Purely Top-Down Updating Algorithms for Stratified Search Trees

Thomas Ottmann
Michael Schrapp
Derick Wood

Data Structuring Group
CS-84-30

September, 1984
PURELY TOP-DOWN UPDATING ALGORITHMS FOR STRATIFIED SEARCH TREES

Thomas Ottmann
Michael Schrapp
Derick Wood

ABSTRACT

The existence of purely top-down updating algorithms for balanced search trees is of importance when maintaining such trees in a concurrent environment, where purely top-down means a single sweep from the root to frontier along a search path. We present algorithms for internal- and external-search trees in the general framework of stratified trees. This enables us to demonstrate that many classes of balanced search trees have such updating schemes, although, for example, weight-balanced trees do not fit into this framework.

1. INTRODUCTION

When search trees are to be maintained in a concurrent environment it is advantageous, as [GS] point out, if updating can be carried out in a single root-to-frontier scan of the associated search path. The reason for this is that a simple locking protocol can then be used in which a "window" (of a fixed and predetermined size) of locked nodes is moved down the search path. This is, for example, always the case for insertion into a binary search tree, because a search either determines that the given key is already present, when no further action need be taken, or it determines the external node that should be replaced by an additional internal node. However deletion from a binary search tree is not always carried out in a single scan if the usual technique is used, see [AHU2, K]. The usual method of deleting a given key at a node u with two non-external

---

(1) Work carried out partially under NATO Grant No. RG 155.81 and the work of the third author was partially supported by Natural Sciences and Engineering Research Council of Canada Grant No. A-5692.

(2) Institute für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6380, D-7500 Karlsruhe, W. Germany.

(3) Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.
children is to replace the key at \( u \) with the maximal key in \( u \)'s left subtree and then delete this value from \( u \)'s left subtree. Thus the deletion algorithm can back up the search path an unbounded number of nodes. Fortunately it is possible to modify this deletion algorithm, so that it has a single scan, by first using rotations to move \( u \) to the frontier, see Figure 1.1, and removing \( u \) when one of its children becomes an external node.

![Figure 1.1](image)

Such updating algorithms are said to be purely top-down updating algorithms. For binary search trees such algorithms are easy to obtain as shown above, but when given balanced binary search trees it is unclear whether or not such algorithms even exist! To understand why this may be so we need the notion of a redundant update. An insertion of a key \( x \) into a search tree \( T \) is redundant if \( x \) is already present in \( T \) before the insertion. Similarly the deletion of a key \( x \) from a search tree \( T \) is redundant if \( x \) is not present in \( T \) prior to the deletion. A redundant update is either a redundant insertion or a redundant deletion.

When inserting a key into a height-balanced search tree, say, with a purely top-down algorithm, it must ensure that the resulting search tree is height-balanced whether or not the insertion is redundant. This concern with redundant updates is the reason for the complexity of the VLSI dictionary machine [ORS] compared with the simple systolic search tree of [BeK]. In the case of height-balanced search trees purely top-down updating algorithms can be found. However for weight-balanced trees [NR] the existence of purely top-down updating algorithms remains tantalizingly open. Although [NR] give purely-top-down updating algorithms for weight-balanced trees, these algorithms do not handle redundant updates. Similarly [Z] gives a top-down deletion algorithm for 2-3 trees, which on closer examination requires two scans. Finally the algorithms in [GS] are only for simple routing schemes. Furthermore the insertion algorithm only works for even order B-trees and the hints for the construction of a deletion algorithm are insufficient.
2. SEARCH TREES AND ROUTING SCHEMES

Recall that a tree $T$ of $n$ nodes is either:

(i) the empty tree if $n = 0$; it is denoted by a null node, or:

(ii) an $(m+1)$-tuple $(u, T_1, \ldots, T_m)$ if $n > 0$, where $u$ is a $m$-ary internal node, for some $m \geq 1$, with subtrees $T_1, \ldots, T_m$, (in left-to-right order), of $n_1, \ldots, n_m$ nodes, respectively, where $n = 1 + n_1 + \cdots + n_m$. The node $u$ is the root of $T$.

