. `x = y + z`, etc (i.e. data operations)
5. `destroy(x)` (if no other pointers to `x`)

Where `x`, `y`, `z` are names of nodes or fields in them.

Data structures implementable with these shape constraints and these operations includes linked lists and binary search trees, and in general corresponds to `struct`'s in C or objects in Java. An example of a data structure not in this group would be a structure of variable size such as an array.



There are two primary models of temporal data structures.

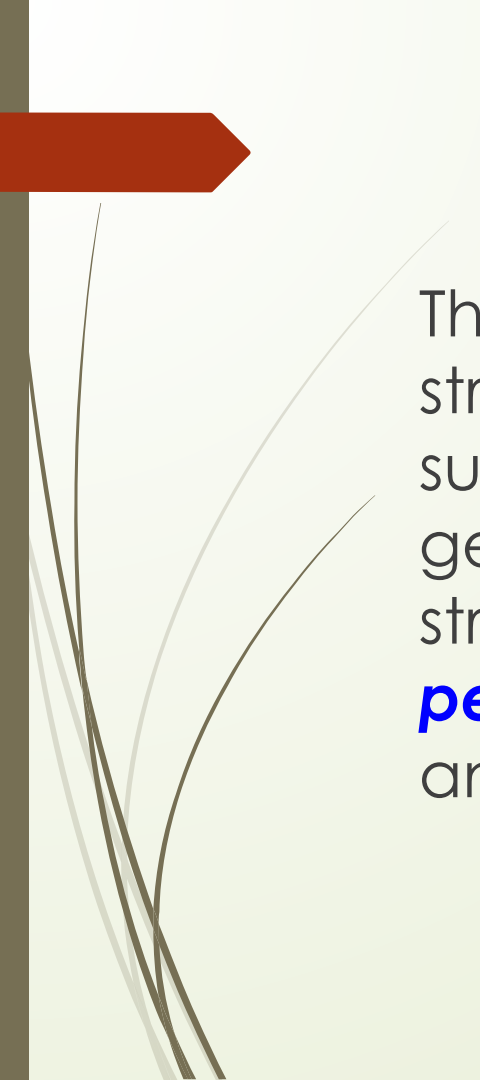
The first, called **persistence**, is based on the branching-universe model of time travel. In this model, going back in time and making changes creates a new branch of the data structure that differs from the original branch.

The second, called **retroactivity**, works on the idea of round-trip time travel. Here, a time traveller goes back in time, makes a change, and then returns to observe the effects of his or her change. This model gives us a there is a linear timeline with no branching.




# Persistence

based on the branching-universe model of time travel. In this model, going back in time and making changes creates a new branch of the data structure that differs from the original branch.



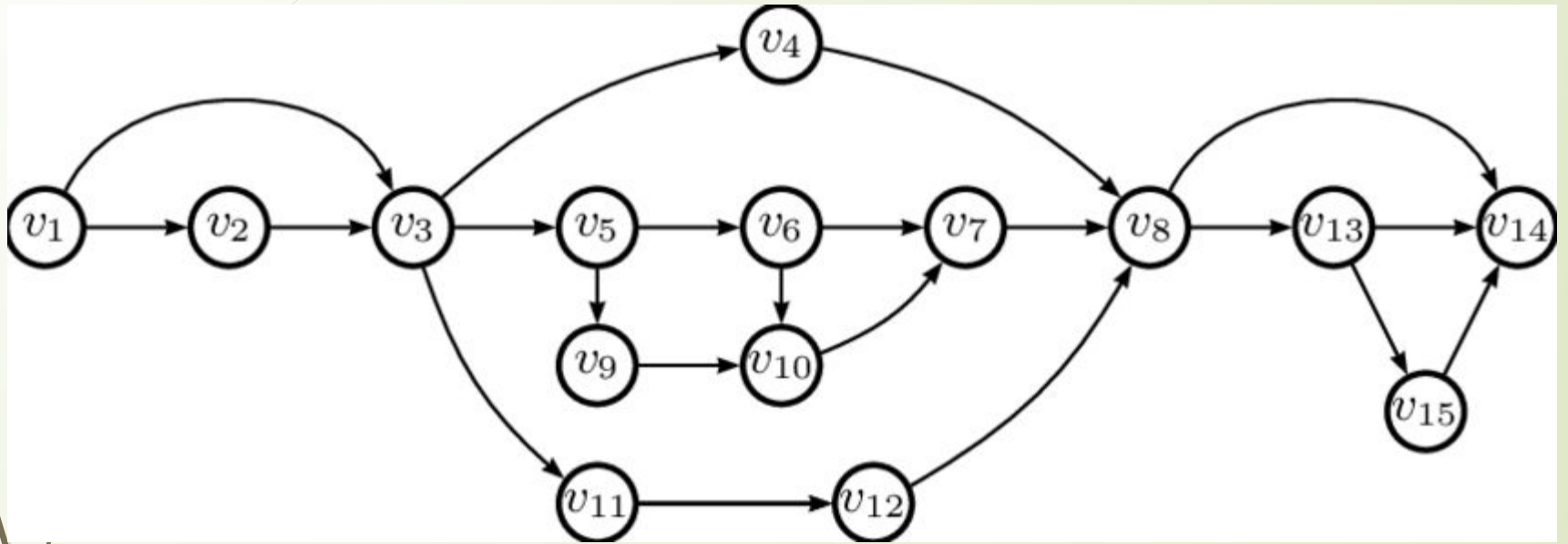
Think of the initial configuration of a data structure as **version zero**, and of every subsequent update operation as generating a new version of the data structure. Then a data structure is called ***persistent*** if it supports access to all versions and it is called ***ephemeral*** otherwise.

