ETH ZURICH, DEPARTMENT INFORMATIK

Making Data Structures Persistent

Shengyu Huang

November 13, 2018

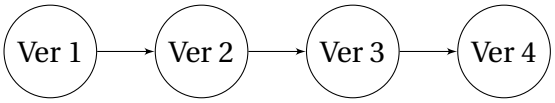
1 INTRODUCTION

In most of the data structures we see, if we do modifications, we will not be able to trace back the old versions and have only access to the latest version. We call such data structures *ephemeral*. But think about text and file editing, we can redo and undo things because we have a copy of the working history. So being able to go back in time could be desirable for a data structure. *Persistent data structures* are about maintaining history. If we take the initial configuration of a data structure as version zero (which we assume to be empty here), an operation that changes the state of this data structure generates version one. That is, if we index all the modifications, a modification operation `i` produces version `i`.

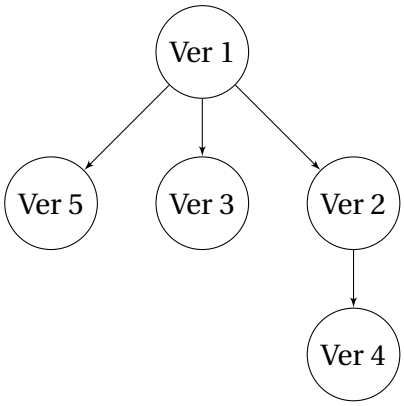
If we can only view the working history but have no way to change the data structure on the old versions, we call it partial persistence. That is, a partially persistent data structure supports access to all versions, but only the newest version can be modified.

If the data structure furthermore allows us to act on the old versions, we call it a *fully persistent* data structure. Fully persistent data structures allow us to both access and update all available versions.

By visualizing the working history of persistent data structures, we can gain some useful intuition. Versions in partially persistent data structures can be represented by a path, while versions in fully persistent data structures can be seen as forming a tree.



(a) Partial persistence forms a version path



(b) Full persistence forms a version tree

Figure 1.1: Visualization of versions in persistent data structures

In this report, we will introduce two generic techniques to make linked structures persistent at small cost in time and space. First, we need to formally define a few terms that will run through the whole report.

Definition. A *linked data structure* is a finite collection of nodes, each containing a fixed number of named fields. Each field is either an information field that holds a single piece of information of a specified type, or a pointer field that holds a pointer to a node or the special value *nil* indicating no node.

Stacks, queues, and trees are common linked structures, while arrays are not. We can think of a linked structure as a labeled directed graph where the out-degree of vertices is upper bounded. In addition, access to a linked structure is provided by a fixed number of named *access pointers* indicating nodes of the structure, called *entry nodes*. In the case of binary search trees, we have only one access pointer, the *root pointer*, i.e. the pointer to the root of the tree, and the root being the only entry node.

We are going to use AVL trees as our example, but the techniques we discuss are applicable to all linked data structures. An AVL tree is a binary tree where the height difference of two child subtrees is no more than one for all nodes. We store a *balance factor* in each node to help us maintain balance of the tree.

Definition. In a binary tree, the *balance factor* of a node `N` is defined to be the height difference of its two child subtrees, i.e.

```
BalanceFactor(N) := Height(RightSubtree(N)) - Height(LeftSubtree(N))
```

We consider three kinds of operations for AVL trees: searching, insertion, and deletion. We call insertion and deletion update operations. Searching in an AVL tree can be done the same way as in a normal binary search tree. Insertion and deletion

in an AVL tree consists of two stages: the first stage is the same as doing insertion or deletion in a binary search tree. The second stage is rebalancing, which could change the pointer fields and balance factors of nodes when performing single or double rotation.

We can think of a persistent AVL tree as a linked structure with each version of the ephemeral AVL tree embedded in it, so that each searching, insertion, or deletion in a version of the ephemeral AVL tree can be simulated (ideally in constant time) in the corresponding part of the persistent AVL tree. The problem we want to solve is to maintain the correct correspondence between the persistent AVL tree and the ephemeral AVL tree.

