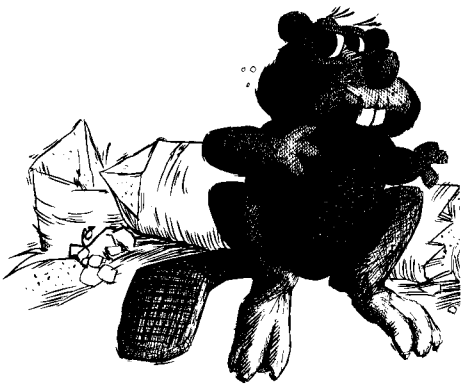
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT



Updating Binary Trees with Constant Linkage Cost

Thomas Ottmann
and
Derick Wood

Data Structuring Group
Research Report CS-89-45

October, 1989

Updating Binary Trees with Constant Linkage Cost *

Thomas Ottmann† Derick Wood‡

Abstract

We provide a unifying framework for balanced binary trees that enables us to obtain general conditions under which insertions and deletions have a constant number of promotions and, hence, a constant number of link changes. At the same time, the update algorithms are also logarithmic in the worst case.

This general result provides insight into the constant linkage cost update algorithms for red-black, red-h-black, and half-balanced trees. Moreover, it enables us to design new constant linkage cost update algorithms for these classes of trees as well as for other classes. Specifically, we are able to give constant linkage cost algorithms for α-balanced trees.

1 Introduction

Binary search trees have been extended to maintain not only the ordering of records by their primary keys, but also, simultaneously, orderings on secondary keys. The best known example of such a search structure are the priority search trees of McCreight [8]. They are search trees with respect to their primary keys and priority trees with respect to their secondary keys; thus, enabling them to answer queries of the form: return all records whose primary key lies in some range and whose secondary key is greater than some bound. Other example structures are the persistent search trees of Sarnak and Tarjan [12] and the dynamic contour trees of Frederickson and Rodger [5].

Extended search trees introduce new maintenance problems; both orderings have to be maintained when updates are performed. If the underlying

*This work was supported under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and under a grant from the Information Technology Research Centre.

†Institut für Informatik, Universität Freiburg, Rheinstraße 10-12, D-7800 FREIBURG, WEST GERMANY.

‡Data Structuring Group, Department of Computer Science, University of Waterloo, WATERLOO, Ontario N2L 3G1, CANADA.

tree is not balanced, the problems are minor; however, we normally use balanced trees to ensure $O(\log n)$ behavior in the worst case. In this case, we use promotions (or rotations) to restructure the given tree after an update. As is well known, promotions preserve the binary search ordering, but not, in general, the secondary key ordering. For example, in a priority search tree each promotion can cause one "trickle down" operation to re-establish the secondary key ordering. Since the height of the tree is $O(\log n)$, a trickle down operation requires $O(\log n)$ time in the worst case. If an update produces p promotions, then, in the worst case, it takes $O(p \log n)$ time. Moreover, if $p = O(\log n)$, then an update takes $O(\log^2 n)$ time in the worst case, which is unacceptable. For this reason, McCreight did not base priority search trees on AVL trees, since deletion requires $O(\log n)$ promotions in the worst case. Instead he chose the half-balanced trees of Olivié [10], since they need at most three promotions in the worst case for both insertion and deletion. We call such a class of trees and the associated updating algorithms *constant linkage cost* (*CLC*, for short).

The half-balanced trees were the first class demonstrated to be constant linkage cost. As was shown later by Tarjan [13], the half-balanced trees are the same as the symmetric binary B-trees of Bayer [3], but the updating algorithms in [3] are not CLC. Indeed, they are also equivalent to the red-black trees of Guibas and Sedgewick [6], but the update algorithms given in [6] are also not CLC. The updating algorithms for red-black trees given in Sarnak and Tarjan [12] are, however, CLC.

But why are red-black trees CLC? Are AVL trees or weight-balanced trees CLC? In examining the CLC demonstrations in [10, 12, 13] no underlying principles are exposed; the algorithms appear just as the rabbit appears from the proverbial magician's hat. In this paper, we provide the principle that underlies the CLC red-black update algorithms. For this purpose we introduce a general framework for binary trees related to the stratified trees of [14] and to the dichromatic framework of [6]. We define classes of binary trees to be made up of strata; the trees appearing in the strata are chosen from a given finite set; the boundary nodes of each stratum are colored black; and the interior nodes of each stratum are colored red. The coloring is used solely to identify the strata, nothing more.

Within the general framework, we argue inductively that CLC update operations cause some boundary nodes to cross their boundaries—a push-up or a pull-down effect—and, under appropriate conditions, boundary crossings do not cause any link changes. The usefulness of this general result on CLC update operations is demonstrated by showing that the red-black update algorithms of [12] fall within this framework; CLC update algorithms for the red-h-black trees of Icking et al. [7] (independently discovered by Andersson [2]) are obtained; and, finally, CLC update algorithms for the

classes of α-balanced trees of Olivié [9] are derived, thereby solving (at least partially) a problem left open by him.

In Section 2 we briefly recall the definitions in [14] and adapt them to our current needs. We present generic update algorithms in Section 3 and show in Section 4 that the algorithms for red-black trees and red-h-black trees are special cases of the generic procedure. Once the right spectacles are worn, it is easy to obtain algorithms for new classes of balanced trees which require a constant number of link changes for each insertion or deletion and which are tunable to performance. Then, we define a class of trees that has CLC deletion, but not CLC insertion, thereby demonstrating that deletion is no more difficult than insertion—the opposite of what is usually assumed. In the remainder of the section, we prove that none of our classes of trees is AVL or k-height balanced, for any $k > 0$. Finally, we close, in Section 5, with some open problems.

2 Stratification

Though it is not necessary for our theory, we assume for simplicity that trees are binary. Recall that a binary *tree* T of n nodes is either

(i) the *empty* tree if $n = 0$, or

(ii) a triple (u, T_l, T_r) if $n > 0$, where u is a binary internal node with left subtree T_l of n_l nodes and a right subtree T_r of n_r nodes, where $n = n_l + n_r + 1$. The node u is the *root* of T.

The *height* of a tree T of n nodes, denoted by $height(T)$, is defined recursively as either 0 if $n = 0$ or $1 + max(\{height(T_l), height(T_r)\})$, if $n > 0$, where $T = (u, T_l, T_r)$. Similarly, the *weight* of a tree T of n nodes, denoted by $weight(T)$, is defined as 1 if $n = 0$ or $weight(T_l) + weight(T_r)$ if $n > 0$, where $T = (u, T_l, T_r)$. The nullary nodes of a tree are usually called *external nodes* or *leaves*, while the other nodes are said to be *internal nodes*. The weight of a tree is simply the number of external nodes in it.

A (binary) *search tree* is a tree containing items with keys from an ordered universe in its nodes, one item per node. Usually items are stored only in the internal nodes. The nullary nodes represent intervals between keys and the items are arranged in symmetric order. If x is any node, the key of the item in x is greater than the keys of all items in its left subtree and less than the keys of all items in its right subtree. (One can also adopt the convention that items are stored only in external nodes while the internal nodes contain routing information, see [11] for example.) In any case, we can perform an *access* operation by starting at the root and recursively descending the left or right subtree, if the query key is less than or greater than the key

stored at the root, respectively. The search terminates either successfully, once we have found an item with the query key, or unsuccessfully, if we fall out of the tree at the expected position among the leaves.

For simplicity, we do not distinguish between search trees with stored items and their underlying graphical structure. Stratification means that we view a tree as consisting of a number of *strata* or *layers*. The tree can be decomposed into a small "irregular" part of bounded size at the top, the *apex*, and a number of strata which are glued together at their borders. Each stratum consists of small trees of a prespecified set of *stratum trees*.