- 
- The obvious way to provide persistence is to make a copy of the data structure each time it is changed.
  - This has the drawback of requiring space and time proportional to the space occupied by the original data structure.
  - It turns out that we can achieve persistence with  $O(1)$  additional space and  $O(1)$  slowdown per operation for a broad class of data structures.



# Level of Persistence

1. Partial Persistence – In this persistence model, we may query any previous version of the data structure, but we may only update the latest version. This implies a linear ordering among the versions.
2. Full Persistence – In this model, both updates and queries are allowed on any version of the data structure. The versions here form of a branching tree.
3. Confluent Persistence – In this model, we use combinators to combine input of  $> 1$  previous versions to output a new single version. Rather than a branching tree, combinations of versions induce a DAG structure on the version graph.



Directed Acyclic Graph (DAG)



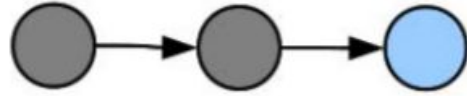
4. *Functional Persistence* – This model takes its name from functional programming where objects are immutable. The nodes in this model are likewise immutable: revisions do not alter the existing nodes in the data structure but create new ones instead. Okasaki discusses these as well as other functional data structures in his book [10].

The difference between functional persistence and the rest is we have to keep all the structures related to previous versions intact: the only allowed internal operation is to add new nodes. In the previous three cases we were allowed anything as long as we were able to implement the interface.

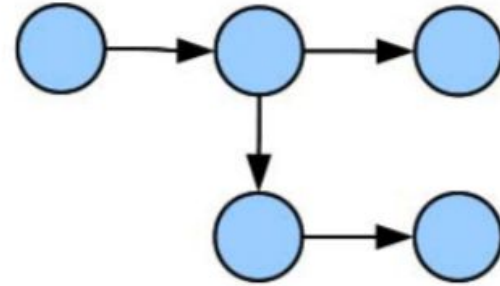
Each of the succeeding levels of persistence is stronger than the preceding ones. Functional implies confluent, confluent implies full, and full implies partial.

Functional implies confluent because we are simply restricting ways on how we implement persistence. Confluent persistence becomes full persistence if we restrict ourselves to not use combinators. And full persistence becomes partial when we restrict ourselves to only write to the latest version.

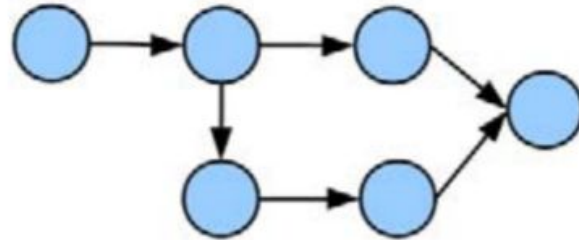
[10] Chris Okasaki: *Purely Functional Data Structures*. New York: Cambridge University Press, 2003.




(a) Partial.



(b) Full



(c) Confluent/ Functional



# General techniques for making data structures persistent

- Fat nodes
- Path Copying
- Modification Box



# Fat Nodes

One natural way to make a data structure persistent is to add a modification history to every node. Thus, each node knows what its value was at any previous point in time. (For a fully persistent structure, each node would hold a version tree, not just a version history.)

This simple technique requires  $O(1)$  space for every modification: we just need to store the new data. Likewise, each modification takes  $O(1)$  additional time to store the modification at the end of the modification history. (This is an amortized time bound, assuming we store the modification history in a growable array. A fully persistent data structure would add  $O(\log m)$  time to every modification, since the version history would have to be kept in a tree of some kind.)



Extracted from

<https://ocw.mit.edu/courses/6-854j-advanced-algorithms-fall-2005/resources/persistent/>



# 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  $(i, x)$



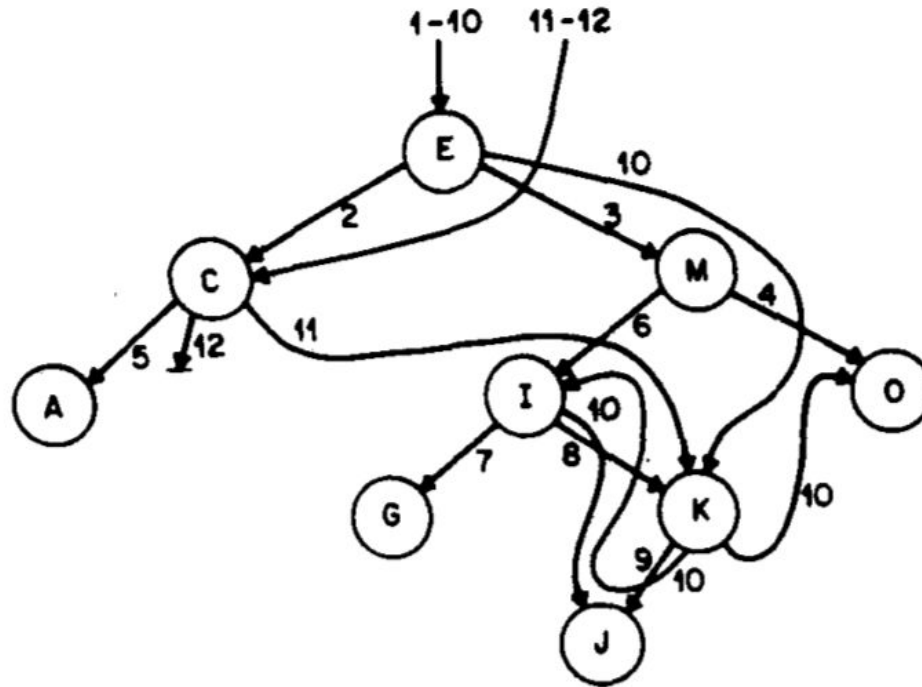


FIG. 1. A partially persistent search tree built using the fat node method, for the sequence of update operations consisting of insertions of *E*, *C*, *M*, *O*, *A*, *I*, *G*, *K*, *J*, followed by deletion of *M*, *E*, and *A*. The “extra” pointers are labeled with their version stamps. Left pointers leave the left sides of nodes and right pointers leave the right sides. The version stamps and original null pointers of nodes are omitted, since they are unnecessary.



