Dynamic Trees with Almost-Optimal Access Cost

Mordecai Golin
Hong Kong University of Science and Technology
golin@cse.ust.hk

John Iacono[1]
Université libre de Bruxelles and New York University
johniacono@gmail.com

Stefan Langerman[2]
Université libre de Bruxelles
sl@slef.org

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo
imunro@uwaterloo.ca

Yakov Nekrich
Cheriton School of Computer Science, University of Waterloo
yakov.nekrich@googlemail.com

—— Abstract ——

An optimal binary search tree for an access sequence on elements is a static tree that minimizes the total search cost. Constructing perfectly optimal binary search trees is expensive so the most efficient algorithms construct *almost optimal* search trees. There exists a long literature of constructing almost optimal search trees *dynamically*, i.e., when the access pattern is not known in advance. All of these trees, e.g., splay trees and treaps, provide a *multiplicative* approximation to the optimal search cost.

In this paper we show how to maintain an almost optimal weighted binary search tree under access operations and insertions of new elements where the approximation is an *additive* constant. More technically, we maintain a tree in which the depth of the leaf holding an element e_i does not exceed $\min(\log(W/w_i), \log n) + O(1)$ where w_i is the number of times e_i was accessed and W is the total length of the access sequence.

Our techniques can also be used to encode a sequence of m symbols with a dynamic alphabetic code in $O(m)$ time so that the encoding length is bounded by $m(H+O(1))$, where H is the entropy of the sequence. This is the first efficient algorithm for adaptive alphabetic coding that runs in constant time per symbol.

---

1   Introduction

The dictionary problem is one of the most fundamental problems in computer science. It requires maintaining a set of elements in a data structure and being able to efficiently search for and find them when needed. In the comparison model, balanced binary search trees (BSTs) provide an optimal worst case solution for this problem. We consider leaf-oriented binary search trees, where all of the data is located in leaves and internal nodes store keys needed to guide the search to the leaves. For a set of n elements, it is well known that the perfectly balanced search tree has height $\lceil\log(n+1)\rceil$ and $\lceil\log(n+1)\rceil$ comparisons[3] are required to access an element, both in the worst and average cases. In many practical applications, some elements are known to be accessed more frequently than others; unbalancing and restructuring the tree so that more frequently accessed elements are stored higher up, can lead to better search times. Let d_i be the depth of the ith element e_i (stored at a leaf), w_i the frequency of accessing that element and $W = \sum_i w_i$ the total number of accesses. The total access cost is $\sum_i d_i w_i$; normalizing gives the *tree cost* which is $\frac{1}{W}\sum_i d_i w_i$. A tree that minimizes the tree cost minimizes the total access cost and is an *optimal* BST.

There is a long literature on constructing optimal BSTs, both exactly and approximately[4]. In the approximate case, there are algorithms that provide both multiplicative and additive errors. In the dynamic version of the problem the frequencies w_i are not known in advance but are calculated cumulatively as accesses are made. The problem then is to update the tree to be optimal for the current observed frequencies. Surprisingly, while there are many results on dynamic approximately optimal BSTs with constant multiplicative-error, prior to this paper there was not much known about constant additive-errors.

In this paper we revisit this problem and describe how to maintain dynamic approximately optimal BSTs with constant *additive*-error (this will be formally defined in the next subsection). The cost of re-building the tree after an access operation is bounded by $O(\log^{(f)} n)$ for any constant f, with the additive error growing linearly with f. As in standard BSTs, our technique permits insertions of new elements to the dictionary at any time.

A variant of our approach can also be used to obtain an almost-optimal adaptive alphabetic code with $O(1)$ encoding cost.