The height of a tree $T$ of $n$ nodes, denoted by $\text{height}(T)$, is defined recursively as either 0 if $n = 0$ or $1 + \max\{\text{height}(T_i) : 1 \leq i \leq m\}$ if $n > 0$, where $T = (u, T_1, \ldots, T_m)$. Similarly the weight of a tree $T$ of $n$ nodes, denoted by $\text{weight}(T)$, is defined as 1 if $n = 0$ or $\text{weight}(T_1) + \cdots + \text{weight}(T_m)$ if $n > 0$, where $T = (u, T_1, \ldots, T_m)$.

The nullary nodes of a tree are usually called external nodes (or leaves), while the remaining nodes are said to be internal nodes. The weight of a tree is simply the number of external nodes it has.

In order to illustrate the basic notions required in this paper we use the following class of trees as a running example.

A binary-ternary tree is a tree in which every node is either nullary, binary or ternary. See Figure 2.1 for an example of such a tree.

![Figure 2.1 A Binary-Ternary Tree](image)

The notion of a search tree is well known, see [AHU1, AHU2, K] for example. The basic idea is that an internal $m$-ary node is associated with $m-1$ keys from some totally-ordered universe of keys. We take the universe of keys to be the rationals throughout this paper. However there exist two distinct ways of associating sets of keys with a tree to give a search tree. The most popular is to associate keys with internal nodes giving the internal-search tree, see Figure 2.2 for an example of a binary-ternary internal-search tree. However it is also possible to associate keys with external nodes giving the external-search tree, see Figure 2.3 for an example of a binary-ternary external-search tree. In this case it is necessary to provide routing or separating keys (also called routers in [KW]) in
the internal nodes so that searching can still be carried out correctly. Often, but not always, the set of separating keys is a subset of the set of keys in the search tree. In the literature there are a number of routing schemes available for assigning such separating keys. We consider four methods. Figure 2.3 illustrates the left-maximum scheme. The separating key of a binary node \( u \) is the maximum key in \( u \)'s left subtree. For a ternary node \( u \) the first separating key is the maximum key in \( u \)'s left subtree and the second is the maximum key in \( u \)'s middle subtree. In general the \( i \)th separating key is the maximum key in the \( i \)th subtree. As a second method Figure 2.4 illustrates the right-maximum scheme, for which the \( i \)th separating key of a node is the maximum key in the \( (i+1) \)st subtree. When searching an external-search tree which has this routing scheme the decision to continue the search in the first or second child of a node can only be made after referring to the first child's final separating key. The third method, the \((\leq, <)\) scheme, is illustrated in Figure 2.5. In this routing scheme the \( i \)th separating key of a node \( u \) is a key which is both greater than or equal to the maximum key in the \( i \)th subtree of \( u \) and less than the minimum key in the \((i+1)\)st subtree of \( u \). Fourth, and finally we have the \((<, <)\) scheme in which the \( i \)th separating key of a node \( u \) lies strictly between the maximum key in \( u \)'s \( i \)th subtree and the minimum key in \( u \)'s \((i+1)\)st subtree. In this scheme the separating keys and the set of keys represented by the search tree are disjoint, see Figure 2.6.
Whatever the routing scheme the action to be taken during a search for a query key must be uniquely and correctly determined by the separating keys. This is the case for each of the four routing schemes introduced above as the interested reader can readily verify.

The deletion of a key in an external-search tree which has either the left-
maximum or right-maximum routing scheme can cause the routing key of some
node(s) on the search path to be modified. For example deleting the key 6 in Fig-
ure 2.3 causes the separating key of the root to be modified, while in Figure 2.4 it
causes the separating keys of two nodes to be modified. It is not immediately
clear that a purely top-down deletion algorithm exists for either of these schemes
since the new separating key, the predecessor of 6 in each case, is only discovered
on reaching the frontier. The \((\leq, <)\) scheme on the other hand does not have
this difficulty since the separating keys do not have to be keys in the tree, hence
the deletion of the key 6 in Figure 2.5 has no effect on the separating key of the
root. The \((\leq, <)\) scheme is said to be simple, while the other three schemes are
non-simple (in [KW] these are said to be clean and dirty, respectively). Both the
left-maximum and right-maximum schemes are non-simple, but only deletion
causes difficulty so we say they are insertion-simple, while the \((<, <)\) scheme
causes difficulties for insertion so it is deletion-simple. In Figure 2.6 observe that
insertion of 7 requires the root value to be changed to a successor of 7 not in the
tree. Such a value cannot be computed until the frontier is reached, so it is
unclear whether or not a purely top-down insertion algorithm exists in this case.