Let Z be a finite set of trees, $l_Z = min(\{weight(T) : T \in Z\})$, and $h_Z = max(\{weight(T) : T \in Z\})$; we call Z a *stratum set*. Observe that we do not require that all trees in Z are of the same height. But usually we require Z to be *nontrivial*; that is, $1 < l_Z < h_Z$. An *apex* set is simply a nonempty finite set of trees. To define a class of stratified trees inductively, we use a tree constructor. Let T_0 be a tree with weight t, let $T_1, \ldots, T_t$ be trees, and let T_0's external nodes be enumerated from left to right from 1 to t. Then, we denote by $T_0[T_1, \ldots, T_t]$ the tree obtained by replacing, for all i, $1 \leq i \leq t$, the i-th external node of T_0 with T_i.

Let Z be a stratum set and A be an apex set; so far we do not require any specific properties of Z and A except that both sets must be finite. We define the class of (A, Z)-*stratified* trees to be the smallest class of trees such that

(i) each tree in A is said to be (A, Z)-stratified, and

(ii) if T_0 is (A, Z)-stratified and has weight t, then $T_0[T_1, \ldots, T_t]$ is (A, Z)-stratified, for all $T_1, \ldots, T_t$ in Z.

The class of (A, Z)-stratified trees is denoted by $S(A, Z)$. Because we did not restrict the possible apexes and trees in stratum sets so far, there may be different decompositions for a given tree $T \in S(A, Z)$. We may, however, consider trees with different decompositions as different and assume that with each tree $T \in S(A, Z)$ its decomposition into an apex from A and strata with trees chosen from Z is explicitly at hand. The easiest way to ensure this is to color boundary nodes and the root node black, and the interior nodes of the strata and apex red.

We can, therefore, define the *stratum height*, denoted by $sh(T)$, as follows: $sh(T) = 0$ if $T \in A$ and $sh(T) = 1 + sh(T_0)$ if $T = T_0[T_1, \ldots, T_t]$.

We can consider a tree $T \in S(A, Z)$ as a multiway tree if we "collapse" the trees in the apex and in the stratum set, respectively, into single nodes. All internal collapsed nodes except possibly the root have degree d where $l_Z \leq d \leq h_Z$, $l_Z = min(\{weight(T) : T \in Z\})$, and $h_Z = max(\{weight(T) : T \in Z\})$. By definition, all leaves of this corresponding multiway tree have the same distance to the root, namely $sh(T)$. If Z is nontrivial, we have

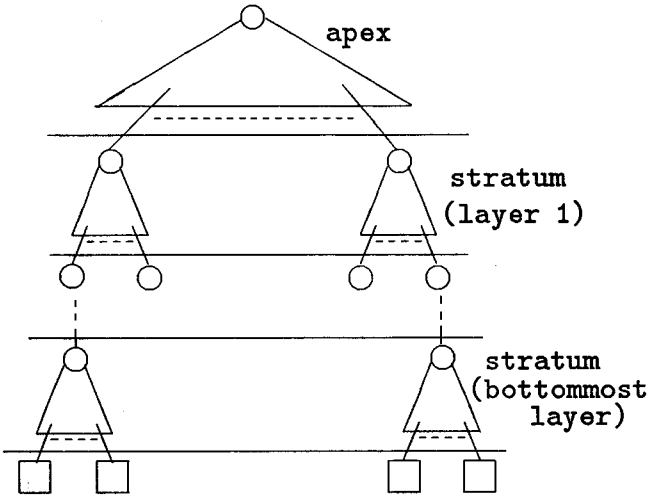


Figure 1: Decomposition of a stratified tree

$l_Z \geq 2$ and, therefore, $sh(T) = O(\log N)$, for $T \in S(A, Z)$, where N is the total number of collapsed nodes in T. Clearly, N differs from the total number n of nodes in T by at most a constant factor. Similarly, expanding collapsed nodes back to trees in Z and A, respectively, increases the height by at most a constant factor. This gives the following theorem.

Theorem 2.1 *Let Z be a nontrivial stratum set and let A be an apex set. Then, for each tree $T \in S(A, Z)$, $sh(T) = O(\log n)$ and $height(T) = O(\log n)$, where $n = weight(T)$.*

If $sh(T) > 0$, all leaves of the apex of T and all leaves of stratum trees except those at the bottommost level are identified with roots of stratum trees chosen from Z. Throughout this paper we visualize decompositions of stratified trees as shown in Figure 1. That is, we draw lines just above the leaves of the apex and above the leaves of the trees in the stratum set.

Example 2.1 Let A and Z be identical sets of trees containing exactly the four trees shown in Figure 2. Then $S(A, Z)$ coincides with the class of symmetric binary B-trees [3] which equals the class of half-balanced trees [10] and the class of red-black trees [6]. Figure 3 shows a decomposition of a tree $T \in S(A, Z)$ with stratum height 2.

We say that a tree *complete* if its leaves appear on at most two different levels. The stratum and apex trees of Example 2.1 as well as the tree shown in Figure 3 are complete trees.

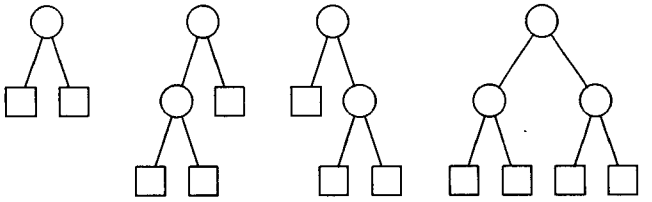


Figure 2: Stratum and apex sets defining the class of symmetric binary B-trees

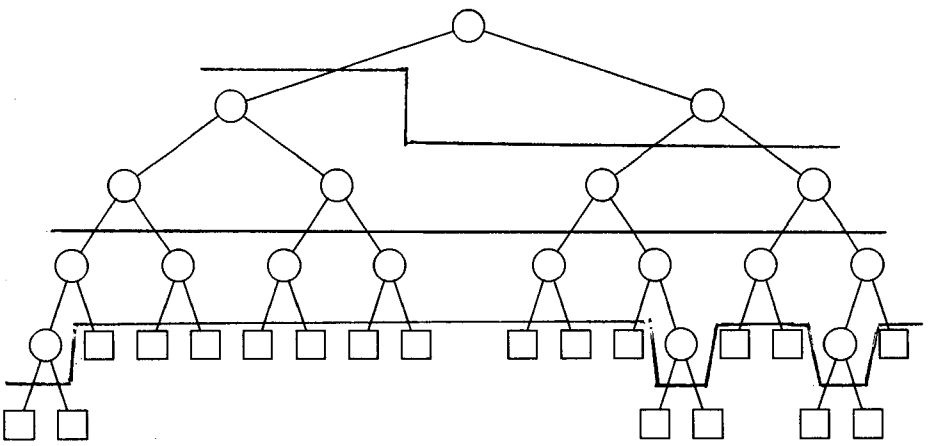


Figure 3: Decomposition of a tree of stratum height 2

Example 2.2 Let A_h be the set of all complete trees of height at most $h+1$ and let Z_h be the set which contains the complete tree of weight 2^h and height h and, furthermore, all complete trees of weight j, for $2^h + 1 \leq j \leq 2^{h+1}$, and height $h+1$. For $h = 1$ we obtain once more the sets A and Z of Example 2.1. The class $S(A_h, Z_h)$ is identical with the class of red-h-black trees introduced in [7].

