Discrete Mathematics and Theoretical Computer Science **3**, 1999, 95–107

Partially Persistent Search Trees with Transcript Operations †

Kim S. Larsen‡

Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, Campusvej 55, DK–5230 Odense M, Denmark,
`kslarsen@imada.sdu.dk`

received 1 st June 1998, revised 21 th May 1999, accepted 21 th May 1999.

When dictionaries are persistent, it is natural to introduce a transcript operation which reports the status changes for a given key over time. We discuss when and how a time and space efficient implementation of this operation can be provided.

Keywords: Data structures, Search trees, Persistence, Complexity.

1 Introduction

When balanced binary search trees are made partially persistent using the node-copying method [5], the possibility of searching efficiently for information in the past is added to the system. The operations of updating the present version and searching in the present as well as in the past are asymptotically as efficient as in the corresponding normal (non-persistent) binary search tree.

In database applications, it is sometimes desirable to produce transcripts of information change over time. If we wish to obtain a transcript of information related to some key k from version number v_1 to v_2, this can be obtained by independent search operations in all versions in that interval in time $O(ph)$, where h is the maximum height of the search tree in that interval, and $p = v_2 - v_1 + 1$ is the number of versions between v_1 and v_2. We discuss when and how this can be reduced to $O(h + p)$ by maintaining one extra pointer with a version number in each node, without changing the asymptotic complexity of any of the existing operations.

In database applications, search trees are usually leaf-oriented, which means that all keys reside in the leaves, and internal nodes contain routers guiding the search to the correct leaf. Leaves are often of another type than the internal nodes, and contain extra pointers or space consuming values associated with

† A preliminary version of this paper appeared in the proceedings of the 15th Symposium on Theoretical Aspects of Computer Science 1998 (STACS'98), Lecture Notes in Computer Science, Vol. 1373, pages 309-319, Springer-Verlag, 1998.

‡This work was carried out while the author was visiting the Department of Computer Sciences, University of Wisconsin at Madison. Supported in part by SNF (Denmark), in part by NSF (U.S.) grant CCR-9510244, and in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM-IT).

their keys. We remain faithful to this model, and the extra pointer which is introduced only appears in internal nodes. Leaves will contain no extra information compared to the single version scenario.

The complexity of the transcript operation turns out to depend on how often leaves can get new parents during rebalancing. We introduce the concept of search trees with *limited leaf action* as a necessary and sufficient condition for a balanced search tree scheme to be equipped with an efficient transcript operation using the method outlined in this paper.

The most interesting case to consider is the case where $O(ph)$ is large, i.e., there are a large number of versions. For the case where we change to a new version whenever a fixed constant number of updates has been carried out (or earlier), we show that the transcript operation can be computed in time $O(h+p)$, where h is the height of the tree in version v_1. If updates in the given tree take time $O(h)$, they remain $O(h)$. As in [5], space consumption is linear in the number of changes made to the structure.

Among the balanced search tree schemes which turn out to have limited leaf action are red-black trees [7] and treaps [3], for instance. So, here updating becomes $O(\log n)$ (for treaps, expected time) and transcripts $O(\log n + p)$ (for treaps, the $\log n$ part is expected time).

In the next section, we first define leaf-oriented search trees, then we extend them to become partially persistent, and finally, we include the transcript additions. In the following sections, we discuss correctness, complexity, and future work.

2 Transcript Trees

When search trees are used in database applications, the trees are usually *leaf-oriented*. This means that only the leaves contain keys. The internal nodes contain routers, which are of the same type as the keys and which direct the searches to the correct location as usual in a search tree. However, routers need not be present as keys in the tree. This means that we do not have to update routers whenever a deletion takes place. For an internal node, the keys in its left subtree are smaller than or equal to its router, and the keys in its right subtree are larger. A leaf-oriented tree is always a *full* tree, i.e., every internal node has two children. This simplifies the deletion operation.