Previous and Related Work

There are a number of data structures that maintain (unweighted) dynamic trees with $O(\log n)$ depth, starting with the classic balanced trees of Adelson-Velski and Landis [1] and other handbook solutions [7, 16]. These data structures maintain all leaves at height $O(\log n)$ and thus support both searches and updates, i.e., insertions and deletions, in $O(\log n)$ time. The k-neighbor tree of Maurer et al. [22] achieves tree depth $(1+\delta)\log n$ and update cost $O((1/\delta)\log n)$ for any positive $\delta > 0$. Andersson [4] improved this result and showed how to maintain a tree of height $\log n + O(k)$ in $O(\log n)$ time per update. Even tighter bounds on constant and improved update times were described by Andersson and Lai [6] and Fagerberg [11]. We refer to [5] for an extensive survey of results in this area.

Gilbert and Moore [14] introduced an $O(n^3)$ time algorithm for constructing optimal BSTs. This was improved in 1971 by Knuth [21] to $O(n^2)$, which is still the best known method for solving the general case of the problem. Those two algorithms assume that frequencies for

---

[3] Throughout this paper log denotes the binary logarithm and $\log^{(f)}$ is the log function iterated f times.
[4] In this paper the term "optimal" refers to the optimality of the tree with respect to access frequencies. Splay trees, for example, can utilize other features of the access sequence in addition to frequencies.

both successful (elements in the tree) and *unsuccessful* (not in the tree) searches are given in advance and optimize accordingly. If the problem is restricted to successful searches then optimal BSTs can be constructed in in $O(n \log n)$ time using the Hu-Tucker algorithm and its variants. [13, 18]. Klawe and Mumey [19] show that, under some general conditions as to how the algorithms can operate, $\Omega(n \log n)$ is the best possible construction time, although, for certain restricted types of input, $O(n)$ can be achieved [17, 19].

Let $p_i = w_i/W$ be the empirical probability of element i in the access sequence. The Shannon Entropy of the sequence is $H = \sum_i p_i \log(1/p_i)$ which is known to be a lower bound on the cost of tree in which all data is in the leaves[5]. If a tree was guaranteed to have $d_i \leq c + \log(1/p_i)$ for all i then the total cost of all accesses would be at most $\sum_i w_i(c + \log(1/p_i)) = WH + cW$, i.e., within a constant additive error per access. In the static case multiple authors [2, 23, 28] have provided $O(n)$ time algorithms for constructing such trees with $c = 2$.

Now consider the dynamic case, in which trees are rebuilt based on cumulative frequencies viewed so far. *Splay trees* [25] and *Treaps,* [24] maintain *static optimality*, essentially keeping element e_i at depth $d_i = O(\log(1/p_i))$ for the current cumulative frequencies, in the amortized sense. This guarantees constant *multiplicative* errors in the dynamic case. There was no comparable result for maintaining almost optimal trees with additive errors, i.e., keeping element e_i at depth $d_i = \log(1/p_i) + O(1)$. The best technique would be to rebuild the tree from scratch at every step.

The dynamic (or adaptive) alphabetic coding problem is closely related to the dynamic alphabetic tree problem just described. The coding problem is to produce an encoding for a sequence of symbols $S[1] \ldots S[m]$ over an ordered alphabet $\{a_1, \ldots, a_n\}$ so that (1) no codeword is a prefix of any other and (2) the codeword for a_i is lexicographically smaller than the codeword for a_j iff $a_i < a_j$. In the adaptive scenario the input sequence is not known in advance; hence, we need to update the code every time a symbol is encoded. Dynamic Huffman [20, 26] and dynamic Shannon [12] algorithms solve this problem for the non-alphabetic case. The algorithm of Gagie [12] maintains a dynamic alphabetic code, such that the total encoding length is bounded by $(H + 2)m$ and runs in $O(m(H + 1))$ time.

Alphabetic coding is related but not equivalent to the alphabetic trees problem. Any alphabetic tree can be transformed into an alphabetic code in a straightforward way. Hence any dynamic alphabetic tree structure provides us with an alphabetic coding method. But this imposes a lower bound on the encoding time: if the code is represented by a tree, then we have to encode the symbols bit-by-bit. Hence any tree-based alphabetic coding method requires $\Omega(mH)$ time to encode the sequence. On the other hand, not every adaptive coding method can be transformed into a method for maintaining an alphabetic tree. For example, the method of Gagie [12] does not store the alphabetic tree and therefore can not be used to implement a dynamic dictionary.