3. STRATIFIED TREES

Let \(X\) be some given set of trees and \(\alpha\) be a positive integer. Then \(X\) is
said to be \(\alpha\)-proper if for each integer \(t \geq \alpha\) there is at least one tree \(T\) in \(X\)
with \(\text{weight}(T) = t\). This implies that for every set of keys of size \(t \geq \alpha\), there is
an external-search tree and an internal-search tree from \(X\) for the given set.
Most sets of trees are 1-proper, for example the set of binary-ternary trees.

Stratified trees have, as their name implies, strata or layers which consist of
trees of the same height, a height specified in advance. Let \(Z\) be a set of trees of
the same height \(\beta\), \(l_Z = \min\{\text{weight}(T) : T \text{ in } Z\}\), and
\(h_Z = \max\{\text{weight}(T) : T \text{ in } Z\}\). We call such a \(Z\) a stratum set. In Figure 3.1
a stratum set for the set of binary-ternary trees is displayed, where \(l_Z = 6\) and
\(h_Z = 8\).

![Figure 3.1 A Stratum Set](image-url)
Apart from some initial portion of a stratified tree called the apex, a stratified tree consists of layers of Z-trees. Let \( K = \max(\alpha l_Z, \lceil (l_Z - 1)/(h_Z - l_Z) \rceil l_Z) \) and let \( \gamma \) be the smallest integer such that for all \( t, \alpha \leq t \leq K \), there is a tree \( T \) in \( X \) with \( \text{weight}(T) = t \) and \( \text{height}(T) \leq \gamma \). (The value of \( K \) is one more than the value of \( K \) in the original definition \([vLO]\). This is necessary in the proofs of Theorems 4.2 and 5.1.) Let \( A = \{ T \in X : \text{height}(T) \leq \gamma \} \). We say \( A \subseteq X \) is an apex set for the given \( Z \). Figure 3.2 displays an apex set for the set of binary-ternary trees and the stratum set of Figure 3.1.

Figure 3.2 An Apex Set

To define a set of stratified trees inductively we need a tree constructor, which we now introduce. Let \( T_1, \ldots, T_t \) be trees, \( T \) be a tree with weight \( t \), and let \( T \)'s external nodes be enumerated in left to right order from 1 to \( t \). Then we denote by \( T[T_1, \ldots, T_t] \) the tree obtained by replacing, for all \( i, 1 \leq i \leq t \), the \( i \)th external node of \( T \) with \( T_i \).

Figure 3.3 A Constructor Example

We say a stratum set \( Z \) is an acceptable if
(i) \(1 < l_z \leq h_z\),
(ii) \(\{t : l_z \leq t \leq h_z\} = \{\text{weight}(T) : T \text{ in } Z\}\), and
(iii) for all \(T\) in \(X\), where \(t = \text{weight}(T)\), and all \(T_1, \ldots, T_t\) in \(Z\), \(T[T_1, \ldots, T_t]\) is in \(X\).

The stratum set of Figure 3.1 is acceptable.

**Definition** Let \(X\) be a set of trees, \(Z\) be an acceptable stratum set, and \(A\) the apex set for \(Z\). Then the set of \(Z\)-stratified trees, with respect to \(X\) and \(A\), is the smallest set of trees that satisfy:

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

The set of \(Z\)-stratified trees is denoted by \(S(X, Z)\).

In the following proposition we summarize the main results to be found in [vLO].

**Proposition 3.1** Let \(X, Z, A\), and \(S(X, Z)\) be as defined above. Then:

1. \(S(X, Z) \subseteq X\).
2. \(S(X, Z)\) is \(\alpha\)-proper.
3. For all trees \(T\) in \(S(X, Z)\), \(\text{height}(T) = O(\log(\text{weight}(T)))\).
4. \(S(X, Z)\) is a logarithmically-maintainable set of trees, that is if \(T\) in \(S(X, Z)\) is a search tree, then an insertion into or a deletion from \(T\) and the subsequent restructuring to again obtain a tree in \(S(X, Z)\) can be carried out in \(O(\log(\text{weight}(T)))\) steps.