Searching and Updating

To *insert* a key k in the tree, search for k as usual. An unsuccessful search ends up in a leaf, say l. A new internal node u is created in place of l, and l and a new leaf l' containing the key k are made the children nodes of u. The one containing the smaller key will be the left child. The router of u is a copy of the key contained in its left child. Thus, there are new pointers in the new node, but only one pointer in the existing structure is changed. See Figure 1 on the following page.

To *delete* a key k from the tree, first search for k as usual in a search tree. If the key is found in the leaf l, its parent is replaced by the sibling node of l. Again, a node and its pointers are deleted, but only one pointer is changed. See Figure 2 on the next page.

The insertion and deletion operations are called *update* operations. It is easy to prove that while a router r is present in the tree, no new node with the same router can be created. This is important since otherwise rotations could violate the search tree invariant.

We assume that some scheme for maintaining balanced search trees will be used. Thus, the nodes contain additional fields for registering heights, colors, or other balance information. The update operations may manipulate these fields, and there will be a collection of rebalancing operations, which after each

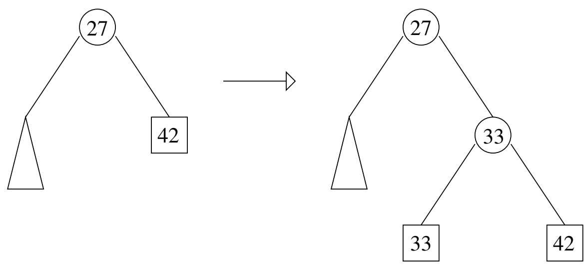

Fig. 1: Insertion of 33 into a leaf-oriented tree.

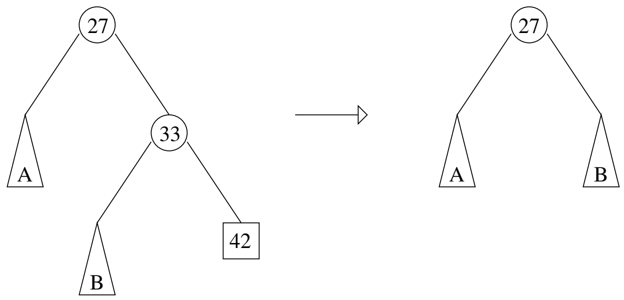

Fig. 2: Deletion of 42 in a leaf-oriented tree.

update, by manipulating the fields and applying rotations, will make sure that the balance constraints, if violated by the update, are again fulfilled. We discuss this in greater detail later.

However, we require that any rebalancing operation can be expressed as a constant number of single rotations carried out as described now (see Figure 3 on the following page).

The single rotation can be either *right* (in the direction of the arrow) or *left* (in the opposite direction of the arrow). We give the requirements for a right rotation; the other is similar. Assume that the parent of v is a node w, and assume without loss of generality that it points to v via its left pointer. Then carry out $v.left{:=}u.right$; $u.right{:=}v$; $w.left{:=}u$.

This could be done differently. However, the whole point is that nodes must keep their identity since we introduce new pointers which should aim at the correct nodes, even when these are moved around by rebalancing operations. Balance information can be updated by the rebalancing operations in any way desired.

Note that this requirement is no (real) restriction. This is exactly the way single rotations are always performed, and double rotations are always expressed such that the result of one equals the result of two consecutive single rotations. In fact, any binary search tree can be constructed from any other binary search tree with the same keys by applying a sequence of single rotations. We do not know of any scheme which does not conform to this requirement, except the ones which apply some form of global rebuilding as in [2, 6].

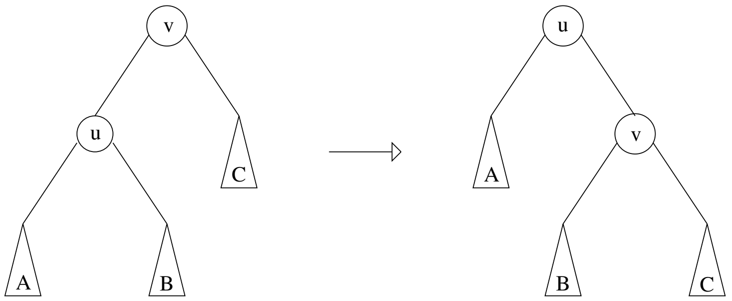

Fig. 3: Single rotation (to the right).