Van Leeuwen and Overmars [14] imposed additional constraints on apex and stratum sets which ensure that there exist update procedures for (A, Z)-stratified trees that can be performed in $O(\log n)$ steps. One of their conditions is that all trees in the stratum set Z must be of the same height. Hence, the stratum trees of Examples 2.1 and 2.2 do not fit into their framework. We will see that it is not necessary to require stratum trees to be of equal height in order to obtain update procedures which take $O(\log n)$ time. If we want to achieve update procedures that require only constant link changes for each insertion or deletion, the condition of van Leeuwen and Overmars is even prohibitive. This will become clear in Section 3.

For appropriately chosen A and Z, the class $S(A, Z)$ may be identical with a known class of balanced search trees. Sometimes, $S(A, Z)$ will be only a proper subclass of a known class X of balanced search trees. If $S(A, Z)$ can be maintained in $O(\log n)$ steps with CLC, the same may not be true for the class X but only for a subclass, namely $S(A, Z)$. Moreover, the update algorithms obtained by specializing the generic procedures for stratified trees may be different from any known update procedure specifically designed for X.

3 Updating stratified trees

We want to design generic CLC-update algorithms for (A, Z)-stratified trees. This requires that we look at the structure of the apex and stratum trees in some detail. We cannot simply argue from the weights of trees and ignore their structure as van Leeuwen and Overmars [14] do. However, the general flavor of both algorithms is similar and resembles the well known maintenance algorithms for B-trees [4].

Whenever an (A, Z)-stratified tree is given, we assume that its decomposition into an apex chosen from A and strata (or layers) of trees chosen from Z is explicitly at hand. We describe the generic maintenance algorithms and, simultaneously, derive conditions which ensure that they can be carried out with constant linkage cost.

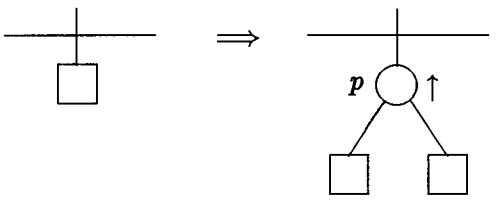


Figure 4: Insertion of a new item at the leaves

3.1 Insertion

In order to insert a new item with key x into an (A, Z)-stratified tree, we first search for x. Because we assume that the item is not yet present in the tree we will fall out of the tree at the expected leaf. We replace the leaf by an internal node for the new item and call *push_up*; see Figure 4. The task of the procedure *push_up* is to push a binary node that is just below a stratum border immediately above it. Whenever *push_up*(p) is called, p is either a leaf of an apex tree chosen from A or a leaf of a stratum tree chosen from Z. The first case applies if and only if p is just below the topmost stratum border of the given tree. A call of *push_up*(p) may lead to local restructuring of the related stratum or apex tree and termination, or to a recursive call at the root of a stratum tree. More precisely, we proceed as follows.

Let p be a leaf of tree T, $T \in A \cup Z$. Then, *push_up*(p) triggers the following actions. First, we change the stratum border just above p to make p a node immediately above the border; see Figure 5 for the case $T \in Z$. Note that this local change of a stratum border does not involve any structural change in the tree. All parent-child relationships between the nodes of the tree remain unchanged. Let T_p^+ denote the tree obtained by replacing the leaf p by an internal node with two leaves as its children. Clearly, $weight(T_p^+) = weight(T) + 1$. If T_p^+ is also in Z and the locally changed stratum border is not the topmost one, we have finished. Similarly, *push_up*(p) terminates if T_p^+ is in A and the locally changed stratum border is the topmost one. Therefore, let us assume that $T_p^+ \notin Z$ and that the changed stratum border is not the topmost one. If there is a $T' \in Z$ such that $weight(T') = weight(T_p^+)$, then replace T_p^+ by T' and append all subtrees hanging from the leaves of T_p^+ below the stratum border to the leaves of T' in the same left-to-right order. Note that this transformation involves a structural change of the tree. A finite number of parent-child relationships between the nodes of the tree has to be changed. The number is bounded by the size of the tree T_p^+. After this transformation is completed, *push_up*

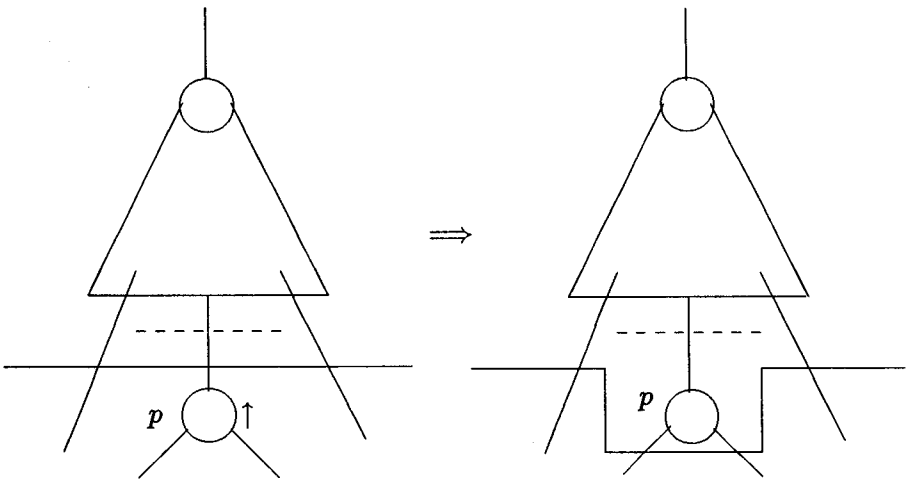


Figure 5: Local change of a stratum border during insertion

terminates. If the locally changed stratum border is the topmost one and $T_p^+ \notin A$, but, if there is a $T' \in A$ such that $weight(T_p^+) = weight(T')$, then we proceed analogously. Let us now consider the case that the locally changed stratum border is not the topmost one, $T_p^+ \notin Z$, and there is no tree $T' \in Z$ such that $weight(T_p^+) = weight(T')$. The tree T_p^+ has a binary root q and a left subtree T_l and a right subtree T_r. Let us assume that there are trees T_l' and T_r' in Z with $weight(T_l) + weight(T_r) = weight(T_l') + weight(T_r')$. Then, we can replace T_l and T_r by T_l' and T_r', assign the subtrees hanging from the leaves of T_l and T_r to the leaves of T_l' and T_r' in the same left-to-right order, and recursively call $push\_up(q)$. This transformation is illustrated in Figure 6. It obviously resembles the splitting of an overflowing node in the B-tree maintenance algorithm [4]. Note that the transformation carried out in this case involves structural changes of the tree if and only if $T_l \neq T_l'$ or $T_r \neq T_r'$. In the case that the locally changed stratum border is the topmost one, we proceed similarly. That is, let us assume that $T \in A$, but $T_p^+ \notin A$ and that there is no tree T' in A with $weight(T_p^+) = weight(T')$. In this case we assume that there are a $k \geq 2$ and k trees $T_1, \ldots, T_k \in Z$ such that $weight(T_p^+) = \sum_{i=1}^{k} weight(T_i)$. Furthermore, we assume that A contains a tree of weight k. These assumptions allow us to introduce a new layer. That is, we replace T_p^+ by $T_1, \ldots, T_k$, assign the subtrees hanging from the leaves of T_p^+ to the leaves of $T_1, \ldots, T_k$, and, finally, assign the roots of $T_1, \ldots, T_k$ to the leaves of an apex tree of weight k. This transformation is illustrated in Figure 7. Note that the transformation carried out in this case