Thus the \(Z\)-stratified trees are balanced subsets of \(X\). As [vLO] point out, if \(X\) is one of the well known sets of (height-) balanced trees, for example AVL trees, 2-3 trees, and son trees, then it has a \(Z\)-stratified subset. However as is proved in [OSW2] the weight-balanced trees of [NR] do not have such \(Z\)-stratified subsets.

4. **SIMPLE ROUTING SCHEMES**

In this section we consider simple routing schemes, since this enables us to concentrate on the main idea of purely top-down updating algorithms. In order to achieve purely top-down updating algorithms we must ensure that once the frontier of the given tree has been reached, the insertion or deletion of an external node not only results in a stratified tree but also this is accomplished solely by local restructuring. In other words we only allow restructuring within a bounded region or window \(R\) of the external node being considered, indeed the window is
always two layers deep and consists of a $Z$-tree and the $Z$-trees attached to its external nodes, except when in layers 0 and 1 when it consists of an apex and the $Z$-trees attached to its external nodes. See Figure 4.1.

![Diagram of a generic situation](image)

Figure 4.1 The Generic Situation, $2 \leq i \leq f$

One way of implementing this is to ensure that the $Z$-tree containing the external node in question should have neither the minimum nor the maximum weight. Since $l_Z$ may equal $h_Z - 1$ such a $Z$-tree does not always exist. We show below that we may, without loss of generality, assume $l_Z \neq h_Z - 1$, so we proceed on this assumption. In order to perform the updating operation we may either leave the $Z$-tree unchanged in the case of a redundant operation or replace it by a $Z$-tree with either one more or one less external node giving a new stratified tree. This leaves the problem of identifying the $Z$-tree of the desired weight in the proper position, that is where the addition or removal of a key has to be carried out. As we don't know the structure of the tree at the frontier when starting at the root, we must take into account the possibility that either a maximum or minimum weighted $Z$-tree may occur.

We do this by building a $Z$-tree of the desired weight higher up in the given tree (close to the apex) and, essentially, moving it down the search path. This is similar to the strategy used in the modified deletion algorithm in Section 1. But first we show that $l_Z = h_Z - 1$ need not occur.

**Lemma 4.1** Let $X, Z, A, K, \beta, \gamma$ be as above.

Let $X, Z$ and $S(X, Z)$ define a set of $Z$-stratified trees. Then there is a $\bar{Z}$ such that $S(X, \bar{Z})$ is a set of $\bar{Z}$-stratified trees with $3 < l_{\bar{Z}} < h_{\bar{Z}} - 4$.

**Proof:** Treat two $Z$-tree layers as one $\bar{Z}$-tree layer. Thus $\beta = 2\beta$, $l_{\bar{Z}} = l_Z^2$, and $h_{\bar{Z}} = h_Z^2$. Clearly $S(X, \bar{Z})$ forms a set of $\bar{Z}$-stratified trees. Moreover since $l_Z \geq 2$ we have $l_Z > 3$ and since $h_Z > l_Z \geq 2$ we have
That Proposition 3.1 holds for \( S(X, Z) \) follows directly from its validity for \( S(X, Z) \). However the values of \( K \) and \( \gamma \) need to be changed as does apex set. We now return to the updating algorithm.

