Persistent Linked Structures at Constant Worst-Case Cost[1]

Ali R. Boroujerdi
Naval Research Laboratory
4555 Overlook Avenue S.W.
Washington, DC 20375

Bernard M. E. Moret
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Abstract

We present a method for making linked structures with nodes of in-degree not exceeding 1 partially persistent at a worst-case time cost of $O(1)$ per access step and a worst-case time and space cost of $O(1)$ per update step. The last two improve the best previous result, which gave $O(1)$ amortized bounds on time and space. Our results extend to full persistence.

1 Introduction

Making a change to an ordinary data structure destroys the old version, leaving only the new one. Such a structure is said to be *ephemeral.* With a *persistent* data structure, on the other hand, old versions are not destroyed, making it possible to access or modify old versions as well as the newest one. A structure is said to be *partially persistent* if every version can be accessed but only the newest version can be modified and *fully persistent* if every version can be both accessed and modified.

Researchers have devised partially or fully persistent forms for a number of data structures, including stacks [10], queues [7], search trees [8, 9, 11, 13, 14], and arrays [3], as well as others [1, 2, 5, 4]. Some of these results provide *ad hoc* solutions to the problem, while others offer a systematic study of persistence. Overmars [12] proposed three general methods to obtain partial persistence. Although his results are the most general to date, they are quite poor in terms of efficiency. Another study of persistence was made by Driscoll et al. [6]. Their results are more efficient than those of Overmars, partly due to the fact that they restrict their attention to linked data structures.

In their paper, Driscoll et al. [6] proposed a method, called *node-copying*, to make linked structures of constant bounded in-degree partially persistent at a worst-case time cost of $O(1)$ per access step and an amortized time and space cost of $O(1)$ per update step. In this paper, we present a technique, which we call *delayed node-copying*, that applies to linked structures with nodes of in-degree not exceeding 1 (e.g., linked lists, trees, etc.). We make such structures partially persistent at a worst-case time cost of $O(1)$ per access step and a worst-case time and space cost of $O(1)$ per update step.

2 Delayed Node-Copying

As in the node-copying method, we only allow nodes of a fixed size in the persistent structure. When we run out of space in a node, we create a new copy of the node, containing the newest value of each field. We then store a pointer to the new copy in the node's predecessor in the

[1]This research was supported in part by contract N00014-92-C-2144 from the Office of Naval Research.

newest version. If there is no space in the predecessor for such a pointer, the predecessor too must be copied. Thus, in the worst case, we will have to copy $O(n)$ nodes as a result of one update, n being the number of nodes in the ephemeral structure. This is exactly what the node-copying method will do; however, the constant-time constraint of our algorithm does not allow us to copy more than a constant number of nodes per update step.

If an update step would result in the creation of too many new copies, we copy a constant number of nodes and simply delay copying the rest. We will copy the remaining nodes later as time permits. This delaying strategy presents us with three main problems. First, the structure will rarely, if ever, be up-to-date, so that we must be able to extract up-to-date information from an out-of-date structure. Secondly, we need to copy nodes in the right order if access steps are to take constant time. More precisely, we must not copy a node more than a constant number of times before updating (or copying) its predecessor. Thirdly, by the time we are ready to copy a node as a result of update operation i, it could be the case that the node has been copied a number of times as a result of update operations that occurred after i, in which case some or all of those copies may contain out-of-date information. Since updating those copies requires more than constant time (worst-case or amortized) per update step, we need to be able to leave some nodes permanently out-of-date without paying more than a constant-time penalty in access and update operations.

2.1 The Data Structure

Without loss of generality, we assume that the ephemeral structure contains only one type of node. In this case, the persistent structure will contain two types of nodes: *persistent* nodes and *header* nodes. Each persistent node consists of $i + p + e + 3$ fields, where i and p are the number of information and pointer fields in an ephemeral node, and e is a constant to be chosen later. Of these fields, $i + p$ are the same fields as those in an ephemeral node and are referred to as *original fields*, e are for *extra fields*, and the last three are for a *version stamp*, a *header pointer*, and a *copy pointer*. The original fields of a node are assumed to have the same version stamp as that of the node itself, while each extra field has an original field name and a version stamp associated with it. Header nodes will be described shortly.