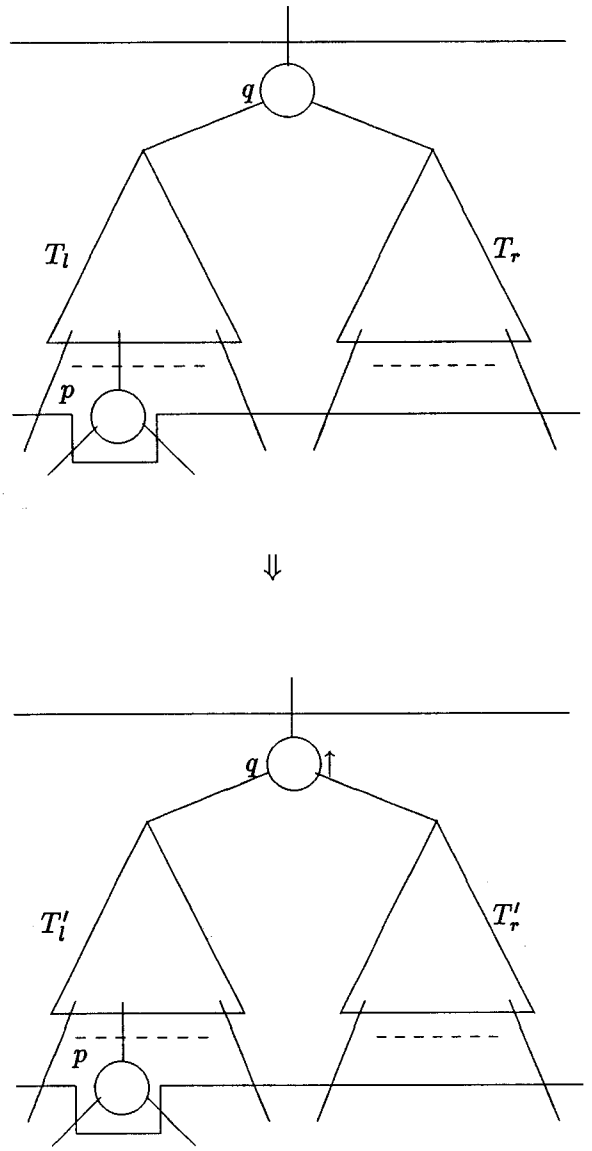


Figure 6: Splitting a stratum tree

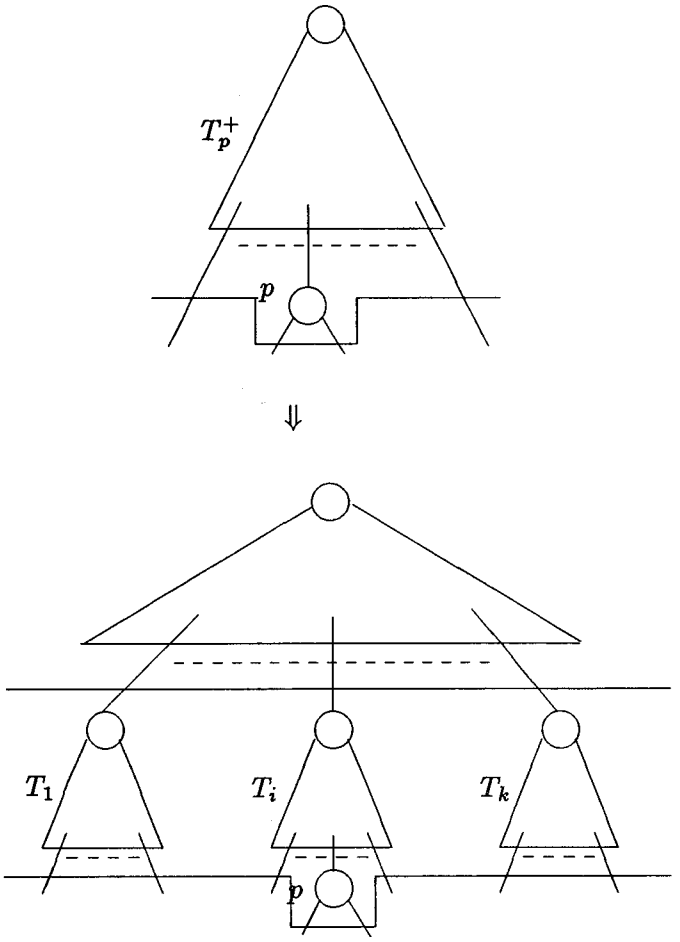


Figure 7: Splitting an apex tree

once again involves structural changes of the tree. However, the amount of work to be done is bounded by the size of T_p^+. The procedure *push_up* terminates after this transformation.

The general picture is now clear. For appropriately chosen sets of apex and stratum trees a call of *push_up* either leads to a finite number of structural changes and termination or leads to a recursive call at the next higher layer. The recursive call is preceded by structural changes if and only if the two subtrees of the stratum tree containing the pushed-up node do not belong to the set of stratum trees. In either case, the amount of work for each insertion into a tree of stratum height s is $O(s)$. The number of link changes is constant if and only if the recursive call of *push_up* involves no structural changes.

Let us call a class Z of stratum trees *weight-insertion closed* or *wi-closed*, for short, if each tree $T \in Z$ satisfies one of the following conditions (i) and (ii).

(i) There is a tree $T' \in Z$ with $weight(T') = weight(T) + 1$.

(ii) There are trees T_l and $T_r \in Z$ such that $weight(T) + 1 = weight(T_l) + weight(T_r)$.

An apex set A is *wi-closed with respect to* Z if each $T \in A$ satisfies one of the following conditions (i) and (ii).

(i) There is a tree $T' \in A$ with $weight(T') = weight(T) + 1$.

(ii) There is a $k \geq 2$ and there are k trees $T_1, \ldots, T_k \in Z$ such that $weight(T) + 1 = \sum_{i=1}^{k} weight(T_i)$ and A contains a tree of weight k.

We are now in a position, after providing one further notion, to define the condition under which insertions are CLC. Let T be a tree in Z of maximal weight k and, for $1 \leq i \leq k$, let T_i be the tree obtained from T by replacing its i-th leaf with an internal node having two leaves as its children. Now, let T_i^l and T_i^r denote the left and right subtrees of T_i, for $1 \leq i \leq k$.

A stratum set Z is *structurally insertion closed* or *si-closed*, for short, if Z is wi-closed and Z contains not only the tree T but also all trees T_i^l and T_i^r, for $1 \leq i \leq k$.

These notions allow us to summarize our above arguments by the following theorem.

Theorem 3.1 *Let Z be a wi-closed stratum set and let A be an apex set which is wi-closed with respect to Z. Then, insertion into an arbitrary (A, Z)-stratified tree can be carried out in $O(\log n)$ steps. Furthermore, if the stratum set Z is si-closed, insertion is CLC.*

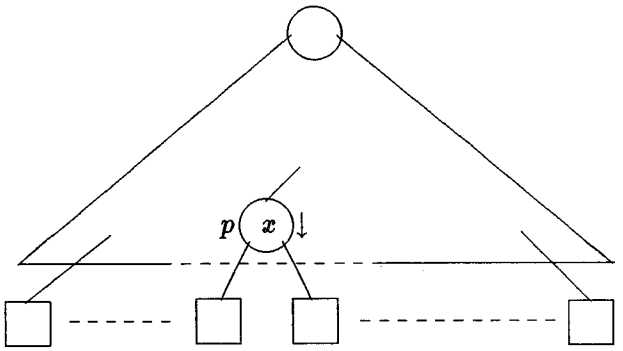


Figure 8: Deletion of an item

Let $l_Z = min(\{weight(T) : T \in Z\})$, $h_Z = max(\{weight(T) : T \in Z\})$, and $1 < l_Z < h_Z$. We call a stratum set *gap-free* if for each t with $l_Z \leq t \leq h_Z$, there is a tree $T \in Z$ with $weight(T) = t$. Van Leeuwen and Overmars [14] assume that stratum sets are gap-free. They also give numeric conditions for the weights and heights of trees in the apex set which ensure that the apex set is wi-closed with respect to a given stratum set. Note that Z can be both wi-closed and have gaps.