2 Fat Node Method

2.1 Partial Persistence

Let `n` be the number of nodes in the AVL tree in a certain version. Let m be the number of update operations. A very naive way to achieve partial persistence is to store every version entirely after each update. This costs $\Omega(n)$ time and space per update. An alternative method is to only store the sequence of update operations, and rebuild the ephemeral data structure of a certain version from scratch for every query. Let m be the number of update operations. This method takes $O(m)$ space, but each access to version i takes $\Omega(i)$ time even if every single update takes only $O(1)$ time.

Instead of storing the update operations in a separate sequence, we let each node keep track of its own history. Updates on a single node could make the history information of this node arbitrarily large, hence the name "fat node".

To be more precise, each fat node in the persistent AVL tree will contain the same fields as a node in the ephemeral structure (a key, a balance factor and two pointers), along with space for an arbitrary number of extra field values. Each extra field value has an associated field name (key, left, right, or balance) and a version stamp.

We simulate ephemeral an update operation as follows. Consider the insertions of nodes x, y, z consecutively. This sequence of update operations produces three versions of the AVL tree. The persistent AVL tree using fat node method is shown in Figure 2.1.

Insert `x` is just setting the root pointer to `x` in version one. In addition, we need to initialize the pointers of `x` as `nil` and the balance factor as 0. When we insert `y` to `x`'s right pointer, we store `(right, y, v2)` and `(balance, 1, v2)` to node `x`. However, we can actually overwrite `x.right` and `x.balance` in version one without worrying losing any information. Because the initial values of two pointers of a node are always `nil` and the initial value of balance factor is always 0. So

if we want to query the field value of a node in some version, but this version is smaller than the smallest version stamp in the queried field, we immediately know the value is `nil` or 0. In this case, we know `x.right` in version one is `nil` because the smallest version stamp in `x.right` is 2.

Similarly, when we insert `z` to `y.right`, we store `(right, z, 3)` into node `y`. Now node `x` has a balance factor of 2 after the insertion. By looking at its child node `y`'s balance factor, i.e. 1, we know we only need to perform a single left rotation.

After this rotation, fields `y.left, x.right, x.balance, y.balance` are changed. If a node already has a field value with version stamp 3, we overwrite it with the value in the final state. This is because we do not need the intermediate values during rebalancing. If some field value is changed but has a version stamp smaller than 3, we store this new information in the node. Here, we store `(right, nil, 3), (left, x, 3)` to node `x` and `y` respectively.

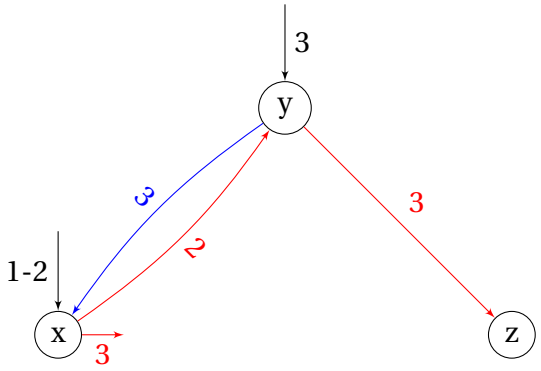


Figure 2.1: Persistent representation of an AVL tree after insertions of x, y, z using fat node method. Pointers are labeled with their version stamps. Red edges are right pointers and blue edges are left pointers.

In general, we need store `(field name, field value, version stamp)` when a field value is changed, and we also need an additional structure to help us keep track of root pointers of all versions in order to initialize the access into a version.

Simulating searching in the persistent AVL tree is as follows. Starting from the root with version stamp `i`, we compare the key with the one in current node we are visiting. If the search key is larger than the key of the current node, we look for all right pointers in the corresponding fat node by going down with the one that has maximum version stamp no greater than `i`. For example, if we want to know the value of `y.left` in version 2, we go from the root pointer `v2` and follow `x.right` with the version stamp 2. The smallest version stamp of `y.left` is 3, which is larger than 2. We can infer that `y.left` must be `nil` in version 2.