Notation

The *weight w_ℓ* of a leaf node ℓ is the total number of times that an element stored in ℓ was accessed. We assume that every item is accessed at least once so $w_\ell \geq 1$ The weight of an internal node u is the total weight of all leaves in the subtree of u; the weight of a subtree is equal to the weight of its root. The total weight W of a tree T is the weight of its root node, i.e., $W = \sum_\ell w_\ell$ where the sum is taken over all leaves ℓ. This is also the total number of accesses made.

---

[5] When data can also be kept in internal nodes, as when three-way comparisons are allowed, the lower bound decreases to $H - \log H$ [3].

When necessary we further denote by $w_\ell^{(j)}$ the number of accesses to ℓ during the first j accesses. Thus $W^{(t)} = \sum_\ell w_\ell^{(t)} = t$.

Relation Between Static and Dynamic Optimal Trees

Consider an optimal static binary search tree for a sequence of W accesses to n elements. As previously noted, the average cost of such a tree is at most $H + 2$ where H is the entropy of the access sequence.

▶ **Lemma 1.** *Let $a_1, a_2, \ldots, a_W$ with $a_i \in \{1, 2, \ldots, n\}$ be a length W access sequence on the elements, i.e., element e_{a_t} is accessed at time t. Let H be the entropy corresponding to the full access sequence. Then*

$$\sum_{t=1}^{W} \log \frac{t}{\max\left(w_{a_t}^{(t-1)}, 1\right)} \leq W \cdot H + 2W.$$

The proof of this Lemma is straightforward and is therefore deferred to the full version of this paper [15].

Suppose that we could build a tree $T^{(t)}$ such that the depth of e_i *after* access t is $d_i^{(t)} \leq \log \frac{t}{w_i^{(t)}} + c$. The access of a_t at time t would be in the previous tree $T^{(t-1)}$ with cost $d_{a_t}^{(t-1)}$. The only exception to the above is if time t is the first access to e_{a_t}, so it was not already in $T^{(t-1)}$. In that case the access cost would be $d_{a_t}^{(t)}$, the depth of the location into which a_i^t would be inserted. Thus define $d_{a_t}^{(t-1)} = d_{a_t}^{(t)}$. Since $w_{a_t}^{(t-1)} = 0$ and $w_{a_t}^{(t)} = 1$, $d_{a_t}^{(t-1)} = d_{a_t}^{(t)} \leq \log t + c = \log \frac{t}{\max\left(w_{a_t}^{(t-1)}, 1\right)} + c$. The total cost of the accesses would then, from Lemma 1, be

$$\sum_i d_{a_t}^{(t-1)} \leq W \cdot H + (2 + c)W,$$

i.e, within a constant additive error of optimal per access, where optimal defined as the cost with the static optimal tree, is lower-bounded by $W \cdot H$.

Our approach to building almost optimal trees is therefore to maintain such trees $T^{(t)}$ over the access sequences.

Our Results

Let $f \geq 1$ be any fixed integer. In this paper we describe a dynamic tree structure that can be maintained under access operations and insertions. The depth of the leaf that holds e_i is bounded by $\min(\log(W/w_i), \log n) + O(f)$ where w_i is the number of times e_i was accessed so far and W is the total length of the access sequence. Hence we can access any element e_i using at most $\min(\log n, \log(W/w_i)) + O(1)$ comparisons. We can also insert new elements into the tree. When an element is accessed (resp. when a new element is inserted), only $O(\log^{(f)} n)$ worst-case time will be needed to update the tree; this update procedure does not require any comparisons. Thus our data structure enjoys the advantages of both the weighted alphabetic tree and the perfect binary tree. At the same time, the cost of maintaining the data structure is low.

This result is obtained by a combination of two ideas. First, our construction is based on approximate weights of elements instead of exact weights. Second, we maintain an unweighted binary tree T^s with leaves "representing" approximate weights. Our dynamic tree is a subtree of T^s. We define the approximate weights in Section 3 and describe the tree T^s in Section 4.