The correspondence between the ephemeral and persistent structures is the same as in [6]. Each ephemeral node, x, corresponds to a set of persistent nodes, $\overline{X}$, called a *family*. Each version of x corresponds to one member of $\overline{X}$, although several versions of x may correspond to the same member of $\overline{X}$. More precisely, version i of x corresponds to the member of $\overline{X}$ with the greatest version stamp not greater than i. The nodes in $\overline{X}$ are formed into a linked list by their copy pointers, in increasing order by version stamp. The ith node on the list is denoted $\overline{x}_i$. We call the last node on the list *live* and all others *dead*. The live nodes and their newest field values represent the newest version of the ephemeral structure. Given live nodes $\overline{x}_i$ and $\overline{y}_j$, we refer to $\overline{x}_i$ as a *current successor* of $\overline{y}_j$ (and hence $\overline{y}_j$ as the *current predecessor* of $\overline{x}_i$) if $\overline{x}_i$ and $\overline{y}_j$ correspond to x and its predecessor y in the newest version of the ephemeral structure; similarly, $\overline{x}_i$ is a *current descendant* of $\overline{y}_j$ (and hence $\overline{y}_j$ is a *current ancestor* of $\overline{x}_i$) if x is a descendant of y in the newest version of the ephemeral structure.

With each family of persistent nodes $\overline{X}$, we associate a header node $\overline{x}_0$ with three pointer fields: *live*, *pred*, and *nextup*. The main function of $\overline{x}_0$ is to allow constant-time access to the live member of $\overline{X}$ through any member $\overline{x}_i$. Thus every member of $\overline{X}$ points to $\overline{x}_0$ through its *header* pointer and $\overline{x}_0$ points to the live member of $\overline{X}$ through its *live* pointer. Given a live node $\overline{x}_i$, we need constant-time access to its current predecessor $\overline{y}_j$. Rather than having a pointer to $\overline{y}_j$ in $\overline{x}_i$, we choose to have a pointer, *pred*, to $\overline{y}_0$ in $\overline{x}_0$. This arrangement saves space since every node has been live at some point. It also saves time since copying a live node does not entail updating its current successors. Informally, we say that a live node $\overline{y}_j$ is out-of-date with respect to a current successor $\overline{z}_k$ if $\overline{y}_j$ does not contain a pointer to $\overline{z}_k$. Further, we say that $\overline{y}_j$ is out-of-date if it is out-of-date with respect to at least one of its successors. The pointer *nextup* points to a record containing information on an out-of-date (not necessarily current) ancestor of the live member of $\overline{X}$. This record contains three fields: a pointer p which points to an out-of-date node $\overline{y}_j$; a pointer q which points to a successor $\overline{z}_k$ of $\overline{y}_j$; and a version stamp *vstamp* which stores the earliest version in which $\overline{y}_j$ is out-of-date with respect to $\overline{z}_k$.

The access and update operations, as we describe them below, require $2p+1$ extra fields in persistent nodes, where p is the number of pointer fields in an ephemeral node. It is possible to use fewer extra fields and still meet the required time constraints—in fact, it is sufficient to have one extra field to obtain the stated time bounds. We have chosen to have $2p + 1$ extra fields for the sake of simplicity, both in the algorithms and in the proofs.

2.2 Update Operations

Our simulation requires that nodes in the ephemeral structure have in-degree less than or equal to 1. While this may be true from an abstract point of view, it may not be true during a manipulation of the ephemeral structure. Since our simulation is meant to be applied mechanically, we assume, without loss of generality, that at any point in time, at most one pointer in the ephemeral (and hence the persistent) structure points to any given node. An ephemeral update operation m is then simulated as follows.

When an ephemeral update step creates a new node x, create a new family of persistent nodes $\overline{X}$ consisting of a single member $\overline{x}_1$ and a corresponding header node $\overline{x}_0$. Set $\overline{x}_0.live$ to $\&\overline{x}_1$, $\overline{x}_0.pred$ and $\overline{x}_0.nextup$ to *null*, $\overline{x}_1.vstamp$ to m, $\overline{x}_1.header$ to $\&\overline{x}_0$, and $\overline{x}_1.copy$ to *null*, where $\&x$ denotes the address of variable x.