The following algorithm formalizes the ideas expressed above. Let \( T \), the given stratified tree, be decomposed into its apex \( T_0 \) and \( f \) layers. Let \( T_i \) be the tree in layer \( i \) on the search path. Let \( w_i \) denote the weight of \( T_i \), \( w_T \) the weight of \( T \), and \( w_R \) the weight of a window \( R \). Note that the height of a layer is the height of the \( Z \)-trees in the layer while the height of the apex is known when constructing the tree from the empty tree. {Every time a new layer is formed, the height of the apex is updated, if necessary. In other words the stratification of the tree is known globally.) Let \( z \) be the key to the inserted or deleted, see Figure 4.2. Since, in simple routing schemes the set of routing values need not be a subset of the keys represented by the search tree it is not necessary to modify them during updating. Of course in the final layer a routing value may need to be removed or added simply because a key has been deleted or inserted, respectively, but this is the only exception. Therefore it is only necessary to ensure that \( T_f \) satisfies \( l_Z < w_f < h_Z \) so that redundant updates can be accommodated. However, as already pointed out, this is insufficient for either deletion in an internal search tree or updating a search tree with a non-simple routing scheme. For in this non-simple case we also have to provide the means of forcing some key or routing value down the tree from one layer to another. This is simply because a key or routing value at an internal node may need to be deleted as a result of the update. For this reason we introduce a more complex updating algorithm for the simple case which provides for a uniform treatment of the simple and non-simple cases. The additional complexity arises because we require that \( l_Z < w_i < h_Z - 1 \), for all \( i \), \( 1 \leq i \leq f \) — the invariant. The reasons for this requirement are dealt with in Section 5. Furthermore as each layer is dealt with we first require an even stronger condition, namely \( l_Z < w_i < h_Z - 3 \) — the strengthened invariant. This is a non-empty interval since, by Lemma 4.1, we may always assume \( h_Z - 4 > l_Z \). The stronger requirement always enables us to position \( T_i \) to include the search path for \( z \), but at the expense of \( T_i \) gaining at most 2 further nodes, in which case \( l_Z < w_i < h_Z - 1 \) still holds. Then UPDATE\( (x, T) \) begins at the root of \( T \) (and, hence, of \( T_0 \)).

**Algorithm** UPDATE\( (x, T) \);

begin

if \( \text{weight}(T) \leq K \) then

(1) \{If the update is not redundant then construct a new \( Z \)-stratified tree \( T' \) with \( \text{weight}(T) + 1 \) or \( \text{weight}(T) - 1 \) external nodes containing the updated set of keys\}

else

begin \{There must be at least one layer\}


Figure 4.2 The Search Path for \( x \)

\[
i := 1; \quad \{ \text{The current layer} \}
\]

\[
\{ \text{Determine } T_1 \text{ from the apex } T_0 \text{ and } x \} \]

\[
\text{if layer one does not satisfy the invariant then} \]

\[(\text{II}) \quad \{ \text{Completely restructure } T_0 \text{ and layer 1 with at most one extra layer so that the layers satisfy the invariant, and } T_1 \text{ and possibly } T_2 \text{ remain on the search path}; \}
\]

\[
\{ \text{Update the height of the apex}; \}
\]

\[
\text{if an extra layer has been added then begin } i := 2; \quad f := f + 1 \quad \text{end;}
\]

\[
\text{repeat}
\]

\[
i := i + 1;
\]

\[
\{ \text{Determine the next Z-tree } T_i \text{ from the search path of } x \text{ in } T_{i-1} \}
\]

\[
\text{if layer } i \text{ does not satisfy the invariant then}
\]

\[(\text{III}) \quad \{ \text{Rebuild } T_{i-1} \text{ and all the trees of layer } i \text{ appended to } T_{i-1} \text{ such that } T_i \text{ remains on the search path and layer } i \text{ satisfies the invariant.} \}
\]

\[
\text{until } i = f; \quad (\text{IV}) \quad \{ \text{Insert or delete } x \text{ in } T_i \text{ if it is a non-redundant update} \}
\]

\[
\text{end}
\]

\[
\text{end } \{ \text{UPDATE} \};
\]

**Theorem 4.2** Let \( S(X, Z) \) be a class of Z-stratified trees with a simple routing scheme. Then for any tree \( T \) in \( S(X, Z) \) and any key \( x \), UPDATE\((x, T)\) is again a tree in \( S(X, Z) \).
Proof: We consider the four possible restructuring separately.

Restructuring I: \( w_T \leq K \).

In this case a new tree of the appropriate size is to be found in \( S(X,Z) \) if one is needed.

Restructuring II: \( T \) has at least one layer.