# 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.



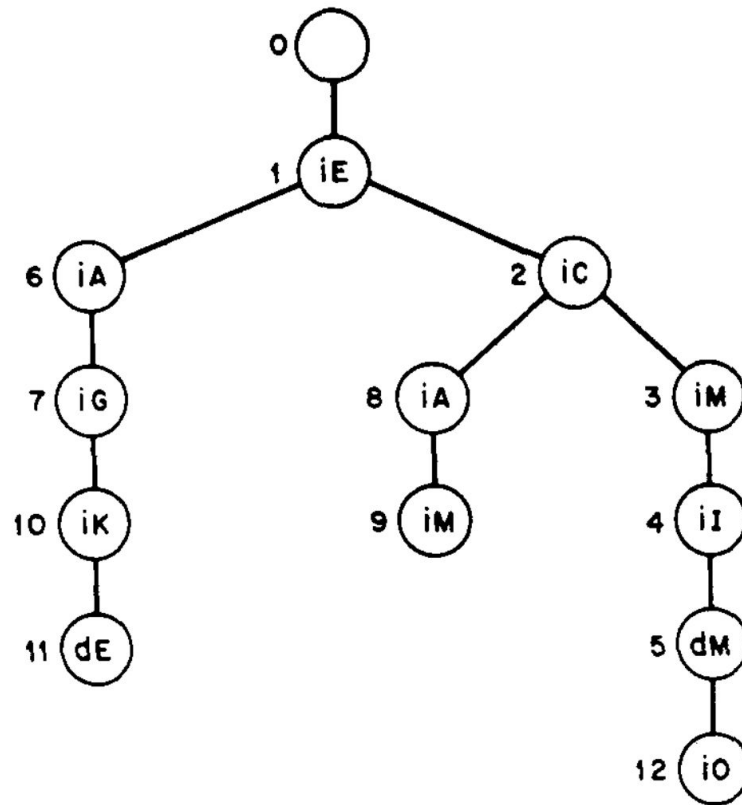


FIG. 3. A version tree. Each node represents an update operation; an “i” or “d” indicates an insertion or deletion of the specified item. The nodes are labeled with the indices of the corresponding operations. The version list is 1, 6, 7, 10, 11, 2, 8, 9, 3, 4, 5, 12.

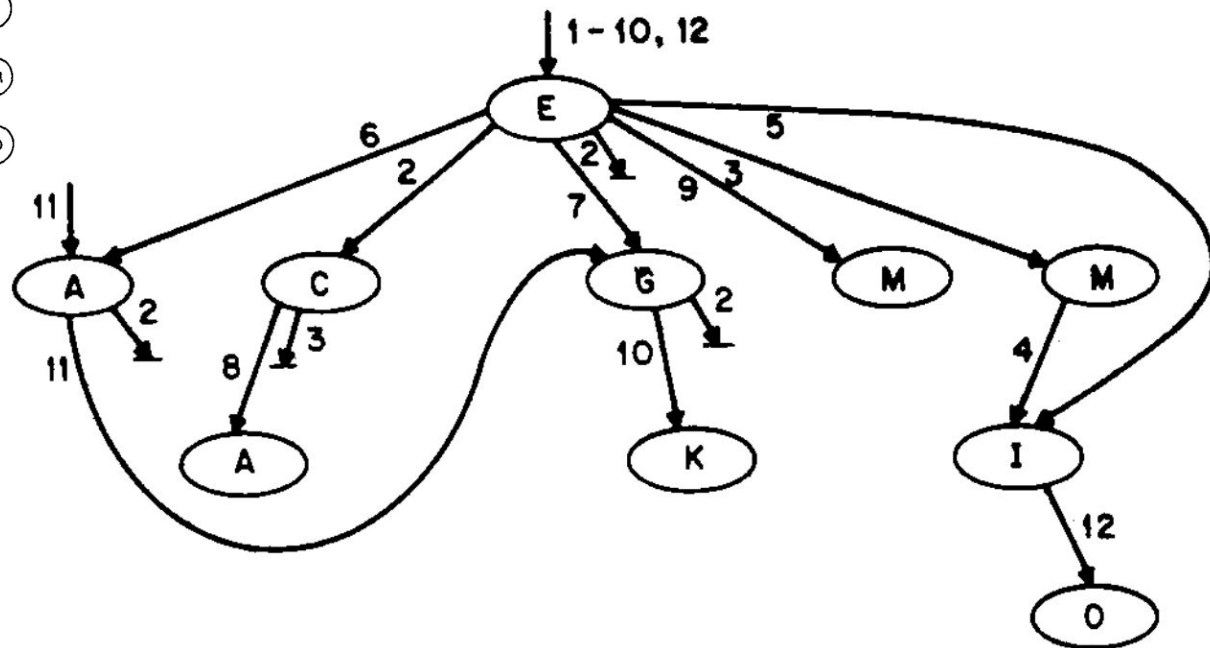
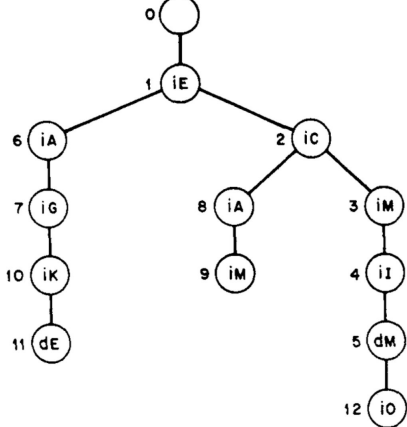
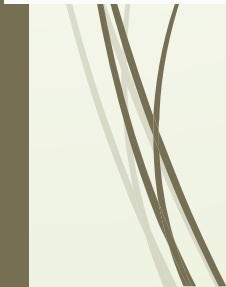


FIG. 4. A fully persistent search tree built using the fat node method, for the update sequence in Fig. 3. The version stamps and original null pointers of nodes are omitted as unnecessary.