When an ephemeral update step changes a field f in an ephemeral node x, simulate the update in two stages: a minimal update on the corresponding persistent node followed by some (suitably chosen) delayed updating.

In the first stage, then, inspect the corresponding persistent node $\overline{x}_i$ and depending on the status of $\overline{x}_i$, take one of the three following steps.

1. $\overline{x}_i$ has version stamp m. Simply change the corresponding original field in $\overline{x}_i$.

2. $\overline{x}_i$ has version stamp less than m and has room for an extra field. Store the new value of f in $\overline{x}_i$ along with the appropriate field name and version stamp m. If f is a pointer field, examine its old and new values. If the old value of f was a pointer to node y, set $\overline{y}_0.pred$ to *null* since y no longer has a predecessor in version m of the ephemeral structure. In addition, since we do not want to update a persistent node with respect to a noncurrent successor, set $r.p$ to *null*, where r is the update record pointed to by

$\overline{y}_0.nextup$.[2] If the new value of f is a pointer to node z, set $\overline{z}_0.pred$ to $\overline{x}_0$ since x is now the predecessor of z in version m of the ephemeral structure.

3. $\overline{x}_i$ has version stamp less than m and has no room for an extra field. Create a new copy, $\overline{x}_{i+1}$, of $\overline{x}_i$ with version stamp m, make the *copy* pointer of $\overline{x}_i$ and the *live* pointer of $\overline{x}_0$ point to it, set its *header* pointer to $\&\overline{x}_0$, its *copy* pointer to *null*, and fill its original fields with the newest values of the fields in $\overline{x}_i$, except for the field corresponding to f which assumes the new value. As in case 2, if f is a pointer field, examine its old and new values. If the old value of f was a pointer to node y, set $\overline{y}_0.pred$ and $r.p$ to *null*, where r is the update record pointed to by $\overline{y}_0.nextup$. If the new value of f is a pointer to node z, set $\overline{z}_0.pred$ to $\&\overline{x}_0$. Finally, create a new update record r pointed to by $\overline{x}_0.nextup$, set $r.p$ to $\overline{x}_0.pred$, $r.q$ to $\&\overline{x}_0$, and $r.vstamp$ to m.

In the second stage, use the update record pointer to identify delayed updating work. Let r be the update record pointed to by $\overline{x}_0.nextup$ and m' the version stamp of r. Let $\overline{y}_j$ and $\overline{z}_k$ be the live members of the families of nodes pointed to by $r.p$ and $r.q$ respectively. Depending on the status of $\overline{y}_j$, take one of the four following steps.

1. $\overline{y}_j$ does not exist, i.e., either $\overline{x}_0.nextup$ or $r.p$ is *null*. Do nothing.

2. $\overline{y}_j$ has version stamp greater than or equal to m'. Then $\overline{y}_j$ is up to date, so there is no need for a new version. Let f' be the pointer field with the greatest version stamp in $\overline{y}_j$ pointing to $\overline{z}_{k-1}$; set f' to $\&\overline{z}_k$ and $r.p$ to *null*.

3. $\overline{y}_j$ has version stamp less than m' and has room for an extra field. Store a pointer to $\overline{z}_k$ in $\overline{y}_j$ along with the appropriate field name, which must be the same as the name of the field with the greatest version stamp pointing to $\overline{z}_{k-1}$, and version stamp m'. In addition, set $r.p$ to *null* and $\overline{z}_0.nextup$ to $\overline{y}_0.nextup$.