Next, we show how updates of our data structure can be implemented by leaf insertions in T^s in Section 5. We reduce the update cost and make all time bounds worst-case in Sections 6 and 7 respectively.

Our second result, concerns the adaptive alphabetic coding problem. Our method enables us to encode the sequence of m symbols with an adaptive alphabetic code in $O(m)$ time, constant time per symbol (in contrast to $O(m(H + 1))$ time in [12]). The length of encoding is bounded by $m(H + 1) + O(m)$ bits. Our solution is based on the same approach as our dynamic tree structure, but we employ a different method to maintain the underlying tree. This method is based on the list maintenance problem [8, 9, 27]. The full details of this result are omitted from this extended abstract but are presented in the full version of this paper [15], in section on alphabetic coding.

2   Preliminaries

An efficient solution for the unweighted search tree problem was presented by Maurer et al. [22]. Their data structure, called a *k-neighbor tree*, is a tree of height $(1 + \delta) \log n$, where δ denotes an arbitrarily small positive constant. A k-neighbor tree is a binary tree T such that (1) all leaves in T have the same depth and (2) if a node $u \in T$ has only one child, then u has at least one right neighbor (on the same level), and (3) if a node u has l right neighbors, then $\min(k, l)$ nearest right neighbors of u have two children.

Since this will be used later, we give a sketch of the insertion into such a tree below.

When a new leaf x is inserted into the tree, we find the node p such that the x must be inserted below p and call a recursive procedure INSERT(p, x). First, we make x a new child of p. If p has two children, the insertion procedure is completed. If p has three children, we look for a neighbor node q of p such that the distance between p and q is at most k and q has only one child. If q is found, we call the procedure MOVE(p, q). If q is not found, we create a new node p' that has one child; the only child of p' is the leftmost child of p. If p is not the root node, then we call the procedure INSERT$(parent(p), p')$; otherwise we create a new root node r_n and make both p and p' the children of r_n.

The arguments of the procedure MOVE(p, q) are two neighbor nodes, p and q, such that p has three children and q has only one child. All nodes u between p and q have two children. The procedure is applied to the children of all nodes u between p and q; every child node is shifted by one position to the right or to the left. At the end p, q, and all nodes u have two children. Thus MOVE(p, q) consists of d shifts, where d is the distance from p to q. Procedure MOVE(p, q) needs $O(k)$ time because every node shift takes $O(1)$ time. When a new leaf is inserted, we execute MOVE(p, q) only one time. Excluding the cost of MOVE(p, q), we spend $O(1)$ time on every tree level. Therefore a new leaf can be inserted into a tree in $O(\log n + k)$ time. A more detailed description of an insertion can be found in [4]. We can delete a leaf using a symmetric procedure.

The height of a k-neighbor tree with n leaves does not exceed $\left\lfloor \frac{\log n}{\log(2 - \frac{1}{k+1})} + 1 \right\rfloor$. Using the fact that for any $k \geq \log n$ the height of the tree is bounded by $\log n + O(1)$, Andersson [4] showed how, by using an appropriate value of k the tree height can be bounded by height $\log n + 2$ using only $O(\log n + k) = O(\log n)$ time per operation. It is this version of the data structure that we will use later.
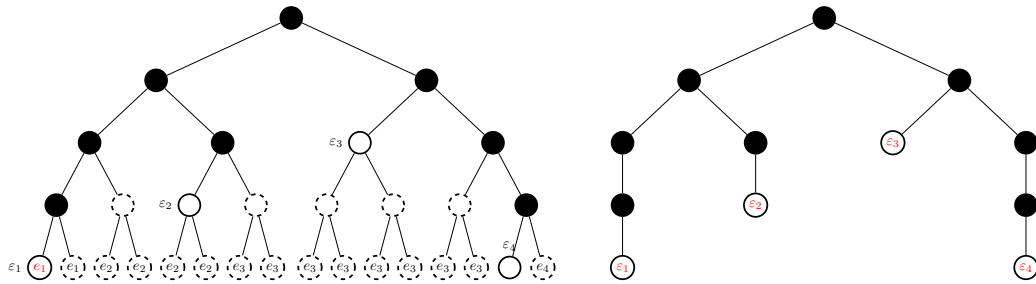
Figure 1 Left: Balanced tree of approximate weights $w_1' = 1$, $w_2' = 2$, $w_3' = 4$, and $w_4' = 1$. Elements $e_1, \ldots, e_4$ are stored in nodes $\varepsilon_1, \ldots, \varepsilon_4$ respectively. Pseudo-leaves are shown with dashed lines. Internal nodes of T^s that are not nodes of T are also drawn with dashed lines. Leaves of T are shown with solid lines and internal nodes of T are depicted by filled circles. Right: Almost-optimal tree corresponding to the tree on Fig. 1.