3.2 Deletion

In order to delete an item from an (A, Z)-stratified tree, we first locate the item by a search starting at the root. We may assume that the item is stored in an internal node just above the bottommost stratum border; that is, in a node which has only leaves as its children. For, by taking successors or predecessors in symmetric order, we can always reduce a deletion to this case. Now we are faced with a problem which can be considered as the reverse of the one occurring during insertion. An internal node just above a stratum border has to be pulled down below it. In other words, the stratum tree containing this node loses one of its internal nodes and one of its leaves; see Figure 8. We could design the procedure *pull_down* by making only very weak assumptions about the apex and stratum sets as we did in the case of insertion and the procedure *push_up*. However, in order to keep things simple, we will be more specific than necessary. We assume that the stratum set Z is nontrivial and gap-free and that the apex set A contains a tree T of weight t, for each t with $1 \leq t \leq max(\{h_Z, 2l_Z \quad 1\})$. We say that Z is *weight-deletion closed* or *wd-closed*, for short, and A is *wd-closed with respect to* Z if they satisfy the above conditions.

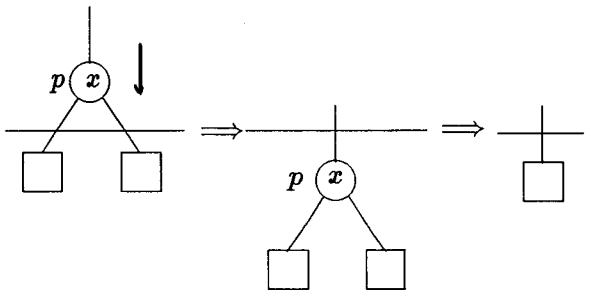


Figure 9: Deletion at the leaves

A call of *pull_down*(p) may lead to a local structural change and termination or may lead to a recursive call for a node one layer higher up in the tree. We derive conditions for Z which ensure that no structural changes precede the recursive call. When *pull_down*(p) is called for the first time, p is a node immediately above a stratum border. We will ensure that the same condition holds for every further call of *pull_down*. A call of *pull_down*(p) triggers the following actions.

First, we locally change the stratum border just below p and make p a node immediately below the border. When this transformation has been carried out for the first time at the bottommost level, we replace p by a leaf and delete the item stored in p; see Figure 9. Note that the local change of the stratum border does not involve any structural change in the tree. We must, however, ensure on higher levels that the tree with root p is a stratum tree in order to maintain the whole tree in $S(A, Z)$.

Let p be a node of stratum tree T and $weight(T) = t$. By pulling p down below the stratum border we have changed T into a tree T_p^- of weight $t-1$. If $t > l_Z$, we can replace T_p^- by a Z-tree and halt. This requires a constant number of link changes. If $t = l_Z$, we consider all siblings of T_p^- in the same stratum. If at least one of them is of weight greater than l_Z or if there are more than $l_Z - 1$ siblings, then we can redistribute the elements such that all the, at least l_Z, subtrees become trees in Z. It requires the restructuring of at most $h_Z + 1$ subtrees and takes only a constant number of steps.

If there are only $l_Z - 1$ neighboring siblings and they are all "minimally" filled, that is they all have weight l_Z, we have to call *pull_down* recursively. However, we want to ensure that no structural changes precede the recursive call. This goal leads naturally to the following definition.

A stratum set Z is *structurally deletion closed* or *sd-closed*, for short, if it is wd-closed and satisfies the following conditions (i) and (ii).

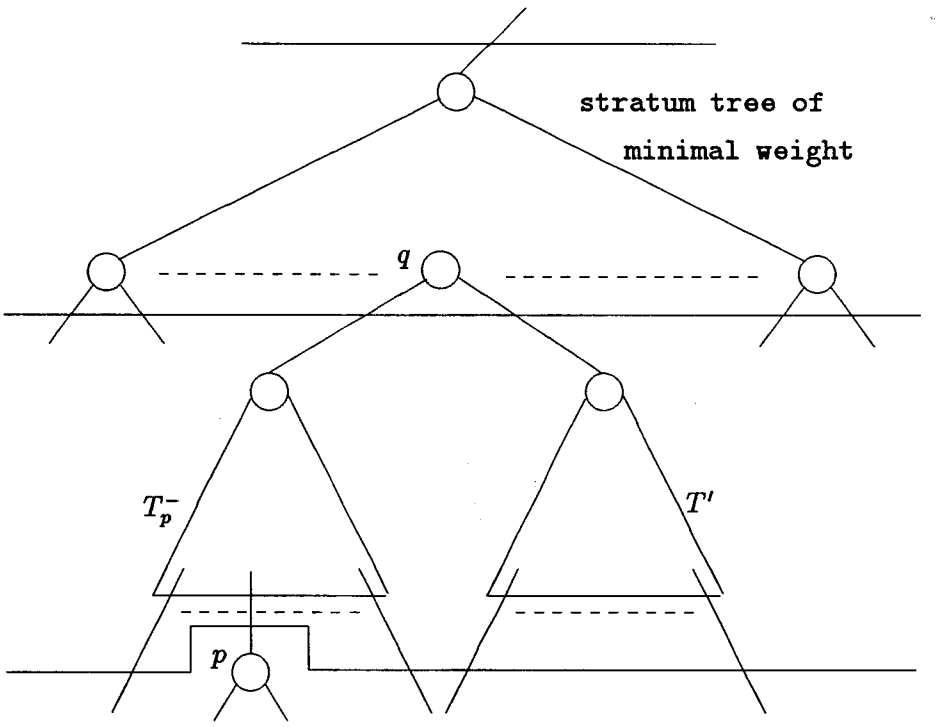


Figure 10: Merging of two minimally filled stratum trees

(i) Each tree $T \in Z$ with $weight(T) = l_Z$ has the property that each leaf of T has a leaf node as its sibling. (In other words, each parent of a leaf has only leaves as its children.)

(ii) For each tree $T \in Z$ with $weight(T) = l_Z$ and, for each tree T^- obtained from a tree in Z of minimal weight by replacing an arbitrary internal node with two leaves as its children by a leaf, the trees (u, T, T^-) and (v, T^-, T) also belong to Z, where u and v are new root nodes.

Let us now assume that Z is sd-closed. Then, the situation to be dealt with can be depicted as shown by Figure 10. Our assumptions ensure that there is a node q just above the next stratum border which has T_p^- and a minimally filled stratum tree T' as its subtrees. Therefore, we can call $pull\_down(q)$. The tree (q, T_p^-, T') or (q, T', T_p^-) is a stratum tree by the assumption that Z is sd-closed. The procedure is repeated in this way until it either finishes or reaches the apex. The trees in the first layer and the apex can always be rebuilt in a constant number of steps to form a tree in $S(A, Z)$. As long

as T_p^- has at least one sibling T' on the first layer with $weight(T') > l_Z$, rebuilding does not decrease the stratum height. If T' has only l_Z leaves, T' is the only sibling of T_p^- on the first layer, and if both are subtrees of a node q, the tree (q, T_p^-, T') or (q, T', T_p^-) can be rebuilt to form a new apex of weight $2l_Z - 1$. In this last case the stratum height decreases by one.

The general picture of the deletion procedure is this. Initially, the procedure *pull_down*(p) is called for the node p which holds the item to be deleted. Then *pull_down* is recursively called for a sequence of nodes in minimally-filled stratum trees which have the minimal number of siblings, all minimally-filled, on the same stratum. As soon as a node in a stratum tree is reached which does not have this property, finite restructuring terminates the deletion. It is clear that the total amount of work to be done is $O(\log n)$ and that only a constant number of linkage changes are required if the stratum set is sd-closed. We summarize our discussion in the following theorem.