Partial Persistence

We now make the search tree partially persistent using the *node-copying method* [5]. A *partially persistent* structure is a structure which supports multiple versions, such that all versions can be accessed, but only the newest version can be modified. We assume that versions are numbered using consecutive integers. All nodes have a field stating under which version they were created. The structure has a number of *entry* pointers (pointers to roots of different versions) which also have version numbers.

For balanced search trees, the node-copying method can be used in a simplified form [5]. We use an extended version of that simplified form. We retain what is referred to as the *copy pointer*, equip it with a version number, and update it as in the original method. For completeness, and because we build on top of this basic method, we describe it here, but emphasize that a very similar description has been given in [5].

There are three types of information in a balanced search tree: keys, pointers, and balance information. As information considered, we are only interested in the keys. Since we can only update the newest version, balance information is not needed for older versions. We always rebalance after an update, so when we switch to a new version, the older versions will remain balanced forever. So, only keys and pointers from old versions must be kept, and it is safe to overwrite old balance information.

When we insert a new key, we create an entirely new node to put it in. So, no key information is being changed. Instead, it is one pointer in one node that is being changed to include the new node. Similarly, when we delete a key, this is done by changing one pointer to cut out the node, in which the key resides. So, we only need to discuss pointer updates; not key updates.

To avoid copying too many nodes every time a pointer must be changed, nodes have one extra field for a pointer update. So, nodes in the tree have fields: *key*, *left*, *right*, *vn*, *extra*, and *copy*, where "vn" is short for "version number". The field *extra* is composite, and has the following fields: *ptr*, *dir*, and *vn* for recording the new pointer, which pointer it replaces (*dir* is *Left* or *Right*), and in which version it was done ("ptr" is short for "pointer" and "dir" is short for "direction"). The field *copy* is also composite with fields *ptr* and *vn*.

An update in the newest version i is handled as described now. We explain the action which must be taken for *one* pointer change. If an update (or a rebalancing operation) involves several pointer changes, the procedure is repeated.

If the node u in which the update is made has version number i, the update overwrites the existing pointer. Otherwise, if the extra field has version number i *and* its direction field indicates the pointer to be updated, the pointer in the extra field is overwritten.

If neither case applies, there are two possibilities depending on whether or not the extra field has already been used. If it has not, the update is made there, also setting the direction field and setting the version number to i.

Otherwise, a new node v must be made. This is the only case, where the procedure does not terminate immediately. The key and the *newest* left and right pointers from u are copied into v, i.e., the pointer in the extra field of u and the pointer from u which was *not* overwritten in the extra field. Then v's version number is set to i and its extra and copy fields to nil. Finally, the copy field in u is set to point to v, and its version number is set to i. Thus, copy pointers link together sequences of nodes which are basically the "same" node at different times. Now, the parent of u must be updated to point to v instead. This is done *recursively* using the method just outlined, i.e., the effects of an update continue up in the tree along the update path for as long as there are nodes where the extra field has already been used by an earlier version, or has been used in the opposite direction of the update path by the current version. If the root is copied, then a new entry pointer to the new root with version number i is created. For an example, see Figure 4. Nodes, extra pointers (in the middle), and copy pointers (dashed) have version numbers. Thick lines indicate parts of the tree which were present before the operation. The current version number is 3.

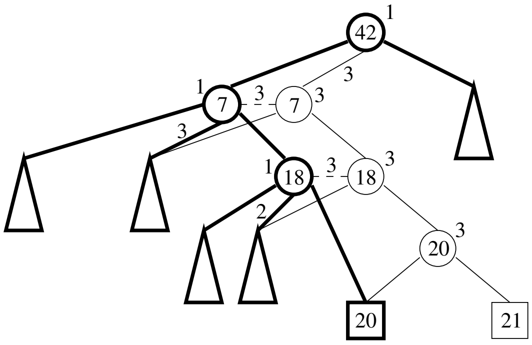

Fig. 4: Inserting 21.