4. $\overline{y}_j$ has version stamp less than m' and has no room for an extra field. Let f' be the pointer field with the greatest version stamp in $\overline{y}_j$ pointing to $\overline{z}_{k-1}$. Let m'' be either m' or the greatest version stamp in $\overline{y}_j$, whichever is greater. Create a new copy, $\overline{y}_{j+1}$, of $\overline{y}_j$ with version stamp m'', make the *copy* pointer of $\overline{y}_j$ and the *live* pointer of $\overline{y}_0$ point to it, set its *header* pointer to $\&\overline{y}_0$, its *copy* pointer to *null*, and fill its original fields with the newest values in $\overline{y}_j$, except for f' which is set to point to $\overline{z}_k$. In addition, set $r.vstamp$ to m'', $r.p$ to $\overline{y}_0.pred$, and $r.q$ to $\&\overline{y}_0$.

2.3 Access Operations

As in the node-copying method, we use access arrays, one per access pointer, to store the different versions of the access pointers. We can then access an entry node in any version in constant time. To simulate an ephemeral access step that applies to version m and accesses field f of ephemeral node x, we access, in the corresponding persistent node $\overline{x}_i$, the field with the greatest version stamp no greater than m that corresponds to f. In addition, any pointer

[2] Strictly speaking, this is only necessary if $r.p$ points to $\overline{x}_0$. In this case, however, it is crucial to do so, lest we copy a node more than a constant number of times before updating (or copying) its current predecessor.

to a persistent node $\overline{y}_j$ that applies to version m is taken to be a pointer to either $\overline{y}_j$ or to its copy $\overline{y}_{j+1}$, whichever has the greatest version stamp no greater than m.

2.4 Proof of Correctness

Our simulation of ephemeral access and update operations clearly satisfies the stated time and space bounds. Before proving its correctness, we define a few terms.

- $\overline{x}_i$ is out-of-date by k with respect to $\overline{y}_j$ in version m iff $\overline{x}_i$ and $\overline{y}_j$ correspond to version m of ephemeral nodes x and y, and there exists a field f in version m of x such that f points to y and the corresponding field in $\overline{x}_i$ points to $\overline{y}_{j-k}$.
- $\overline{x}_i$ is out-of-date by k with respect to $\overline{y}_j$ iff it is out-of-date by k with respect to $\overline{y}_j$ in some version m and there exists no version m' such that $\overline{x}_i$ is out-of-date by k', $k' > k$, with respect to $\overline{y}_j$ in m'.
- $\overline{x}_i$ is out-of-date with respect to $\overline{y}_j$ iff it is out-of-date by k, $k > 0$, with respect to $\overline{y}_j$.
- $\overline{x}_i$ is said to be *out-of-date* iff it is out-of-date with respect to some node $\overline{y}_j$; a persistent node that is not out-of-date is said to be *up-to-date*.
- $\overline{x}_i$ is said to be *leading* iff its predecessor $\overline{y}_j$ is out-of-date with respect to $\overline{x}_i$.
- Given a live node $\overline{x}_i$, we refer to the closest leading current ancestor $\overline{y}_j$ of $\overline{x}_i$ as the *principal leading ancestor* of $\overline{x}_i$, and refer to the current predecessor of $\overline{y}_j$ as the *principal out-of-date ancestor* of $\overline{x}_i$.
- Given a live node $\overline{x}_i$, its current predecessor $\overline{y}_j$, and the update record r pointed to by $\overline{x}_0.nextup$, $\overline{x}_i$ is *active* iff either $\overline{y}_j$ is out-of-date with respect to $\overline{x}_i$, $r.p$ points to $\overline{y}_0$, and $r.q$ points to $\overline{x}_0$, or $\overline{y}_j$ is up-to-date with respect to $\overline{x}_i$ and $\overline{y}_0.nextup$ points to r; a live node that is not active is said to be *inactive*.
- The *absorbance factor* of a live node $\overline{x}_i$ with respect to a subgraph G of the persistent structure is defined as $\epsilon - (\iota + \phi + \sigma + 1)$, where ϵ is the number of unused extra fields in $\overline{x}_i$, ι is the number of inactive current successors of $\overline{x}_i$ in G, ϕ is the number of leading current successors of $\overline{x}_i$ in G, and σ is the number of active current successors such that each successor $\overline{y}_j$ has an absorbance factor less than 0 with respect to $G'_{\overline{y}_j}$, where $G'_{\overline{y}_j}$ is the connected component of G' containing $\overline{y}_j$, and G' is the graph induced by removing $\overline{x}_i$ and all edges incident to it from G.
- The absorbance factor of a live node $\overline{x}_i$ is the absorbance factor of $\overline{x}_i$ with respect to $G_{\overline{x}_i}$, the graph consisting of $\overline{x}_i$ and its current descendants in the persistent structure. A live node $\overline{x}_i$ is said to be *absorbant* iff it has an absorbance factor greater than or equal to 0.