3 Approximate Weights

Consider an ordered weighted set of elements $E = \{\, e_1 < e_2 < \ldots < e_n \,\}$ let w_i denote the weight of e_i and $W = \sum_{j=1}^{n} w_j$. Define the approximate (or quantized) weight of an element e_i as $w_i' = \lceil w_i/\tau \rceil$ for $\tau = \frac{W}{n}$. Thus all approximate weights are integers between 1 and n. Note that $\sum \frac{w_i}{\tau} = \frac{n}{W} \sum_i w_i = n$. Hence $W' = \sum \left\lceil \frac{w_i}{\tau} \right\rceil \leq \sum_i \frac{w_i}{\tau} + n = 2n \leq 2W$.

▶ **Lemma 2.** *Suppose that the depth of a leaf ℓ_i in a tree T' does not exceed $\log(W'/w_i') + c$. Then the depth of ℓ_i in T' does not exceed $\min(\log(W/w_i), \log n) + c + 1$.*

Proof. Since $w_i' \geq 1$ for all i, $\log(W'/w_i') \leq \log W' \leq \log n + 1$. Furthermore $W' \cdot \tau \leq 2W$ and $w_i' \cdot \tau \geq w_i$. Hence $\frac{W'}{w_i'} = \frac{W' \cdot \tau}{w_i' \cdot \tau} \leq \frac{2W}{w_i}$ and $\log \frac{W'}{w_i'} \leq \log \frac{W}{w_i} + 1$.

In summary $\log \frac{W'}{w_i'} \leq \min(\log \frac{W}{w_i}, \log n) + 1$. ◀

The problem of maintaining an almost-optimal tree T' for quantized weights $\{\, w_1', \ldots, w_n' \,\}$ is thus equivalent to the problem of maintaining an almost-optimal tree for exact weights $\{\, w_1, \ldots, w_n \,\}$. The tree T' has another important property: the depths of all leaves in T' are bounded by $\lceil \log n \rceil + O(1)$.

4 Warm-Up: Almost-Optimal Static Trees

In this section we introduce our approach and basic notions that will be used in the following sections. By way of introduction we describe a method that produces an almost-optimal tree for a static set of elements with fixed weights.

We keep weights of elements as entries in an array B of size $m = 2W' \leq 2n$ so that there are two entries for each unit of weight. The first $2w_1'$ entries of B are assigned to e_1, the following $2w_2'$ entries are assigned to e_2, and so on. In general we assign entries $B[l_i]$, …, $B[r_i]$ to the element e_i where $l_i = (2\sum_{j=1}^{i-1} w_i') + 1$ and $r_i = 2\sum_{j=1}^{i} w_i'$. Let T^s denote a conceptual perfectly balanced tree on B. The i-th leaf of T^s corresponds to the entry $B[i]$ of B, every internal node has two children, and the height of T is $\log m = \log n + 1$[6]. The leaves of T^s will be called *pseudo-leaves*. Leaves corresponding to entries in $B[l_i..r_i]$ will be called *pseudo-leaves of the element e_i* (or pseudo-leaves associated to e_i).