Searching becomes slightly more complicated after these changes. Searching starts by following an entry pointer. There will not necessarily be an entry pointer for each version. Instead, when accessing version j, the entry pointer with the largest version number i, $i \leq j$, should be used.

At each node, a decision to continue the search to the left or the right must be made. As usual, this decision is made by comparing the key to be found with the router (key) in the node. However, when the decision has been made to proceed to the left, for instance, then the *newest* left pointer no newer than j must be followed, i.e., if the left pointer has been updated in the extra field and the version number of the extra field is at most j, then the pointer in the extra field should be followed. Otherwise, the original left pointer is taken.

The following result, or more general ones, are proven in [5].

Theorem 1 When the node-copying method is applied to a pointer structure for which there is an upper bound p on the number of pointers that can point to any one node, then if nodes in the persistent version are equipped with at least p extra pointers, the following holds.

- The asymptotic complexity of searching in any version in the persistent structure is equal to the asymptotic complexity of searching in the standard structure corresponding to that version.
- The persistence actions carried out after one pointer change are performed in amortized constant time.
- When searching in the newest version, only the last node in a copy sequence can be accessed.

Note that in connection with the trees we use here, only the parent of a node can point to that node, so one extra field suffices. Also, since we use only a constant amount of new space in each step, space usage per pointer update is also amortized constant.

With the definitions in this section, the structure is no longer a tree, but a directed acyclic graph. However, we will keep referring to it as a tree.

Transcript Facilitating Additions

In this section, we extend the updating and rebalancing procedures even further. We first describe our goal informally and then we give the exact description of the procedures, the properties of which will be used in the correctness proof later.

The general idea is that if we want to keep track of some key k, then we position ourselves at the internal node under which k is found (or would be inserted). When changing version, from v_1 to v_2 say, the internal node with that property may be deleted, or an insertion or rotation may have the effect that it is now another node which has the given property.

Therefore, whenever we make an update or a rotation, we also build a path from the node with the given property in version v_1 to the one with that property in version v_2, such that later, during a transcript operation, it will be fast to get to that new node. We make sure the path is protected in the sense that no later pointer updates can alter it.

The result is a tree as in [5] with extra pointers and copies of nodes creating protected paths which run through the tree (over time) at the levels just above the leaves.

We now turn to the concrete additions. Note first that when an internal node is deleted, no pointer updates can ever be applied to it again. This means that its copy pointer will never be used. In other words, we are free to use it for other purposes (this is safe since the copy pointer is never used in searching). Additionally, when an insertion or a rotation is carried out, we may *trigger* a copying of a node which would not have been made at that time in the original method. This means that in contrast to [5], we can have several copies of a node with the same version number. These are the only changes we are making. Whenever the copy pointer is set, its version number is also set, and it is set to the newest (that is, the current) version. From now on, we will not mention the version number when discussing the setting of the copy pointer.

For deletion, there are two cases. First assume that both of the children of the internal node u to be deleted are leaves. The deletion is performed, and the copy pointer of u is set to point to the newest copy of the parent of u.

Now assume that the internal node to be deleted has only one leaf among its two children. After the deletion has been performed, the copy pointer of u is set as follows. If the leaf is its left child, then its copy pointer is set to point to its *in-order internal successor* (the left-most internal node in its right subtree). This case is illustrated in Figure 5 on the following page. Note that persistence actions will continue above the node with key 7, but this is not shown. Similarly, if the leaf is its right child, then its copy pointer is set

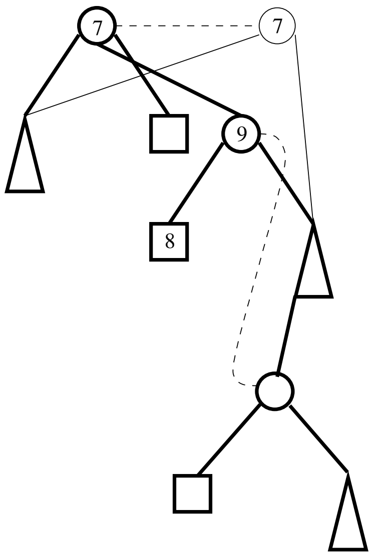

Fig. 5: Deleting the key 8 from a transcript tree.

to point to its *in-order internal predecessor* (the right-most internal node in its left subtree). The successor (or predecessor) is found by a search in the newest version of the structure starting at the internal node to be deleted.