## Path Copying

Path copy is to make a copy of all nodes on the path which contains the node we are about to insert or delete. Then we must cascade the change back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until we reach to the root. We maintain an array of roots indexed by timestamp. The data structure pointed to by time  $t$ 's root is exactly time  $t$ 's data structure.





# 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)

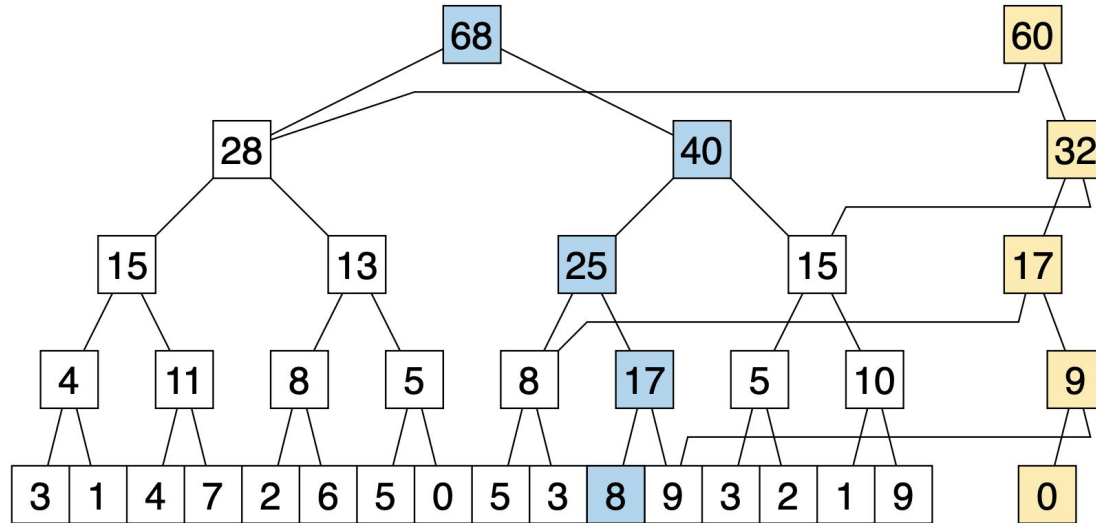
Perform each query exactly the same as you would in a non-persistent data structure, starting from the given root node or nodes

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





# 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  $O(n \log n)$



## 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.






# Modification Box

Extracted from

<https://ocw.mit.edu/courses/6-854j-advanced-algorithms-fall-2005/resources/persistent/>



Sleator, Tarjan et al. came up with a way to combine the advantages of fat nodes and path copying, getting  $O(1)$  access slowdown and  $O(1)$  modification space and time. Here's how they did it, in the special case of trees.

In each node, we store one *modification box*. This box can hold one modification to the node—either a modification to one of the pointers, or to the node's key, or to some other piece of node-specific data—and a timestamp for when that modification was applied. Initially, every node's modification box is empty.

Whenever we access a node, we check the modification box, and compare its timestamp against the access time. (The access time specifies the version of the data structure that we care about.) If the modification box is empty, or the access time is *before* the modification time, then we ignore the modification box and just deal with the normal part of the node. On the other hand, if the access time is *after* the modification time, then we use the value in the modification box, overriding that value in the node. (Say the modification box has a new **left** pointer. Then we'll use it instead of the normal **left** pointer, but we'll still use the normal **right** pointer.)



Modifying a node works like this. (We assume that each modification touches one pointer or similar field.) If the node's modification box is empty, then we fill it with the modification. Otherwise, the modification box is full. We make a copy of the node, but *using only the latest values*. (That is, we overwrite one of the node's fields with the value that was stored in the modification box.) Then we perform the modification directly on the new node, without using the modification box. (We overwrite one of the new node's fields, and its modification box stays empty.) Finally, we cascade this change to the node's parent, just like path copying. (This may involve filling the parent's modification box, or making a copy of the parent recursively. If the node has no parent—it's the root—we add the new root to a sorted array of roots.)