In this case \( R \) consists of \( T_0 \) and layer one and \( K + 1 \leq w_R \leq Kh_Z \). \( R \) is replaced by an \( R' \) consisting of an apex \( T_0' \) and at most two layers, each satisfying the strengthened invariant. That this is always possible is seen by computing the bounds on \( w_{R'} \). A minimal value of \( w_{R'} \) is obtained when \( R' \) consists of an apex and a single layer. The apex may have weight as small as \( \alpha \) and each \( Z \)-tree in layer one, apart from one, may be as small as \( l_2 \). The remaining \( Z \)-tree has weight at least \( l_2 + 1 \) yielding a total weight of \( \alpha l_Z + 1 \). However since \( w_{R'} = w_R \), we have \( w_{R'} \geq K + 1 \), by assumption and since \( K \geq \alpha l_Z \) this implies \( w_{R'} \geq \alpha l_Z + 1 \).

that is the smallest replacement \( R' \) can be accommodated. Note that this is the only place in the proof where the modified value of \( K \) is needed. The upper bound is obtained when \( R' \) consists of an apex and two layers and these are as large as possible. That is \( w_0 = K \) and the \( Z \)-trees all have weight \( h_Z \) apart from one in each layer with weight \( h_Z - 4 \). Hence:

\[
    w_{R'} \leq (Kh_Z - 4)h_Z - 4
\]

We require that the bounds on \( w_R \) are themselves bounded from below and above by the bounds on \( w_{R'} \) in order that a replacement tree in \( S(X,Z) \) for \( R \), satisfying the strengthened invariant, can be found. Since we have already shown that \( w_R \geq w_{R'} \) it remains to show that:

\[
    Kh_Z \leq (Kh_Z - 4)h_Z - 4
\]

or

\[
    4(h_Z + 1) \leq Kh_Z(h_Z - 1)
\]

Now this holds if:

\[
    4(h_Z + 1)/(h_Z - 1) \leq Kh_Z
\]

or

\[
    4 + 8(h_Z - 1) \leq Kh_Z.
\]

But \( h_Z \geq l_Z + 5 \geq 8 \), by assumption, and \( 8(h_Z - 1) \) is maximized when \( h_Z \) is minimized, that is \( h_Z = 8 \) yielding \( 8/7 \). Now \( 51/7 \leq Kh_Z \) since \( K \geq \alpha l_Z \geq 3 \) and \( h_Z \geq 8 \). Thus we have shown that the apex and first layer can always be reconstructed to satisfy the strengthened invariant. We leave to Lemma 4.3 the task of showing that \( T_1 \) (and \( T_2 \)) can always be positioned to include the search path for \( x \) at the cost of \( w_1 \) (and \( w_2 \)) only satisfying the invariant.

Restructuring III: There are at least two layers.
Now \( i \geq 2 \) and we have the generic situation portrayed in Figure 4.1. If \( w_i \) satisfies \( l_z + 1 \leq w_i < h_z - 1 \) then the invariant holds and no restructuring is necessary. We can always restructure \( R \) to give \( R' \) so that \( T_i \) satisfies the strengthened invariant. This follows because \( w_R \) satisfies \( l_z l_z + 1 \leq l_z \leq h_z(h_z - 1) \) and \( w_{R'} \) must satisfy \( l_z l_z + 1 \leq w_{R'} \leq h_z h_z - 4 \). Now the interval for \( w_{R'} \) contains the interval for \( w_R \) hence every \( R \) can be reconstructed as required. Again we refer to Lemma 4.3 for the proof that the new \( T_i \) can always be repositioned to include the search path for \( x \).

**Restructuring IV:** We have reached \( T_f \) and:

\[
l_z + 1 \leq w_f \leq h_z - 2
\]

Now an insertion yields:

\[
l_z + 2 \leq w_f \leq h_z - 1
\]

while a deletion yields:

\[
l_z \leq w_f \leq h_z - 3
\]

giving, in both cases, a \( Z \)-stratified tree once more. \( \square \)

The algorithm clearly can be carried out in time \( O(\log n) \), where \( n = \text{weight}(T) \), since a single pass from the root to the frontier is performed and for each node on the search path only a constant number of other nodes are visited.