2.1.1 Space and Time Analysis

The worst case for deletion in an AVL tree can cause rotations to be made all the way up to the root. That is, we might need to do $O(\log n)$ rotations to rebalance the tree. In this situation, we need additional $O(1)$ space to store the changed states in each node and the root pointer per update step. Because a single or double rotation(s) only affects a constant number of fields to be updated, and no more than $O(\log n)$ will be made because the longest path in an AVL tree is upper bounded. The total additional space cost per deletion is therefore $O(\log n)$.

The case for insertion is simpler. Because insertion in AVL trees can only induce a constant number of updates, the additional space cost per insertion is $O(1)$.

As for searching, in order to know the field value of some node in a particular version `i`, we need to compare it with the version stamps stored in each fat node during navigation. If we use balanced search trees to store the version stamps of each fat node, we need $O(\log m)$ time to find the correct version in one single fat node. In addition, we need to find the root pointer `i` before navigating through the tree. If we store the root pointers in an array, we will have the worst-case time cost of $O(m)$ or amortized $O(1)$ because we need to expand the array at some point. If we store root pointers in a balanced binary search tree, the time complexity is $O(\log m)$.

Since the height of an AVL tree is at most $O(\log n)$, the total time for searching is $O(\log m + \log m \log n) = O(\log m \log n)$. Similarly, insertion and deletion take $O(\log m \log n)$ in a persistent AVL tree using fat node method.

2.2 Full Persistence

By using the same fat node structure, we can apply the fat node method to obtain fully persistent AVL trees. Here, each fat node also contains the same fields as an ephemeral node, as well as space for an arbitrary number of extra field values to store tuples `(field name, field value, version stamp)` for update operations.

2.2.1 Version Tree and Total Version List

Although the idea is the same, the difficulty is that the versions in a fully persistent structure do not have a natural linear ordering like we have in the case of partial persistence. Figure 1.1 (b) has already shown an example of version trees.

The lack of a linear ordering on versions makes navigation through a representation of a fully persistent structure problematic. To eliminate this difficulty, we impose a total ordering on the versions consistent with the partial ordering defined by the version tree. We represent this ordering by a list of the versions in the appropriate order. We call this the *total version list* of the structure. When a new

version, say i, is created, we insert i in the total version list immediately after its parent (in the version tree). The resulting list defines a preorder on the version tree. This implies that the total version list has the following crucial property: for any version i, the descendants of i in the version tree occur in the total version list from i, if i has any descendants.

In addition to performing insertions in the total version list, we need to be able to determine, given two versions i and j, whether i comes before or after j in the version list. We will use a order maintenance data structure proposed by Dietz and Sleator[1] that supports order queries and insertion in $O(1)$ worst-case time.

2.2.2 Searching and Updating in a fully persistent AVL tree

With the total ordering of the versions we defined above, we can navigate through the persistent AVL tree. But instead of comparing the versions with their numeric values, now we look at their relative positions in the total version list. Note that all fields in each node has a subset of the versions, and the ordering of these versions is defined the same way as in total version list. We say the versions of some field value in node `x` forms a `version list`. Basically, version lists are subsequences of the total version list.

The extra steps for updating in fully persistent AVL trees comes from maintaining the correct correspondence between the total version list and the version list of the updated field.