We can now prove the correctness of the simulation, which is immediately implied by the following theorem.

Theorem 1 *Given room for $2p+1$ extra fields in persistent nodes, where p is the maximum out-degree in the ephemeral structure, the simulation of an ephemeral update step preserves the following invariant.*

For every live node $\overline{x}_i$ and every current successor $\overline{y}_j$ of $\overline{x}_i$, either

1. $\overline{x}_i$ is up-to-date with respect to $\overline{y}_j$, or

2. $\overline{x}_i$ is out-of-date by 1 with respect to $\overline{y}_j$ and $\overline{y}_j$ is active and has an absorbance factor of at least $p+1$.

Proof: Consider the simulation of ephemeral update operation m. If an ephemeral update step creates a new node x, the simulation trivially preserves the invariant. If an ephemeral update step changes a field f in an ephemeral node x, we inspect the corresponding persistent node $\overline{x}_i$. Depending on the status of $\overline{x}_i$, the persistent structure is updated in one of three ways. We prove the invariant is preserved in all three cases.

1. $\overline{x}_i$ has version stamp m. In this case, we first change the corresponding original field in $\overline{x}_i$. Let r be the update record pointed to by $\overline{x}_0.nextup$. Let $\overline{y}_j$ and $\overline{z}_k$ be the live members of the families of nodes pointed to by $r.p$ and $r.q$ respectively. After updating $\overline{x}_i$, we examine $\overline{y}_j$ and, depending on the status of $\overline{y}_j$, complete the update in one of three ways. We show that the invariant is preserved in all three cases.

   (a) $\overline{y}_j$ does not exist, i.e., either $\overline{x}_0.nextup$ or $r.p$ is *null*. We do nothing more. If none of $\overline{x}_i$'s current ancestors are leading, the invariant is trivially preserved. Otherwise, let $\overline{u}_g$ be the principal leading ancestor of $\overline{x}_i$. $\overline{x}_i$ must have an inactive current ancestor $\overline{v}_h$ (which could be $\overline{x}_i$ itself) closer than $\overline{u}_g$, or $\overline{y}_j$ would have existed (more precisely, $\overline{y}_j$ would have been the current predecessor of $\overline{u}_g$). The absorbance factor of $\overline{v}_h$'s current predecessor (which could be $\overline{u}_g$) and hence the absorbance factor of $\overline{u}_g$ is not affected by the update of $\overline{x}_i$. Thus the invariant is preserved.

   (b) $\overline{y}_j$ has room for the update with respect to $\overline{z}_k$.[3] We store a pointer to $\overline{z}_k$ in $\overline{y}_j$ and set $r.p$ to *null*. In addition, we set $\overline{z}_0.nextup$ to $\overline{y}_0.nextup$ to keep $\overline{z}_k$ active. We shall refer to these operations collectively as the update of $\overline{y}_j$. To prove preservation of the invariant, we consider the possible status of $\overline{z}_k$ before the update of $\overline{y}_j$.

      i. $\overline{z}_k$ is the principal leading ancestor of $\overline{x}_i$. Then $\overline{y}_j$ must be the principal out-of-date ancestor of $\overline{x}_i$. Updating $\overline{x}_i$ and $\overline{y}_j$ decreases the absorbance factor of $\overline{z}_k$ by at most p. Since $\overline{z}_k$ had an absorbance factor of at least $p+1$ before the update, it will remain absorbant after the update. Therefore, the absorbance factor of $\overline{y}_j$ does not decrease since it is brought up-to-date with respect to an absorbant active current successor, namely $\overline{z}_k$. Thus the invariant is preserved.