We now prove that \( T_i \) can always be repositioned in layer \( i \) so as to include the search path for \( x \). However this requires \( l_z + 1 \leq w_i \leq h_z - 4 \) since repositioning may result in \( T_i \) obtaining extra nodes.

**Lemma 4.3** Let \( R \) be a window on layers \( i-1 \) and \( i \) in a \( Z \)-stratified tree \( T \) having either a simple or non-simple routing scheme. Then assuming \( T_{i-1} \) contains the search path for some key \( x \) and \( l_z + 1 \leq w_i \leq h_z - 4 \), then \( T_i \) can always be repositioned to include the search path for \( x \) such that \( w_i \) then satisfies \( l_z + 1 \leq w_i \leq h_z - 2 \).

**Proof:** If the search path for \( x \) is in \( T_i \) then the Lemma holds. Therefore assume it is not in \( T_i \), but is in some \( Z \)-tree \( D \) a left or right sibling of \( T_i \). Without loss of generality assume \( D \) is to the left of \( T_i \) and that they are adjacent. Let the external node of \( D \) which is on the search path of \( x \) be at position \( k \) relative to the leftmost external node of \( D \). See Figure 4.3.

If \( l_z + 1 \leq w_D \leq h_z - 2 \) then let \( D \) be the new \( T_i \). Otherwise there are three cases to consider.

**Case 1:** \( w_D = l_z \).

Interchanging \( T_i \) and \( D \) ensures that \( T_i \) contains the search path for \( x \) since \( w_i \geq l_z + 1 > k \).
Figure 4.3 Repositioning $T_i$

Case 2: $w_D = h_Z - 1$.
Move the rightmost external node of $D$ to $T_i$. If $k \leq h_Z - 2$ then let $D$ be the new $T_i$, otherwise the search path for $x$ is in $T_i$ (the additional node). In both cases $l_Z + 1 \leq w_i \leq h_Z - 2$ holds.

Case 3: $w_D = h_Z$.
Move the two rightmost external nodes of $D$ to $T_i$. If $k \leq h_Z - 2$ then let $D$ be the new $T_i$, otherwise the search path for $x$ goes through one of the newly added nodes of $T_i$. Again, in both cases, $l_Z + 1 \leq w_i \leq h_Z - 2$ holds.

This completes the case analysis and the Lemma. □

5. NON-SIMPLE ROUTING SCHEMES AND INTERNAL-SEARCH TREES

We only consider non-simple routing schemes for which an insertion or deletion affects at most one separating key in the tree and this critical separating key can be identified when traversing the search path. This assumption is used to modify UPDATE so that an update operation can be carried out in one pass.

When a separating key which might be affected by the update operation is identified (it need not be affected if the operation is redundant) the tree is restructured so that the critical separating key remains within the window as it moves down the search path. Observe that separating keys are never changed only the internal nodes of Z-trees on two layers are reassembled in a different way. The following theorem shows that all restructurings can be carried out so that one separating key can be moved down the search path together with the structure needed to carry out the update operation.

Theorem 5.1 Let $S(X, Z)$ be a set of Z-stratified trees with a non-simple
routing scheme. Then for any tree $T$ in $S(X, Z)$ and any key $x$, \textsc{Update}(x, T) yields a tree in $S(X, Z)$.

**Proof:** The proof is based on that of Theorem 4.2. But instead of just redistributing the external nodes within layer $i$, we ensure that a separating key for $x$ occurring in $T_{i-1}$ can be moved into $T_i$ so that it contains the search path for $x$ and satisfies $l_z < w_i < h_z$.

The method used is a natural generalization of the rotation used for deletion in Figure 1.1. A critical separating key equals the query key and is assumed to route the query key to the left of the critical key (in the $(<, <)$ and left-maximum routing scheme such is the case). The case of routing to the right is dealt with symmetrically. Moreover if a critical key is in $T_{i-1}$ then there must be a key at an external node of $T_{i-1}$ which is less than it, and one which is greater than it. (For if there is no key at an external node of $T_{i-1}$ which is less than the critical key then the critical key must separate $T_{i-1}$ from some other tree $T_{i-1}$ in layer $i-1$ and, hence, it cannot be in $T_{i-1}$.) This ensures that when the critical key $x$ is in $T_{i-1}$ it must separate two $Z$-trees $T_i$ and $V$ in layer $i$. This is illustrated in Figure 5.1.