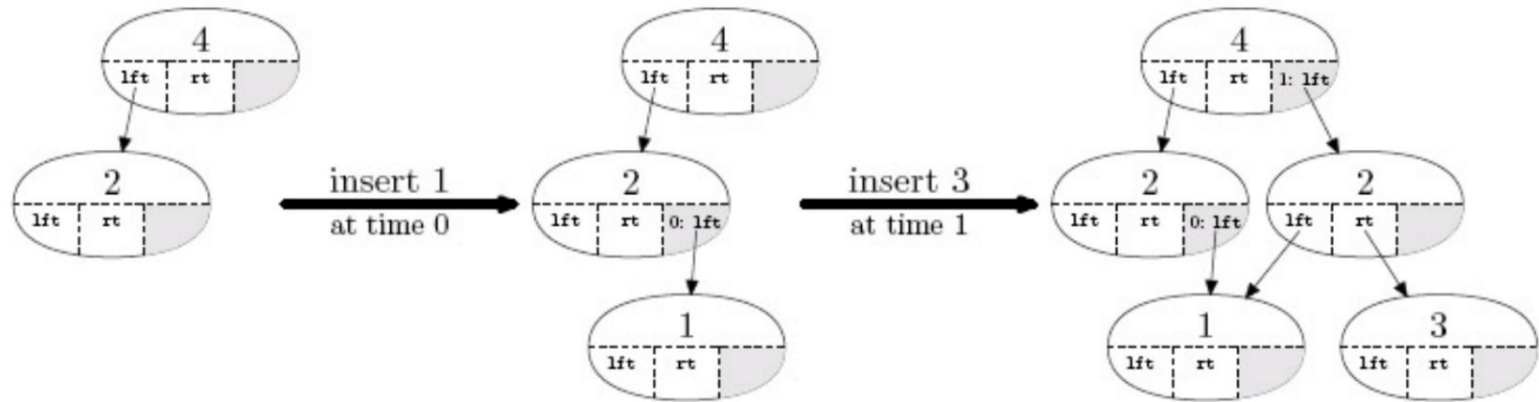


Figure 2: Persistence of a binary tree. We need a modification box the size of the in-degree of each data structure node (just one for trees).

With this algorithm, given any time  $t$ , at most one modification box exists in the data structure with time  $t$ . Thus, a modification at time  $t$  splits the tree into three parts: one part contains the data from before time  $t$ , one part contains the data from after time  $t$ , and one part was unaffected by the modification.

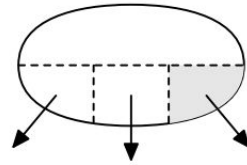


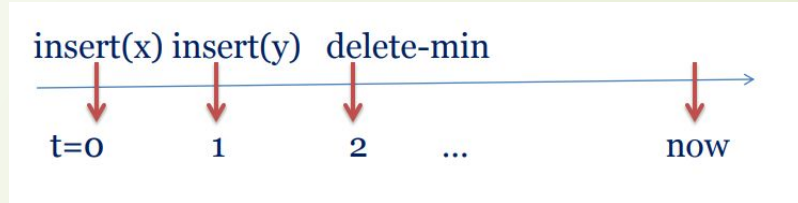
Figure 2.4: How modifications split the tree on time.

How about time bounds? Well, access time gets an  $O(1)$  slowdown (plus an additive  $O(\log m)$  cost for finding the correct root), just as we'd hoped! (We must check the modification box on each node we access, but that's it.)



# Retroactivity

Una estructura de datos en el que las operaciones pueden realizarse sobre una estructura de datos no sólo en el presente sino también en el pasado.



The following operations are supported by retroactive DS:

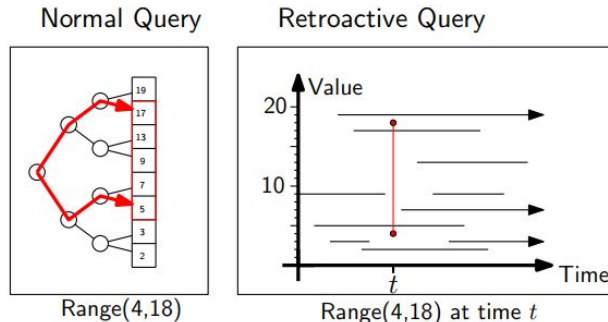
- $Insert(t, update)$  - inserts operation “update” at time  $t$
- $Delete(t)$  - deletes the operation at time  $t$
- $Query(t, query)$  - queries the DS with a “query” at time  $t$

Uppercase Insert indicates an operation on retroactive DS, lowercase update is the operation on the actual DS.

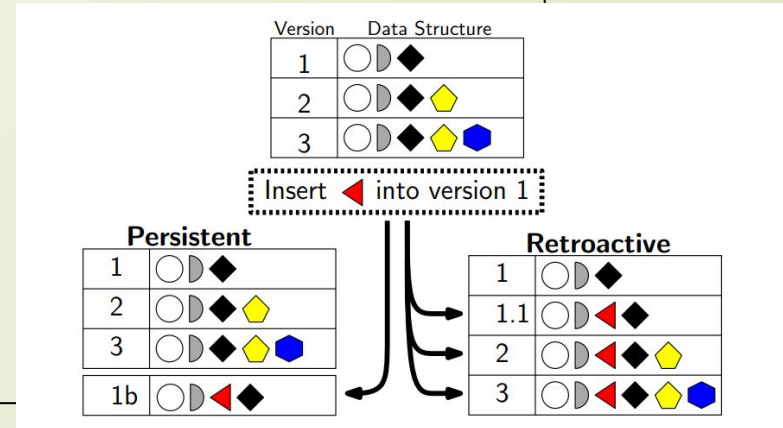


# Persistencia VS Retroactividad

- Las estructuras de datos persistentes, cada versión se trata como un archivo inalterable.
  - Versiones existentes nunca se modifican.
  - Las operaciones de la estructura de datos se realizan en relación a un especificado versión.
  - La actualización crea y devuelve una nueva versión de estructuras de datos. Nunca modifica la versión existente
- ❖ Por el contrario, el modelo retroactivo puede afectar radicalmente al contenido de todas las versiones posteriores




[2] Gráfico que representa un rango , consulta normal y consulta retroactiva




[2] Gráfico ,diferencia entre persistencia y retroactividad





There are three types of retroactivity:

- *Partial* - Query always done at  $t = \infty$  (now)
  - *Full* - Query at any time  $t$  (possibly in the past)
  - *Nonoblivious* - Insert, Delete, Query at any time  $t$ , also if an operation modifies DS, we must say which future queries are changed.
- 



# Main strategies

- 
1. Roll-back Method
  2. **Priority Queue Method**

## Roll back method:

- write down a (linear) chain of operations and queries
- change  $r$  time units in the past with factor  $O(r)$  overhead.