      ii. $\overline{z}_k$ is not the principal leading ancestor of $\overline{x}_i$. Then $\overline{y}_j$ may or may not be the principal out-of-date ancestor of $\overline{x}_i$. The update of $\overline{x}_i$ preserves the invariant by the same argument as in case 1a. The update of $\overline{y}_j$ decreases the absorbance factor of $\overline{z}_k$ by at most p and therefore does not decrease the absorbance factor of $\overline{y}_j$ since $\overline{y}_j$ is brought up-to-date with respect to an absorbant active current successor, namely $\overline{z}_k$. Thus the invariant is preserved.

   (c) $\overline{y}_j$ has no room for the update with respect to $\overline{z}_k$. Let $\overline{u}_g$ be the current predecessor of $\overline{y}_j$. We create a new copy, $\overline{y}_{j+1}$, of $\overline{y}_j$ to store a pointer to $\overline{z}_k$, set $r.p$ to $\&\overline{u}_0$, $r.q$ to $\&\overline{y}_0$, and $\overline{y}_0.nextup$ to $\&r$. We shall refer to these operations collectively as the

[3] i.e., to bring $\overline{y}_j$ up-to-date with respect to $\overline{z}_k$, we either have to update an original field in $\overline{y}_j$, or we have to use an extra field and $\overline{y}_j$ has room for an extra field.

creation of $\overline{y}_{j+1}$. To prove preservation of the invariant, we consider the status of $\overline{z}_k$ before the creation of $\overline{y}_{j+1}$.

i. $\overline{z}_k$ is the principal leading ancestor of $\overline{x}_i$. Then the update of $\overline{x}_i$ decreases the absorbance factor of $\overline{z}_k$ by at most 1 (whether or not $\overline{z}_k$ is a proper ancestor of $\overline{x}_i$) and hence leaves $\overline{z}_k$ absorbant. The creation of $\overline{y}_{j+1}$ leaves the absorbance factor of $\overline{z}_k$ unchanged. Since $\overline{y}_{j+1}$, a newly created node, is up-to-date with respect to $\overline{z}_k$, an absorbant active current successor, its absorbance factor is at least $p+1$. In addition, since $\overline{y}_j$ had no room for the update, $\overline{u}_g$ must be up-to-date with respect to $\overline{y}_j$ and hence out-of-date by 1 with respect to $\overline{y}_{j+1}$. Thus the invariant is preserved.

ii. $\overline{z}_k$ is not the principal leading ancestor of $\overline{x}_i$. Then the update of $\overline{x}_i$ preserves the invariant by the same argument as in case 1a, and the creation of $\overline{y}_{j+1}$ preserves the invariant by an argument similar to that of case 1(c)i.

2. $\overline{x}_i$ has version stamp less than m and has room for an extra field. Let $\overline{y}_j$ and $\overline{z}_k$ be defined as in 1. In this case, the simulation is the same as in 1, except we use an extra field in $\overline{x}_i$ to store the new value of f. The invariant is preserved by the same argument as in 1, except when $\overline{x}_i$ and $\overline{z}_k$ are the same nodes and $\overline{y}_j$ has room for the update with respect to $\overline{z}_k$. If this is the case, the updates of $\overline{x}_i$ and $\overline{y}_j$ combined could decrease the absorbance factor of $\overline{z}_k$ by as much as $p+1$, p being the maximum out-degree in the ephemeral structure. However, since the absorbance factor of $\overline{z}_k$ is at least $p+1$ before the update, $\overline{z}_k$ will remain absorbant after the update. Thus the invariant is preserved.

3. $\overline{x}_i$ has version stamp less than m and has no room for an extra field. Let $\overline{y}_j$ be the current predecessor of $\overline{x}_i$. In this case, we create a new copy, $\overline{x}_{i+1}$, of $\overline{x}_i$ to store the new value of f, create a new update record r, and set $r.p$ to $\&\overline{y}_0$, $r.q$ to $\&\overline{x}_0$, $r.vstamp$ to m, and $\overline{x}_0.nextup$ to $\&r$. We then inspect $\overline{y}_j$ since it is the live member of the family of nodes pointed to by $r.p$ and depending on its status, complete the update in one of two ways.