Theorem 3.2 *Let Z be a wd-closed stratum set and A be an apex set which is wd-closed with respect to Z. Then, deletion of an item from an arbitrary (A, Z)-stratified tree can be carried out in time $O(\log n)$. Furthermore, if Z is sd-closed, deletion is CLC.*

There is still a considerable amount of freedom left by the generic update procedures. If we wish to obtain CLC updates, the above insertion and deletion procedures tell us exactly what to do in the cases of recursive calls of *push_up* and *pull_down*, respectively. However, the restructurings to be carried out in the remaining cases can be done in different ways depending on the variety of trees in the stratum and apex set. This freedom does not affect the asymptotic run-time of the update algorithms. It may, however, be utilized to fine-tune the algorithms for specific classes of trees. An obvious goal would be to minimize the number of required relinkings.

4 Applications

4.1 Symmetric binary B-trees (SBB-trees)

Consider the apex and stratum sets of Example 2.1. It is easy to check that Z is both si-closed and sd-closed. The apex set A is wi-closed and wd-closed with respect to Z. Therefore, insertions and deletions can be carried out in $O(\log n)$ time and updates are CLC. It is interesting to specialize the generic update algorithms in this case and to compare them with others in the literature. Omitting symmetric cases, insertion is completely captured by the transformations shown in Figure 11. It shows the transformations only for stratum trees; the same transformations apply to apex trees analogously.

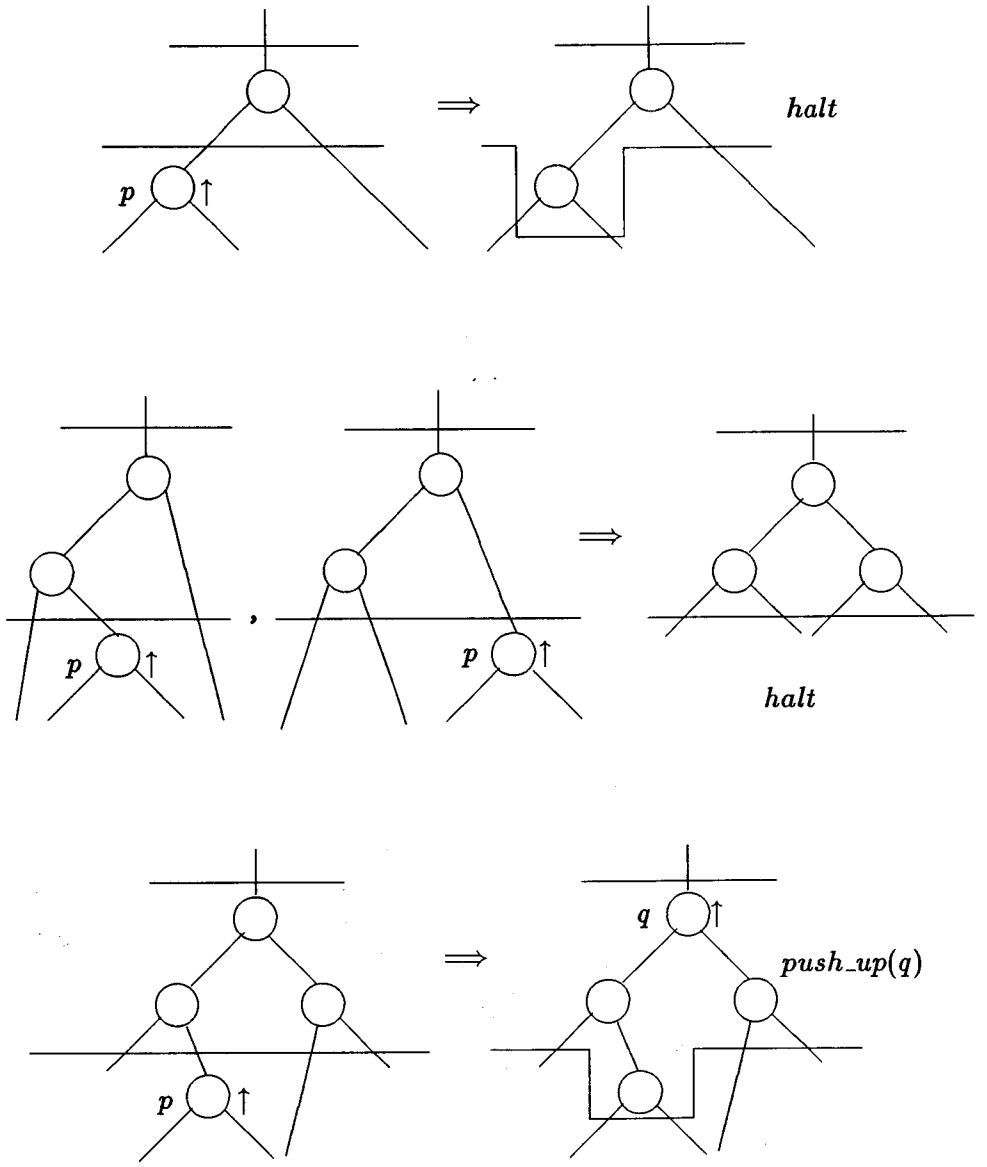


Figure 11: Insertion into symmetric binary B-trees

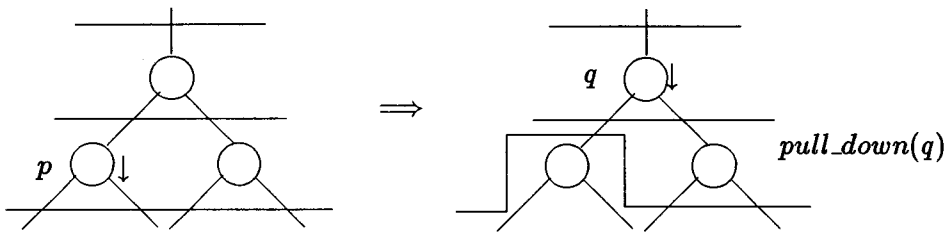


Figure 12: Recursive call of *pull_down* in deletion algorithm for symmetric binary B-trees

Deletion requires a recursive call of *pull_down* if and only if the situation shown by Figure 12 or its symmetric variant applies. In all other cases, we can locally rebuild trees in two adjacent strata or in the topmost stratum and the apex such that an SBB-tree is obtained. Figure 13 shows two of the many possible cases. Note that there is still some freedom in the choice of restructuring.

4.2 Red-h-black trees and α-balanced binary trees

Consider the apex and stratum sets A_h and Z_h of Example 2.2. For each $h \geq 1$, A_h is wi-closed and wd-closed with respect to Z_h. The stratum sets Z_h are both si-closed and sd-closed for all $h \geq 1$. Therefore, it is clear that all classes $S(A_h, Z_h)$ have $O(\log n)$ time CLC-update algorithms. This property is utilized in [7] to build external priority search trees with trees from $S(A_h, Z_h)$ as underlying search trees.

For a node p in a tree $T \in S(A_h, Z_h)$, the difference between the length l_p of a longest path from node p to a leaf and the length s_p of a shortest path from node p to a leaf can become arbitrarily large. But it should be obvious that the asymptotic value of the quotient s_p/l_p is bounded from below by $h/(h+1)$. For, Z_h has only trees of heights h and $h+1$ and the worst we can do is to append repetitively stratum trees of height h in one subtree of p and stratum trees of height $h+1$ in the other.

In his thesis [9], Olivié takes the quotient of the two path lengths as a balance criterion and requires that quotient to be between certain limits. More precisely, let α be a real value such that $0 \leq \alpha \leq 1$. A binary tree T is an *α-balanced binary tree* or an *αBB tree*, for short, if for each node p of T, the length s_p of a shortest path from p to a leaf and the length l_p of a longest path from p to a leaf satisfy the following conditions.