That's the best we can do in general.

Lower bound:  $\Omega(r)$  overhead necessary.


Proof: Data Structure maintains 2 values (registers):  $X$  and  $Y$ , initially  $\emptyset$ . The following operations are supported:  $X = x$ ,  $Y+ = \Delta$ ,  $Y = X.Y$ , query ' $Y?$ '. Perform the following operations (Cramer's rule):

$$Y+ = a_n, X = X.Y, Y+ = a_{n-1}, X = X.Y, \dots, Y+ = a_0$$

which is equivalent to computing

$$Y = a_n X^n + a_{n-1} X^{n-1} + \dots + a_0$$

Now, execute  $\text{Insert}(t = 0, X = x)$ , which changes where the polynomial is evaluated. This cannot be done faster than re-evaluating the polynomial. In history-independent algebraic decision tree, for any field, independent of pre-processing of the coefficients, need  $\Omega(n)$  field operations (result from 2001), where  $n$  is the degree of the polynomial.



## Priority Queues

Now, let us move onto some more positive results. Priority queues represent a DS where retroactive operations potentially create chain reactions but we have still obtained some nice results for. The main operations are *insert* and *delete-min* which we would like to retroactively *Insert* and *Delete*.

a **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

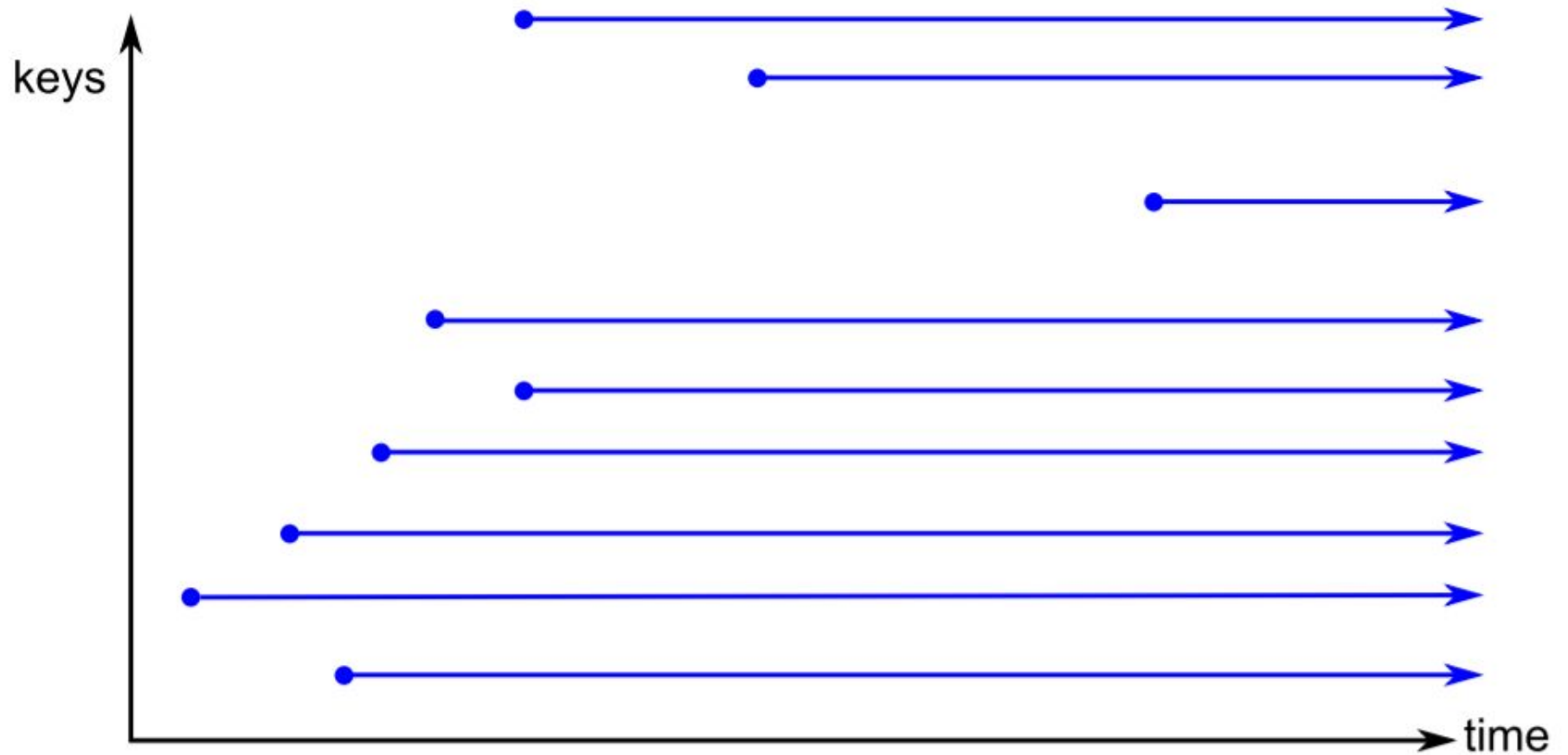


Figure 2: Graph of priority queue featuring only a set of *inserts*

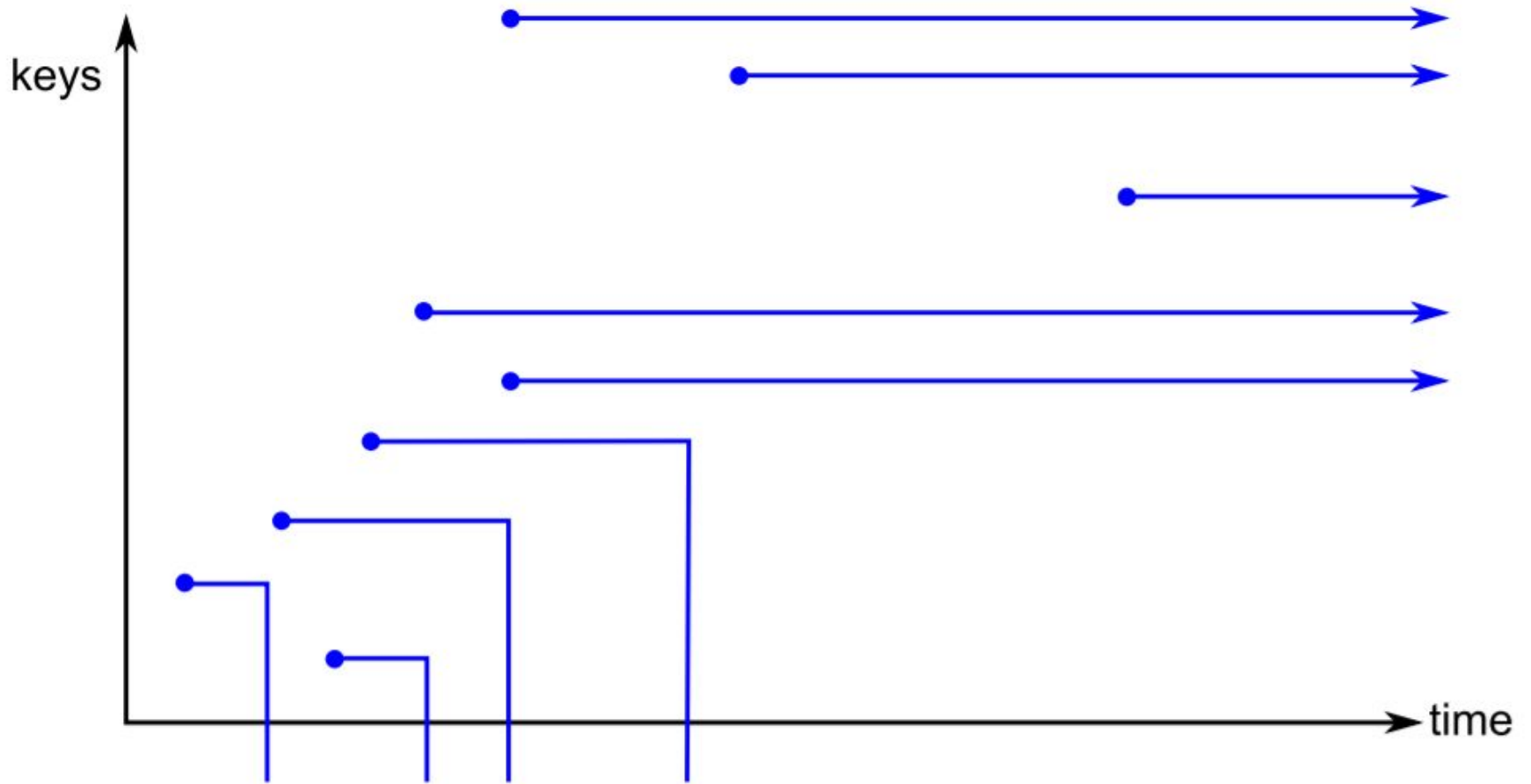


Figure 3: Adding *del-min()* operations leads to these upside down “L” shapes.