   (a) $\overline{y}_j$ has room for the update with respect to $\overline{x}_{i+1}$. We store a pointer to $\overline{x}_{i+1}$ in $\overline{y}_j$ and set $r.p$ to $null$ and $\overline{x}_0.nextup$ to $\overline{y}_0.nextup$. $\overline{x}_{i+1}$, having just been created, is an absorbant node, whereas $\overline{x}_i$ is not. Since $\overline{y}_j$ remains up-to-date with respect to the live member of $\overline{X}$, now an absorbant active current successor, its absorbance factor does not decrease. Thus the invariant is preserved.

   (b) $\overline{y}_j$ has no room for the update with respect to $\overline{x}_{i+1}$. Let $\overline{z}_k$ be the current predecessor of $\overline{y}_j$. We create a new copy, $\overline{y}_{j+1}$, of $\overline{y}_j$ to store a pointer to $\overline{x}_{i+1}$ and set $r.p$ to $\&\overline{z}_0$, $r.q$ to $\&\overline{y}_0$, and $\overline{y}_0.nextup$ to $\&r$. Since $\overline{y}_{j+1}$ is up-to-date with respect to $\overline{x}_{i+1}$, an absorbant active current successor, it has an absorbance factor of at least $p+1$. In addition, since $\overline{y}_j$ had no room for the update, $\overline{z}_k$ must be up-to-date with respect to $\overline{y}_j$ and hence out-of-date by 1 with respect to $\overline{y}_{j+1}$. Thus the invariant is preserved.

Q.E.D.

3 Extensions

The method we have described extends to full persistence. Much more difficult is the question of whether our algorithms can be modified to handle linked structures of bounded in-degree larger than 1. The problem is that of updating (and copying) the right nodes at the right time: while such nodes and times exist, identifying the right node at the right time appears to require more than constant time. Using elaborate data structures may reduce the time needed to locate the right node to $o(\log n)$ time, but $O(1)$ time appears unattainable by our techniques.

References

[1] B. Chazelle. How to search in history. *Inform. and Control* **77** (1985), 77–99.

[2] R. Cole. Searching and storing similar lists. *J. Algorithms* **7** (1986), 202–220.

[3] P. F. Dietz. Fully persistent arrays. *Proc. Workshop on Algorithms and Data Structures* WADS89, pp. 67–74.

[4] P. F. Dietz and R. Raman. Persistence, amortization and randomization. Tech. Rep. 353, Department of Computer Science, U. of Rochester, Rochester, NY, 1990.

[5] D. P. Dobkin and J. I. Munro. Efficient uses of the past. *Proc. 21st Annual IEEE Symp. on Foundations of Computer Science* FOCS80, pp. 200–206.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. of Computer and System Sciences* **38** (1989), 86–124.

[7] R. Hood and R. Melville. Real-time queue operations in pure LISP. *Inform. Process. Letters* **13** (1981), 50–54.

[8] T. Krijnen and L. G. L. T. Meertens. Making B-trees work for B. Tech. Rep. IW 219/83, The Mathematical Centre, Amsterdam, The Netherlands, 1983.

[9] E. W. Myers. AVL dags. Tech. Rep. 82-9, Department of Computer Science, U. of Arizona, Tucson, AZ, 1982.

[10] E. W. Myers. An applicative random-access stack. *Inform. and Control* **17** (1983), 241–248.

[11] E. W. Myers. Efficient applicative data types. *Proc. 11th Annual ACM Symp. on Principles of Programming Languages* POPL84, pp. 66–75.

[12] M. H. Overmars. Searching in the past II: General transforms. Tech. Rep. RUU-CS-81-9, Department of Computer Science, U. of Utrecht, Utrecht, The Netherlands, 1981.

[13] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. and Systems* **5** (1983), 449–477.

[14] G. F. Swart. Efficient algorithms for computing geometric intersections. Tech. Rep. 85-01-02, Department of Computer Science, U. of Washington, Seattle, WA, 1985.