For insertion, we do the following. As described earlier, we first search for the correct leaf, and the only pointer in the existing structure which is changed is the one in the internal node v which points to the leaf in question. We make the insertion as usual for leaf-oriented trees, applying the necessary persistence actions, i.e., v may be copied. When this is completed, we trigger a new copy of v, or of the copy of v, if v has already been copied during the insertion. This last case is illustrated in Figure 6. Note that since the node with key 27 is copied, other actions will take place further up in the tree, but this is not shown.

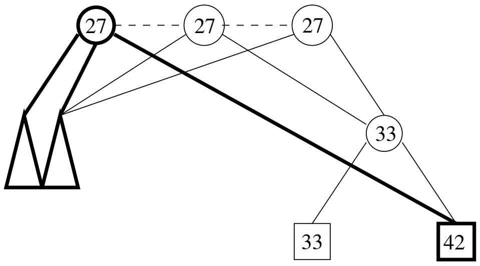

Fig. 6: Insertion of key 33 into a transcript tree.

The actions taken for a rotation are similar to those for insertions. First, the rotation is carried out, and all the necessary persistence actions are taken. Then if the right child of u (refer to Figure 3 on page 98) before the rotation was a leaf, we trigger a new copy of the newest version of node u. In general, to make a description which covers left as well as right rotations, if the middle subtree is a leaf, we trigger a new copy. An example of this is given in Figure 7. In that example, no extra pointers were in use before the rotation. Note that persistence actions due to the copying will continue up in the part of the tree above u (not shown).

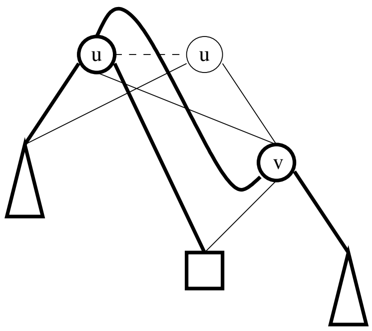

Fig. 7: Right rotation in a transcript tree.

The Transcript Operation

The transcript operation can now be defined. It takes a key k and two version numbers v_1 and v_2 (with $v_1 \leq v_2$) as parameters and prints the history of k between these two versions. More precisely, a line is printed for every version, stating whether or not k is in the tree. In a practical application, there would most likely be values associated with each key, and those could be printed as well (these might not be the same every time k appears in the tree).

The implementation is given in Figure 8 on page 107. We explain the ingredients below.

We have mostly used standard notation in the algorithms, but we comment on a few points. We have used an enumeration type with the two values Left and Right. Boolean expressions are evaluated C-style, i.e., they are evaluated from left to right and as soon as the final value of the whole expression can be deduced, evaluation is aborted. When we use the expression (b ? e_1 : e_2), the boolean expression b is evaluated first. If it evaluates to true, then the result of the whole expression is the result of evaluating e_1. Otherwise, it is the result of evaluating e_2. In order to avoid too many details, we assume that the tree is always non-empty.

We assume that we have a function Leaf which decides if a node is a leaf, and a function Entry which given a version number, returns the entry pointer to be used.

The function *Find* finds the parent of a given key in a given version. This is simply a search, except that we return the parent of the key. The leaf in the direction we would search to find k (or rather, to check whether or not k is in the tree at the time) is referred to as the *leaf possibly containing k*. *Find* calls *Go* which takes one step down in the tree by going left or right as appropriate. Assuming that we are located at the parent of the leaf possibly containing k, *Status* returns the information as to whether or not k is present.