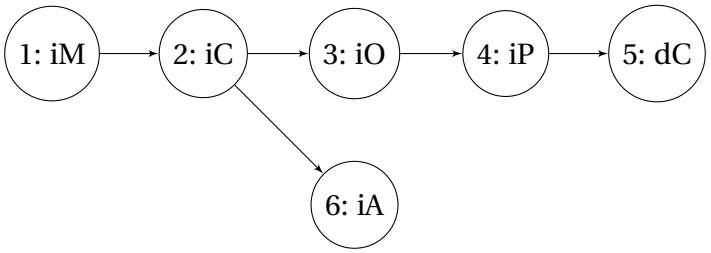


Figure 2.2: A version tree. An "i" or "d" indicates an insertion or deletion of the specified item. The total version list is `(1, 2, 6, 3, 4, 5)`

Now consider a sequence of update operations in Figure 2.2.

After 6 update operations, `M.left` contains version stamps 2 and 5 and its version list is `(2, 5)`. If we want to insert `L` on version 6, we need to store `(left, L, 7)` in node `M`. The new total version list is `(1, 2, 6, 7, 3, 4, 5)` and the new version list of `M.left` is `(2, 7, 5)`.

The catch here is that if we want to search for the value of `M.left` in version 3, the closest version left to 3 in the version list is 7. So we will get `L` instead of the correct value `C`. Therefore, we must store `(left, C, 3)` into node `M` as well.

For a general case, suppose we are at an update step of the update operation i, and this update step modifies some field value of node x. Denote the version list of `x.field` as $(..., i_1, i, i_2, ...)$ after insertion of i. That is, i_1 is the closest version left to i and i_2 is the closest version right of i. Note that i_1 might be equal to i and i_2 might not exist. We also let the total version list be $(..., i_1, ..., i, i+, ..., i_2, ...)$, i.e. $i+$ is the version after i, if such a version exists.

As in the case of partial persistence, if $i_1 = i$, we simply overwrite the field value. If they are different, we store `(field name, field value,` v_i`)` into the fat node `x`. Now, if there exists a version $i+$ between i and i_2 in the total version list, we know in the version interval $[i+, i_2)$, the field value should be the same as in version i_1. However, if we search for the field value of x in this version interval, we will get the value with version i. Therefore, in order to maintain the correspondence of ordering between the total version list and the version list of field, we need to store `(field name, field value in` i_1`,` $i+$`)` into the node in order to maintain the correct correspondence between the total version list and the version list.

In the end we store the root pointer of version `i` into the balanced binary search tree as well.

2.2.3 Space and Time Analysis

Updating and accessing in the case of full persistence is the same as in partial persistence. The only difference is the ordering we impose earlier, but with the order maintenance data structure, we can do insertions and order queries in $O(1)$. When we want to find the closest version left to or equal with our query version in the version list, we can do binary searching in the total version list. This is done by comparing the relative positions of the middle version and the one of the query version in the total version list, we can at least halve the total version list. Therefore, it takes $O(\log m)$ per node as in normal binary search. It follows that we have the same asymptotic efficiency as in the case of partial persistence.

3 Node Copying and Node Splitting

3.1 Node Copying for Partial Persistence

Although the idea of the fat node method is rather intuitive, the fat nodes must be represented by linked collections of fixed-size nodes in the implementation. We eliminate this drawback with our second idea, node copying. With this method we can obtain partial persistence with slowdown of amortized time $O(1)$ per update operation.

We allow nodes in the persistent structure to hold only a fixed number of

field values. When we run out of space in a node, we create a new copy of the node, containing only the newest value of each field. After a node is copied, we must also store the pointer to the new copy in the parent node. If there is no space in the parent, we must copy the parent as well. The cascading effect can be problematic for efficiency. However, if we assume that the underlying ephemeral structure has nodes of constant bounded in-degree and we allow sufficient extra space in each node of the persistent AVL tree, we can derive an $O(1)$ amortized bound on the number of nodes copied.

Before going into details, we need to clarify two terms we are going to use. We call a node of the underlying ephemeral structure an *ephemeral node* and a node of the persistent structure a *persistent node*.

The correspondence between the ephemeral structure and the persistent structure is as follows. Each ephemeral node corresponds to a set of persistent nodes, called a *family*. We call the particular member in the family *live* if it represents the latest version of the ephemeral node and all the other members *dead*. Each version of the ephemeral node corresponds to one member of the family, although several versions of the ephemeral node may correspond to the same member of the family.