(i) $0 \leq \alpha \leq s_p/l_p \leq 1$, if $l_p \geq \frac{1}{1-\alpha}$, and

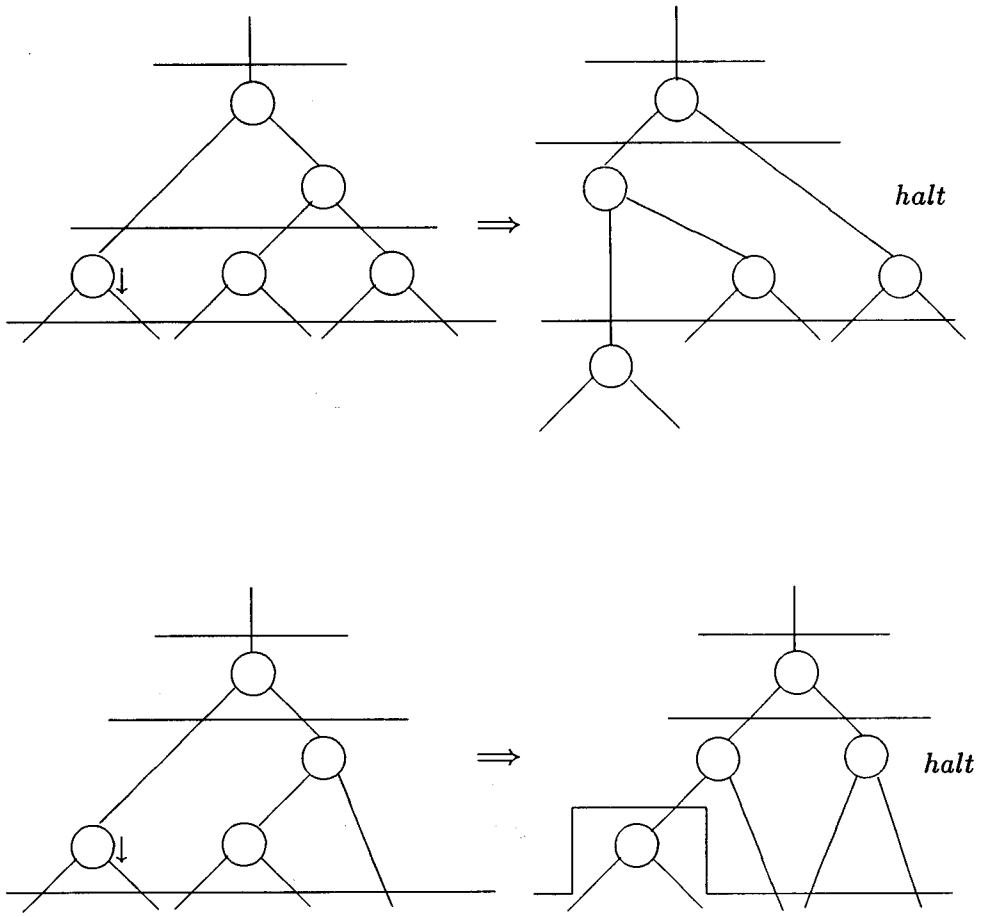


Figure 13: Restructuring of adjacent strata during deletion of an item from a symmetric binary B-tree

(ii) $l_p - 1 \leq s_p$, if $l_p < \frac{1}{1-\alpha}$.

For $\alpha = 0$ we get all binary trees, for $\alpha = 1$ we get the complete binary trees (then only condition (ii) applies), and for $\alpha = 1/2$ we get the half-balanced binary trees [10]. For $0 \leq \alpha \leq 1/2$, condition (i) is the only condition to be considered, as $1/(1-\alpha) \leq 2$. We say a node p *α-balanced* if the subtree with root p is α-balanced.

If the apex set A_h in Example 2.2 contained only complete binary trees, where all leaves have the same distance to the root, we could immediately infer that $S(A_h, Z_h)$ is a class of α-balanced trees with $\alpha = h/(h+1)$. Because this is not the case, we obtain a slightly weaker result.

Theorem 4.1 *For each $h \geq 1$, the stratified class of red-h-black trees $S(A_h, Z_h)$ is a class of α-balanced trees with $\alpha = h/(h+2)$.*

Proof: If $T \in S(A_h, Z_h)$ and $height(T) \leq h+1$, all leaves of T must appear on at most two adjacent levels. Hence, for each node p in T, the length s_p of a shortest path from p to a leaf and the length l_p of a longest path from p to a leaf differ by at most one. Therefore, condition (ii) holds if $l_p < 1/(1-\alpha) = (h+2)/2 \leq h+1$. If $l_p \geq (h+2)/2$ and $l_p \neq s_p$, we have $l_p \geq 2$, $s_p = l_p - 1$ and therefore

$$\frac{s_p}{l_p} = \frac{l_p - 1}{l_p} \geq \frac{h}{h+2};$$

that is, condition (i) holds.

Let us now assume that $T \in S(A_h, Z_h)$ and that $height(T) > h+1$. Then, T consists of an apex tree chosen from A_h and a number of strata with stratum trees chosen from Z_h. If the node p is a node of a stratum tree in the bottommost stratum, we have $l_p \leq h+1$ and l_p and s_p can differ by at most one. Therefore, we can argue as before and conclude that p is α-balanced. If p is not a node of a stratum tree in the bottommost stratum, then there exist integers $r \geq 1$ and s, $0 \leq s \leq h$, such that $l_p \leq r(h+1)+s+1$ and $s_p \geq rh + s$. Therefore, we can conclude

$$\frac{s_p}{l_p} \geq \frac{rh+s}{r(h+1)+s+1} \geq \frac{h}{h+2} = \alpha;$$

that is, condition (i) holds. □

For some specific values of α, Olivié has shown that α-balanced trees have $O(\log n)$ time CLC-update algorithms. The case $\alpha = 1/2$ is contained in [10] and the case $\alpha = 1/3$ is presented in [9] and it is left as an open problem whether there exist other classes of α-balanced trees with CLC-update procedures. Our results show that there are an infinite number

of these classes with α in the range $1/3 \leq \alpha \leq 1$. In particular, one can choose α to be arbitrarily close to 1 and still obtain CLC-update algorithms. However, a note of caution is on order. We do not claim that for arbitrarily chosen values of α, $1/3 \leq \alpha \leq 1$, there exist $O(\log n)$ time CLC-update algorithms for the class of α-balanced trees. Rather, we have shown that there is a subclass of any class of α-balanced trees that has the desired property. The subclass is, in general, a proper subclass that it is closed under insertions and deletions. For example, consider the class $S(A_1, Z_1)$. From Theorem 4.1 we infer that this class is 1/3-balanced. But we know already, see Example 2.1, that $S(A_1, Z_1)$ coincides with the class of 1/2-balanced trees.

We can also apply our theory in a slightly different manner and obtain an "approximate" answer to the question of whether there exist $O(\log n)$ time CLC-update algorithms for arbitrary classes of α-balanced trees. Let us assume that α is rational; that is, $\alpha = r/s$, where r and s are integers with $r < s$. Then, it is easy to find an apex set $A_{r,s}$ and a stratum set $Z_{r,s}$ such that $S(A_{r,s}, Z_{r,s})$ is both si-closed and sd-closed and approximates the class of r/s-balanced trees in the following sense. For each tree $T \in S(A_{r,s}, Z_{r,s})$ and for each node p, the asymptotic value of the quotient s_p/l_p is bounded from below by r/s, where l_p and s_p denote the lengths of a longest and a shortest path, respectively, from p to a leaf; that is,

$$\lim_{l_p \to \infty} s_p/l_p \geq r/s = \alpha.$$

Choose $Z_{r,s}$ to be the set ot trees which contains exactly:

(1.1) the complete binary tree of weight 2^s and height s and the complete binary tree of weight 2^{s-1} and height $s-1$,

(1.2) all complete binary trees of weight $2^{s-1}+1$ and height s,

(2.1) the complete binary tree of weight 2^r and height r,

(2.2) all complete binary trees of weight $2^{r+1}-1$ and height $r+1$, and

(3) for all weights t with $t \notin \{2^r, 2^{r+1}-1, 2^{s-1}, 2^{s-1}+1, 2^s\}$ and $2^r \leq t \leq 2^s$ a (complete) binary tree of weight t and height h with $r \leq h \leq s$.