Transcript works by repeatedly calling *Advance*. The function *Advance* is called with a key k, a node u, and a version number i. Except for the first call, the assumption is that the information returned by the previous call to *Advance* was from u in version $i-1$, i.e., u was the internal node which could have a leaf child containing k. *Advance* finds the next similar node in version i, and returns information on the status of k at that time, as well as the node.

3 Correctness

The only non-trivial operation is the function *Advance*. We must argue that it always advances (so termination is guaranteed) and not too much (such that information is overlooked).

When searching through version i by starting at the entry pointer, one will see version i as it appeared when we switched to the next version. However, the current version can keep changing right until we switch. This complicates the search by *Advance* even after we have switched to a new version because we do not enter via the entry pointer. Thus, we may see parts of version i which were temporary, and which would never be found when entering "correctly". This is the reason for the **while** loop in function *Advance*. Whenever we use *Find*, we are searching in version i as it appeared at some point; not necessarily in its final appearance. For instance, several deletions could be made in the same version so it may be necessary to follow the copy pointer several times.

The reason for triggering a node copying in connection with insertions and rotations is that the parent of the leaf possibly containing k changes. By triggering a copying of the node which used to be the parent, we ensure that no later operations can prevent us from accessing the new parent. Thus, we build a protected path from the parent of the leaf possibly containing k in one version to the (possibly new) parent in the next.

Theorem 2 If we are at the node u equal to $\mathit{Find}(\mathit{Entry}(i-1),k,i-1)$, $i>1$, then $\mathit{Advance}(k,u,i)$ will bring us to v equal to $\mathit{Find}(\mathit{Entry}(i),k,i)$.

Proof Assuming that *Advance* brings us from u to v, it follows a path in the tree; this is not necessarily a direct path from one node to a descendant since copy pointers can also be followed. We call this the *advance path*. The proof is by induction. However, we strengthen the induction hypothesis by adding that no pointer changes can be made to nodes on the advance path except the last.

We prove by induction in the number of modifications in the tree by version i or greater that *Advance* will find the node v. Actually, since nodes are never physically deleted, and pointers are only overwritten if created by the version performing the update, updates in versions greater than i cannot affect the path, so it is sufficient to consider updates by version i.

The base case is when no modifications have taken place. In that case, the start node u and end node v are identical, so *Advance* will certainly find v (immediately after switching to version i, searching in version i is identical to searching in version $i-1$). Additionally, the advance path consists of a single node (which is then last), so the second part of the hypothesis follows trivially.

Assume that some operation changes the advance path. By induction, the change is made by altering or adding a pointer in the last node on the path. We consider the operations in turn.

Assume that an insertion changed the advance path, and assume without loss of generality that the leaf possibly containing k (before the insertion) is to the left. If the insertion is also made to the left, the advance path will continue first by either following the left pointer (if the node in question is from version i), the extra pointer, or, if that was already used, by following the copy pointer and then the left pointer. This brings us to the new internal node which, by the insertion, has become the new parent of the leaf possibly containing k.

Since a new copy is triggered as the last action taken during an insertion, and since updates are always performed in the newest nodes, none of the pointers described in the above can ever be altered. Thus, it is still the case that no pointer changes can be made to nodes on the advance path except the last.

If the insertion is made to the right, then only the copy pointer will be followed (possibly twice) and we are again at the correct location.

Now, we assume that a deletion makes a change to the last node u on the advance path. If u is the internal node which is deleted, the copy pointer is set to point to the correct next location (since the location is actually found by a search in the newest version), and since a deleted node can never be accessed by an updating operation again, no changes can be made to nodes on the advance path except the last. If u is the parent of the internal node to be deleted, the advance path is either not changed (if the extra pointer was used for the update), or the copy pointer from u to the newest version of u becomes part of the advance path.