The persistent node structure contains all the fields an ephemeral node in an AVL tree has, i.e. `key`, `left`, `right`, `balance`. In addition, we have one extra field to store later updates. During an update operation `i`, if we need to store a new field value into the node, and the extra field has been taken, we need to make a new copy of this node. All the field values of this new copy are the same as the old one except for the updated field. We then add `(field name, field value, version i)` to the pointer(s) in the parent node and potentially other predecessor nodes as well. However, since there is only one path from the root to any node, and in an AVL tree the longest path is $O(\log n)$, we only need to copy $O(\log n)$ new nodes per update operation. In order to achieve this complexity, we need to store the path of visited nodes when we navigate through the tree, so that we can update the pointers in predecessor nodes efficiently.

After an update operation, as in fat node method, we also need an auxiliary structure to store root pointers in each version.

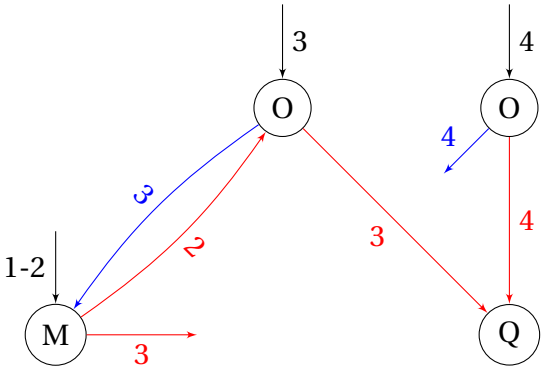


Figure 3.1: An partially persistent AVL tree using node copying after insertions of M, O, Q, followed by the deletion of M. Pointers are labeled with their version stamps. Red edges are right pointers and blue edges are left pointers.

3.1.1 Space and Time Analysis [2]

Before going into detailed analysis, remember that insertion in an array takes amortized constant time because an expensive operation, for example, doubling the array here, happens only once in a while and can make some future insertions constant again. The reason that node copying can achieve better performance is similar to this. A single extra field per node is already enough to let the slowdown of an update operation be amortized constant.

We will use *potential method* here to derive an amortized bound.

Potential Method. [3][4]

Define a potential function Φ on states of a data structure with the following properties:

- $\Phi(h_0) = 0$, where h_0 is the initial state of the data structure.
- $\Phi(h_t) \geq 0$ for all states h_t of the data structure occurring during the course of the computation.

We then define the *amortized cost* of an operation as $c + \Phi(h') - \Phi(h)$, where c is the actual cost of the operation and h and h' are the states of the data structure before and after the operation respectively.

Now consider a sequence of n operations taking actual time $c_0, c_1, ..., c_n - 1$ and producing data structures $h_1, h_2, ..., h_n$ starting from h_0. The total amortized time is

$$(c_0 + \Phi(h_1) - \Phi(h_0)) + (c_1 + \Phi(h_2) - \Phi(h_1)) + ... + (c_{n-1} + \Phi(h_n) - \Phi(h_{n-1}))$$
$$= c_0 + c_1 + ... + c_{n-1} + \Phi(h_n)$$

Since $\Phi(h_n)$ is always non-negative by assumption, the amortized time is a valid upper bound for all operations.

Using potential method, we will see an update operation takes $O(1)$ amortized space and $O(1)$ amortized time. First, we define the potential function $\Phi(T)$ as the number of full live nodes in the persistent AVL tree. The full live nodes are the live nodes whose extra field has been filled. Suppose an update operation generates k copies, where we know $k \leq \log n$, plus a record written into an empty extra field (or a new root added). Each of the k copies costs $O(1)$ space and time, but decreases the potential function by one. To see this, observe that each time a node is copied, this node becomes dead and would not be counted in our potential function. In addition, a new copied node with an empty extra field, i.e. with potential zero, is created. Therefore, we have a negative potential change $-k$.