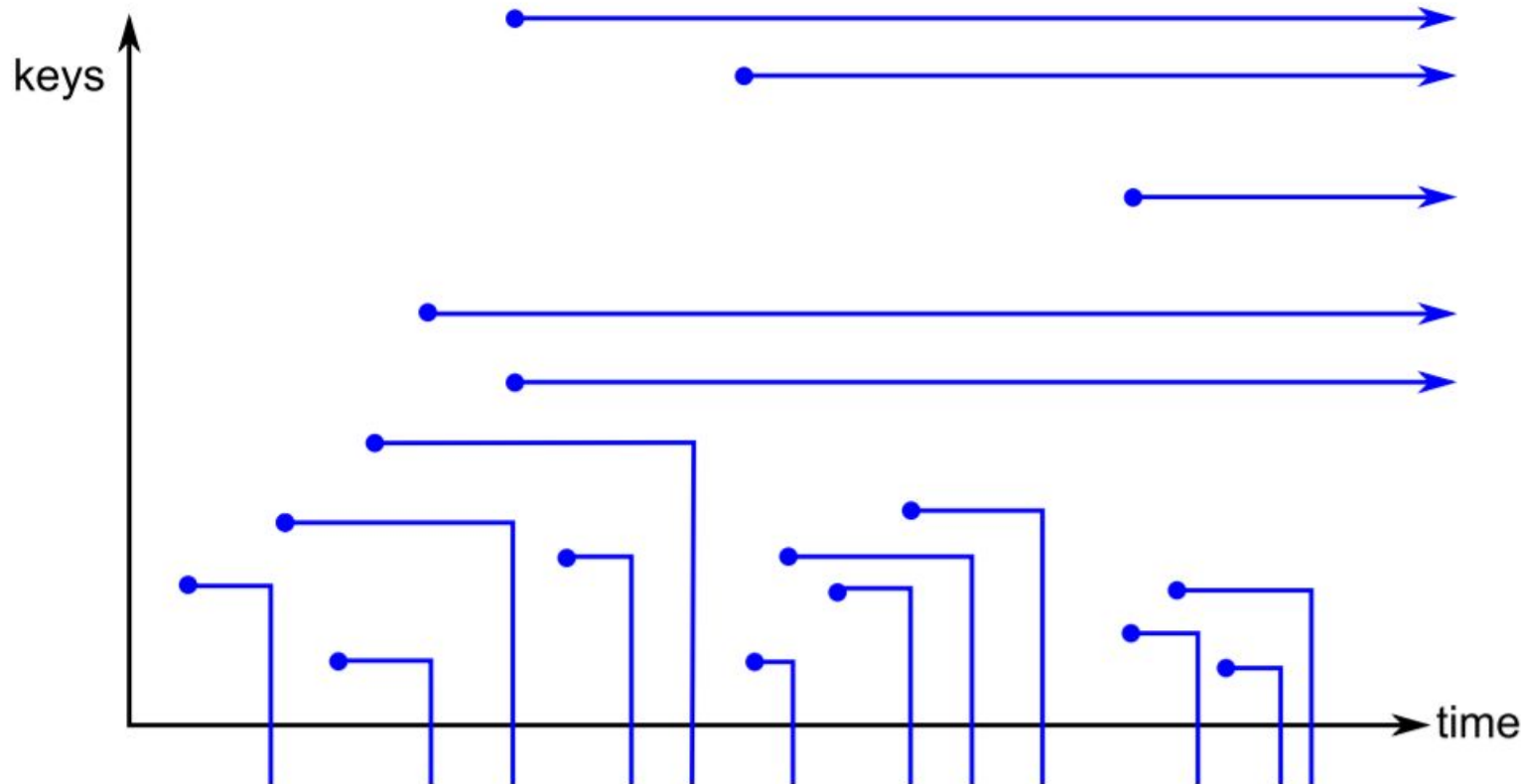


Figure 4: An L view representation of a priority queue with a more complicated set of updates

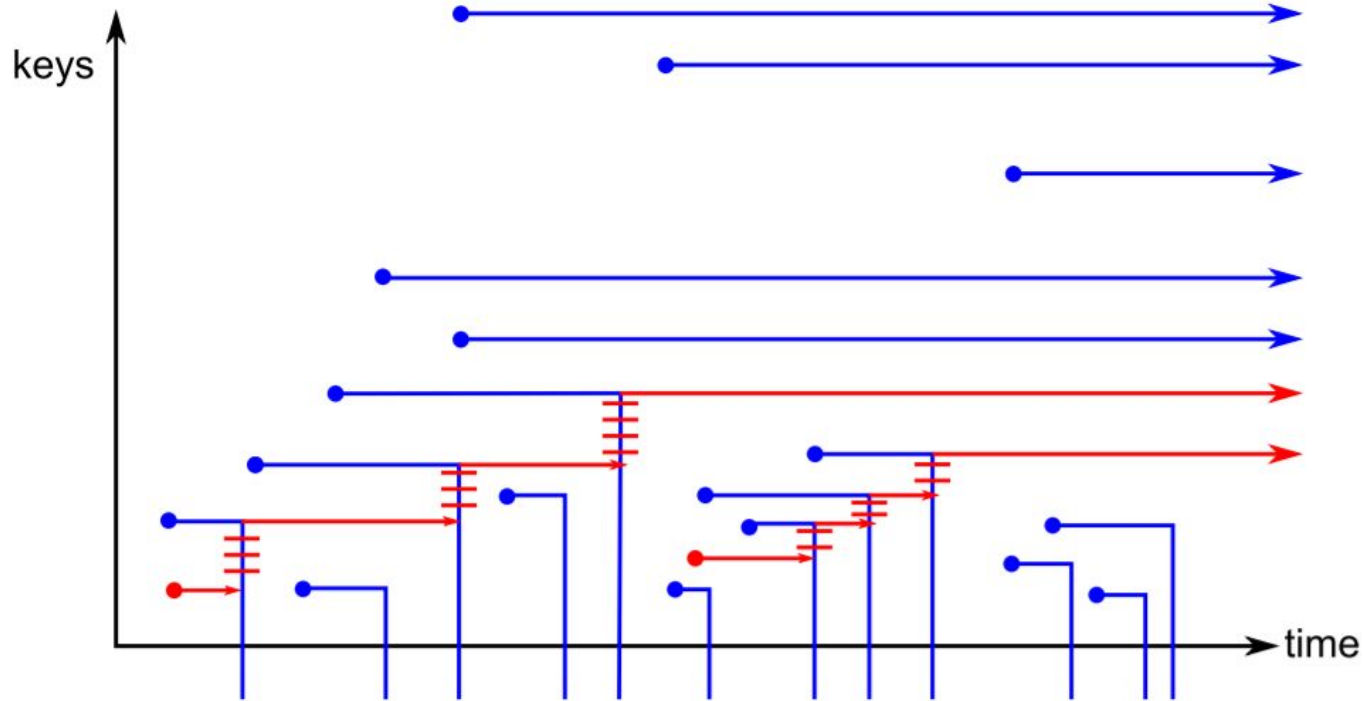


Figure 5: Retroactive inserts start at red dots and cause subsequent *delete-min* operations to effect different elements as shown.





# 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  $t$



# REFERENCIAS:



[1] Retroactive Data Structures, Erik d. demaine and John lacono and Stefan Langerman.

[2] Retroactive Data Structures Michael T. Goodrich, Joseph A. Simons  
Department of Computer Science, University of California, Irvine.

# Partially persistent Fat Nodes 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  $O(\log \log n)$