Finally, we must consider rebalancing operations. As required, they consist of a number of single rotation. Thus, we only need to consider one single rotation. Assume that a single rotation (Figures 3 on page 98 and 7 on page 102) makes a change to the last node on the advance path. If v in Figure 3 on page 98 is the last node, since it keeps it leaf (C), the path is either not changed or it is extended with one copy pointer. For the node u, the situation is similar if A is the leaf possibly containing k. So, assume that B is the leaf in question. This is exactly the case where a copy of u is triggered after the rotation as shown in Figure 7 on page 102. Thus, the new pointer in u leading to the new parent of the leaf possibly containing k cannot be altered again. Thus, except for the new last node (v), nodes on the advance path cannot be changed. □

4 Complexity

The most interesting case to consider is the one where there are many versions, since that is when searching from an entry pointer through each version would be very time consuming. We assume that after a fixed constant number of updates (or earlier) and the rebalancing operations caused by the updates, we change to a new version. We consider the behavior of *Advance* under that restriction.

Complexity of Transcript

Since a copy pointer which is set up during a deletion points to a node which was in the tree at the time (though it may be from an older version), any additional copy pointers from there must be at least as new. This is also the case for copy pointers created by insertions. Since there are only a constant number of updates, the function *Advance* can follow such pointers at most a constant number of times to get from one version to the next.

Rebalancing does not necessarily behave that well. In the following, we consider restrictions on the behavior of rebalancing which lead to the best possible complexities.

A rotation as described earlier which triggers a copying of a node, because the middle subtree is a leaf l, is referred to as an *expensive* rotation, and the rotation is said to be expensive *due to* the leaf l.

Definition 1 A balanced binary search tree scheme is said to have *limited leaf action* if there exists a constant c such that for any leaf l and any update, the number of expensive rotations due to l, which are carried out in response to the update, is bounded by c. □

If a balanced search tree scheme has limited leaf action, then *Advance* will find the correct node in constant time, since, referring to the proof of Theorem 2 on the page before, the advance path is extended with only a constant number of edges for each insertion, deletion, and expensive rotation, and we change version after a bounded number of updates. Thus, the complexity of the transcript operation is $O(\log n + p)$, where n is the number of elements in the start version for the reporting.

It is now interesting to determine which schemes have limited leaf action. Red-black trees [7] do, since, as it is pointed out in [5], rebalancing after an update consists of at most three rotations (single or double). The remaining rebalancing operations are recolorings.

Also treaps [3] have limited leaf action. Only single rotations are used in treaps, and in a sequence of rotations following an insertion, it is always the same node which is the bottom-most node of the two internal nodes in the rotation. It is easy to show that after it has lost at most two leaf children, its children are going to be internal nodes from that point on. Deletions can be viewed as reverse insertions.

AVL-trees [1] and BB[α]-trees [9, 4, 8] also have limited leaf action. As soon as one gets just some fixed constant distance up towards the root from the place of an update, then the relationship between heights and sizes, respectively, of subtrees implies that none of the involved subtrees can be leaves.

Other Complexity Considerations

Searching in a transcript tree is clearly of the same asymptotic complexity as in [5].

For updates, the additions only increase the complexity by a constant factor, since the work carried out in connection with pointer updating and triggering of copies, by Theorem 1 on page 99, is amortized constant per pointer change. Though setting the copy pointer in connection with a deletion requires a search, this action is taken in connection with an update, so the total complexity of updates remains bounded by the height of the tree.

If the transcript operation is implemented using binary search trees without balance constraints, the complexity of the transcript operation becomes $O(h + p)$, where h is the height of the first version from which we report, and updates become $O(h)$, where h is the height of the tree at the time of the update.

From the discussion above, it also follows that search trees where the number of pointer changes in response to an update is amortized constant will use space only linear in the number of updates. This applies to red-black trees and BB[α]-trees, for instance. The expected number of rotations carried out in response to an update in a treap is two, so space consumption for treaps is expected linear.

5 Concluding Remarks

The status of a key is not necessarily changed in every version. Thus, when following a copy pointer, we may skip over a large number of versions. This can be exploited to give a running time which can sometimes be significantly better. The algorithm would then only print a line whenever the status of the key had changed. It would be interesting to investigate this behavior more closely; both theoretically, but also empirically under some average use of search trees.