In the end, we will find a node with an empty extra field and stop cascading of copying, or we copy the root. Either case we increase the potential by one.

Putting it all together, the change in potential is $\Delta\Phi = 1-k$. Thus, the amortized space cost is $O(k+\Delta\Phi) = O(1)$ and the amortized time cost is $O(k+\Delta\Phi+1) = O(1)$.

As for searching, since the size of persistent node is constant, it only takes $O(1)$ to find the correct version in each node, since we only need to check a constant number of fields per node. Overall the time is $O(\log n)$.

3.2 Node Splitting for Full Persistence

In order to achieve full persistence, we can use a variant of node copying called node splitting. We also only allow the persistent nodes to hold only a fixed number of field values, but this time we need more than one extra field in the nodes of an persistent AVL tree. The major difference between node splitting and node copying is that in the former, when a node overflows, a new copy is created and roughly half the extra pointers are moved from the old copy to the new one. This is because we can update on any versions in full persistence, and therefore we should leave space in both the old node and the new copy for later updates. However, the efficient implementation is more complicated than that of node copying. Therefore, we are not going to discuss all the technical details here in the report.

To sum up the results in the paper by Driscoll et al.[2], we can make a linked data structure of constant bounded in-degree fully persistent with a multiplicative factor of amortized time $O(1)$ and space cost of $O(1)$ per update step and a multiplicative factor of $O(1)$ time in the worst case per access, e.g. searching in the AVL tree.

4 Beyond AVL trees

Although we use AVL trees as our example in this report, we point out that the techniques we introduce here can be applied to any general linked structures. By

using a tree as the ephemeral data structure, we are actually granted to simplify the structure of persistent nodes.

For example, if we think of a data structure that resembles DAG (of bounded out-degree by the definition of linked structure), a node might have multiple predecessors. In this case, if we use node copying or node splitting to obtain the persistent data structure, we need to know which node in the family we should update on. In addition, we need to update the pointers that point to the changed node by introducing *inverse pointers*. We might also want to attach a version stamp of the node to keep track of when the node is created. Finally, we should use a singly linked list for a family to group all the members. These additional structures can be omitted in trees because we only have one single access pointer, i.e. the root pointer, and there is only one path to access any node in the tree, which always starts from the root.

Note that the complications of the implementation for general linked structures are only to maintain the correct correspondence between the ephemeral structure and the persistent one. Once we understand the purpose, how to implement a persistent data structure will become clear.

5 Applications and open problems[2]

When we discuss the node copying method, we restrict the ephemeral nodes to have bounded in-degree in order to achieve efficiency. What remains to be shown is whether we can find a way to keep the complexity of amortized $O(1)$ and dispensing with this restriction at the same time.

There are situations where the ability to access and modify past versions may be necessary. We can by using node copying or node splitting method implement partially or fully persistent stacks, queues, and deques with $O(1)$ additional time and space per operation. These data structures are used in computational geometry, implementation of very high level languages, and text and file editing as we have mentioned at the beginning.

We may also want to merge several versions into a new one. Data structures that allow us to access, update, and merge old versions are called *confluently persistent data structures*. Versions control systems like Git and SVN are examples of applications of confluent persistence.

References

[1] P. Dietz and D. Sleator, “Two algorithms for maintaining order in a list”, in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ACM, 1987, pp. 365–372.

[2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent”, *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, Feb. 1989, ISSN: 0022-0000. DOI: 10.1016/0022-0000(89)90034-2. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(89)90034-2.

[3] D. Glasser, *6.854 advanced algorithms*, 2006. [Online]. Available: http://courses.csail.mit.edu/6.854/06/scribe/s2-persistent.pdf.

[4] R. Zabih, *Lecture 20: Amortized analysis*, 2011. [Online]. Available: http://www.cs.cornell.edu/courses/cs3110/2011fa/supplemental/lec20-amortized/amortized.htm.