![Figure 5.1 A Critical Separating Key](image)

It is assumed that the steps outlined in \textsc{Update} and Theorem 4.2 have been carried out up to layer $i-1$. Restructurings I and IV, the termination cases, are straightforward. Restructurings II and III are similar so we only treat III in more detail. We have the generic situation of Figure 4.1 once more. First carry out any restructuring of $R$ which is necessary following the proof of Theorem 4.2 ensuring that the search path for $x$ goes through $T_i$. The result of this restructuring is to give the generic situation of Figure 5.1 in which $l_z + 1 \leq w_i \leq h_z - 2$. To move the critical key $x$ we simply attach either external node $q$ to $T_i$ or $p$ to $V$. Since $x$ separates $T_i$ from $V$ and, in particular, $p$ from $q$, this movement will
cause the critical key \( x \) to be repositioned in either \( T_i \) or \( V \), respectively. The new separating key to be placed at \( r \) should separate either \( q \) from the remainder of \( V \) or the remainder of \( T_i \) from \( p \), respectively. These separating keys are to be found in \( V \) and \( T_i \), respectively. Since the separating keys can be repositioned appropriately, it only remains to demonstrate that the restructuring of \( T_i \) or \( V \) is always possible. We assume, without loss of generality, that \( V \) has weight no greater than any of its siblings to the right of \( T_i \). There are two cases to consider.

**Case 1:** \( w_V \geq l_2 + 1 \).
Move \( q \) from \( V \) to \( T_i \), hence \( l_2 + 1 \leq w_i \leq h_2 - 1 \).

**Case 2:** \( w_V = l_2 \).
Move \( p \) to \( V \), hence \( l_2 + 1 \leq w_V \leq h_2 - 1 \) and \( V \) is the new \( T_i \).

This completes the case analysis and the theorem. \( \Box \)

We are left with the case of stratified internal-search trees. As pointed out in the introduction internal-search trees are only insertion-simple. But deletion can be carried out as specified in Theorem 5.1, the only difference being that at the final layer the critical key is removed since it is the key. Thus we have:

**Corollary 5.2** Let \( S(X,Z) \) be a set of \( Z \)-stratified internal-search trees. Then for any tree \( T \) in \( S(X,Z) \) and any key \( x \) \( \text{UPDATE}(x,T) \) yields a tree in \( S(X,Z) \).

### 6. CONCLUDING REMARKS

In addition to the algorithm given for stratified trees, a purely top-down updating algorithm can also be developed for unbalanced trees with different routing schemes. The downward movement of a separating key can be achieved by rotations. A feature of our algorithm is that it deals with insertions and deletions so that it is oblivious to the kind of update required until the final layer is reached.

In Section 5 we have only considered non-simple routing schemes chosen from the four given in Section 2. In these cases the separating keys depend on at most two keys. In particular for the \((<,<)\) scheme these values depend on the minimal key in the subtree immediately to the right of a separating key and the maximal key in the subtree immediately to its left. Other routing schemes with the value of the separating key depending on more than these keys can also be considered. As long as the separating key depends on at most \( k_1 \) keys immediately to its left and at most \( k_2 \) keys immediately to its right, where \( k_1 \) and \( k_2 \) are constants, our algorithm can be used with the following rider: The window at the
final layer of the tree must be large enough to contain all the separating keys affected by the deletion or insertion of a key. Note that the height of a window is $O(\log(2(k_1 + k_2) - 1)) = O(1)$. In this case we always, have to move at most one separating key downwards. Only if the value of a separating key depends on keys which are not within a constant range to the left or right of the separating key might it be necessary to move more than one separating key down the search path; also the size of the window almost certainly increases in this case. It seems to us that even with such a routing scheme a purely top-down updating algorithm can be achieved albeit having little practical significance.

Acknowledgement

The authors wish to thank P. Widmayer for many helpful discussions and suggestions. Also we wish to thank the referee who was so dismayed at our preliminary version that drastic updating was insisted upon resulting in the present version. However, as always the authors are solely responsible for any remaining obfuscation.

REFERENCES