Choose $A_{r,s}$ to be the set of all complete binary trees of weight t with $2 \leq t \leq 2^s$ and height h with $1 \leq h \leq s$. Then, it is obvious that $S(A_{r,s}, Z_{r,s})$ has the desired property. The trees in this class can be maintained in $O(\log n)$ time with CLC-update algorithms.

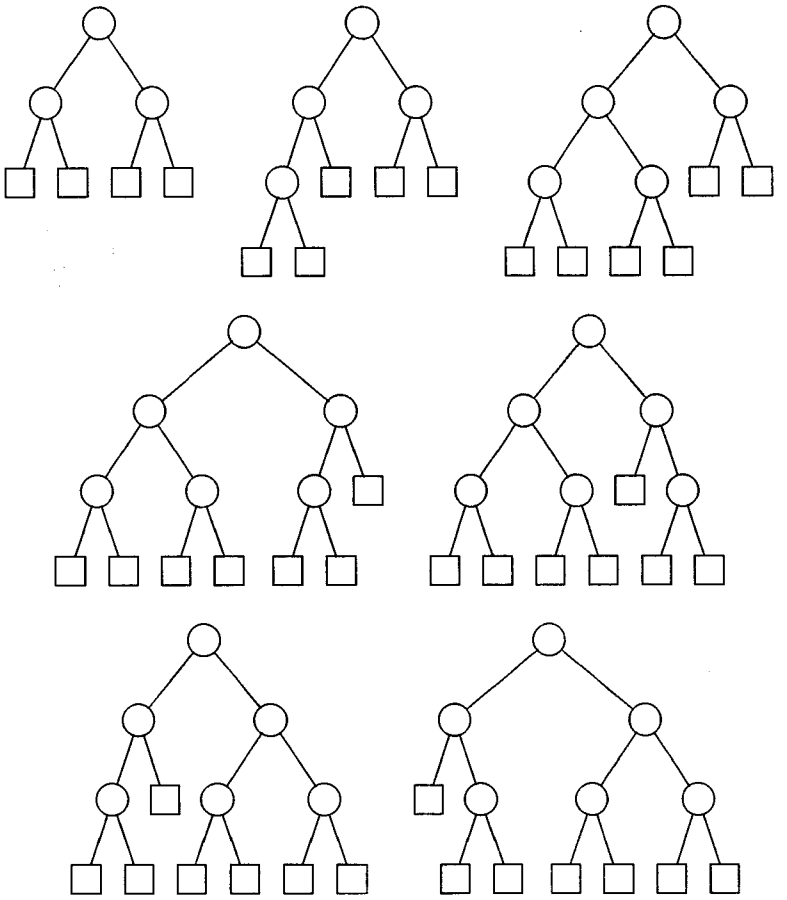


Figure 14: An *sd*-closed stratum set

4.3 Easy deletion

Let Z be the set of 7 trees of weights 4, 5, 6, and 7 shown in Figure 14. Let A be an appropriately chosen apex set for Z. (One may choose A to contain all trees with k leaves with $1 \leq k \leq 7$.) It is easy to check that Z is sd-closed but not si-closed. The set Z is, however, wi-closed. Hence, $S(A, Z)$ can be maintained in time $O(\log n)$. Additionally, it has a CLC-deletion algorithm, but a single insertion may require $\Theta(\log n)$ structural changes. This is an example of a class of balanced trees for which deletion appears to be simpler than insertion. No such class of trees has been reported before.

4.4 Height balanced trees

The attentive reader may have noticed already that none of our example classes of (A, Z)-stratified trees is height balanced. This is not accidental as the next theorem shows. For an integer $k \geq 1$, we call a tree *k-height balanced* if, for each node p in the tree, the heights of p's left and right subtrees differ by at most k. The 1-height balanced trees are also called *height balanced* or AVL-trees [1].

Theorem 4.2 *Let Z be a set of stratum trees which is si-closed or sd-closed and let A be a nonempty apex set. Then, for all $k \geq 1$, $S(A, Z)$ is not a class of k-height balanced trees.*

Proof: Let Z be si-closed and $T \in Z$ with $weight(T) = h_Z = max(\{weight(T) : T \in Z\})$. Let T_l and T_r be the left and right subtrees of T, respectively. Choose a leaf in T_l and replace it by an internal node with two leaves as its children. Denote the tree obtained from T_l in this way by T_l^+. Because Z is si-closed, both T_l^+ and T_r must belong to Z. Note that $height(T) \geq 1 + height(T_r)$. Now choose a binary node p just above the leaves in an apex tree. Append $k+1$ strata to the apex by always appending T to all leaves in the left subtree of p and appending T_r to all leaves in the right subtree of p. Then, p becomes a node of a tree in $S(A, Z)$ such that the heights of p's left and right subtree differ by at least $k + 1$. This shows that the tree is not k-height balanced. If Z is sd-closed, the argument is similar. For, sd-closure also implies that the stratum set must contain at least two trees of different heights. □

5 Concluding Remarks

It is an immediate consequence of Theorem 3.1 that the standard insertion algorithm for AVL-trees which requires only three promotions for each insertion does not fit into our framework. Furthermore, this leaves open the

interesting question: Do there exist CLC-update algorithms for a class of k-height balanced trees which require $O(\log n)$ time regardless of whether the update is an insertion or deletion?

References

[1] G.M. Adel'son-Vel'skii and Y.M. Landis. An algorithm for the organization of information. *Doklady Akademi Nauk*, 146:263–266, 1962.

[2] A. Andersson. Binary search trees of almost optimal height. Technical Report LU-CS-TR: 88:41, Department of Computer Science, Lund University, Lund, Sweden, 1988.

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] G. Frederickson and S. Rodger. A new approach to the dynamic maintenance of maximal points in the plane. In *Proceedings of the 25th Annual Allerton Conference on Communication, Control, and Computing*, pages 879–888, Urbana-Champaign, Illinois, 1987.

[6] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[7] Chr. Icking, R. Klein, and Th. Ottmann. Priority search trees in secondary memory. In *Graphtheoretic Concepts in Computer Science (WG '87), Staffelstein, Lecture Notes in Computer Science 314*, pages 84–93, 1987.

[8] E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[9] H. J. Olivié. *A Study of Balanced Binary Trees and Balanced One-Two Trees.* PhD thesis, Departement Wiskunde, Universiteit Antwerpen, Antwerp, Belgium, 1980.

[10] H. J. Olivié. A new class of balanced search trees: Half-balanced search trees. *RAIRO Informatique théorique*, 16:51–71, 1982.

[11] Th. Ottmann, M. Schrapp, and D. Wood. Purely top-down updating algorithms for stratified search trees. *Acta Informatica*, 22:85–100, 1985.

[12] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[13] R.E. Tarjan. Updating a balanced search tree in O(1) rotations. *Information Processing Letters*, 16:253–257, 1983.

[14] J. van Leeuwen and M. Overmars. Stratified balanced search trees. *Acta Informatica*, 18:345–359, 1983.