A chronological transcript involving more than one key could be produced by merging separate transcripts. However, *Transcript* has been expressed via *Advance* which reports one change at a time. Thus, the total chronological transcript can be produced directly by maintaining a priority queue with an entry for each key and the version number as the priority. This is more “on-line”, i.e., it would start printing out the transcript earlier.

If the number of pointers is not a concern, then there is an easier solution: decide that the standard search tree should have all the parents of leaves connected in a doubly linked list. By [5], we obtain the optimal asymptotic amortized complexities. However, more pointers are needed. The doubly linked list uses additional pointers, but since nodes can now have more than one predecessor (there are now 3), nodes must be equipped with predecessor pointers (see [5]) such that all nodes referring to an updated node can themselves be updated. Additionally, there must be more extra fields corresponding to the number of predecessors (the amortized constant results from [5] depend on this). Though the amortized time complexity would be the same, the worst case complexity of an update would be $\Omega(n)$ instead of $O(\log n)$ obtained by a balanced transcript tree. This happens when all the extra fields in all the nodes in the doubly linked list have been used. In that case, every node in the list must be copied.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An Algorithm for the Organisation of Information, *Doklady Akadamii Nauk SSSR*, in Russian, 146, 263–266, 1962; english translation in *Soviet Math. Doklady*, 3, 1259–1263, 1962.

[2] A. Andersson, Improving Partial Rebuilding by Using Simple Balance Criteria, In Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro (eds.), *1st Wokshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, Vol. 382, 393–402. Springer-Verlag, 1989.

[3] C. R. Aragon and R. G. Seidel, Randomized Search Trees, In *Proceedings of the 30th Annual IEEE Symposium on the Foundations of Computer Science*, 540–545, 1989.

[4] N. Blum and K. Mehlhorn, On the Average Number of Rebalancing Operations in Weight-Balanced Trees, *Theoretical Computer Science*, 11, 303–320, 1980.

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, Making Data Structures Persistent, *Journal of Computer and System Sciences*, 38, 86–124, 1989.

[6] I. Galperin and R. L. Rivest, Scapegoat Trees, In *4th ACM-SIAM Symposium on Discrete Algorithms*, 165–174, 1993.

[7] L. J. Guibas and R. Sedgewick, A Dichromatic Framework for Balanced Trees, In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, 8–21, 1978.

[8] K. Mehlhorn, *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*, Springer-Verlag, 1986.

[9] J. Nievergelt and M. Reingold, Binary Search Trees of Bounded Balance, *SIAM Journal on Computing*, 2:1, 33–43, 1973.

```
func Go(u: Node, dir: Dir, i: Version): Node
    if u.extra ≠ nil and u.extra.dir = dir and u.extra.vn ≤ i then
        return u.extra.ptr
    else
        return (dir = Left ? u.left : u.right)

func Find(u: Node, k: Key, i: Version): Node
    v := (k ≤ u.key ? Go(u, Left, i) : Go(u, Right, i))
    return (Leaf(v) ? u : Find(v, k, i))

func Status(u: Node, k: Key, i: Version): Bool
    dir := (k ≤ u.key ? Left : Right)
    if u.extra ≠ nil and u.extra.dir = dir and u.extra.vn ≤ i then
        return u.extra.ptr.key = k
    else
        return (dir = Left ? u.left.key : u.right.key) = k

func Advance(k: Key, u: Node, i: Version): ⟨Bool, Node⟩
    u := Find(u, k, i)
    while u.copy ≠ nil and u.copy.vn ≤ i do
        u := Find(u.copy.ptr, k, i)
    return ⟨Status(u, k, i), u⟩

proc Transcript(k: Node, v1, v2: Version)
    u := Entry(v1)
    for i := v1 to v2 do
        ⟨s, u⟩ := Advance(k, u, i)
        print i, k, s
```

Fig. 8: The transcript operation.