---

[6] To avoid tedious details, we assume in this section that m and n are powers of 2.

▶ **Fact 3.** *Consider a node u of height $h \geq \lfloor \log r \rfloor$ for some $r \geq 1$. Suppose that r leftmost (rightmost) pseudo-leaves in the subtree of u are pseudo-leaves of e_i. Then there is at least one node v of height $\lfloor \log r \rfloor$ such that all pseudo-leaves in the subtree of v are pseudo-leaves of e_i.*

Consequentially, if $2x$ entries are assigned to some element e_i, then there is at least one node v of height $\lfloor \log x \rfloor$, such that all pseudo-leaves in the subtree of v are assigned to e_i.

We define an almost-optimal tree T as a subtree of T^s. Let ε_i denote an arbitrary node of height $\lfloor \log(w_i') \rfloor$ such that all leaves in the subtree rooted at ε_i are i-nodes. Since we assigned $2w_i'$ pseudoleaves to e_i, such a node ε_i always exists. All pseudoleaves below ε_i correspond to some array entries in $B[l_i..r_i]$. The tree T is a subtree of T^s pruned at nodes ε_i. That is, the nodes ε_i are the leaves of T and all proper ancestors of all ε_i are internal nodes of T. We keep keys in the internal nodes of T that can be used for routing.

The depth of the leaf ε_i does not exceed $\log \frac{W'}{w_i'}$ by more than a constant: every leaf of T^s has depth at most $\log W' + 1$. The depth of ε_i is then at most

$$\log(W') + 1 - (\log(w_i') + 1) = \log \frac{W'}{w_i'} + 2 \leq \log \frac{W}{w_i} + 3.$$

Hence each ε_i has an almost-optimal depth in T. In addition T^s is a perfectly balanced tree with $2n$ nodes and the depth of any node in T^s does not exceed $\log n + 1$. Summing up, the depth of any leaf ε_i that holds the element e_i does not exceed $\min(\log(W/w_i), \log n) + 3$.

An example tree T^s and the corresponding almost-optimal tree T are shown on Fig. 1. An interesting property of our method is that the tree T is not necessarily a full tree: it is possible that some internal nodes have only one child. In the following sections we will show how the tree T^s can be dynamized.

5   Almost-Optimal Dynamic Trees

Our dynamic data structure maintains a balanced tree T^s on a dynamic set B of pseudo-leaves.

This first version of the algorithm will work in phases. A phase will end when the total weight W is increased by a factor of 2 or when the total number of elements is increased by a factor of 2.

Unlike in the previous section, these pseudo leaves are not kept in an array. Instead, T^s is maintained as a k-neighbor tree data structure with $k = \log n$ [22] as described in Section 2. This method guarantees that all leaves of T^s have the same depth and , since the total number of pseudoleaves can at most double within a phase, the height of the tree is bounded by $\log(4W') + 1 \leq \log n + 4$. An update of T^s takes $O(\log^2 n)$ time.

Each phase starts with a correct T^s that had just been built from scratch using the approach of Section 4. Set $\bar{\tau} = \tau = \frac{W}{n}$. This value stays constant within the phase.

During a phase, for every element e_i we keep track of its weight w_i and its approximate weight $w_i' = \lceil w_i/\bar{\tau} \rceil$. Note that this implies that during a phase w_i' can be increased (incremented by 1 at a step) but not decreased.

When w_i' is incremented by 1, the tree T^s is updated: we identify the rightmost pseudo-leaf ℓ_i associated to e_i and insert two new pseudo-leaves, ℓ_n and ℓ_{n+1}, immediately after ℓ_i. When a new element e_f is inserted into a tree, we insert two new pseudo-leaves, ℓ_f and ℓ_{f+1}, into T^s. The leaf ℓ_f is inserted after the leaf ℓ_p, where e_p is the largest element satisfying $e_p < e_f$ and ℓ_p is the rightmost leaf associated to e_p. Every insertion of a pseudo-leaf results in a modification of the tree T^s.
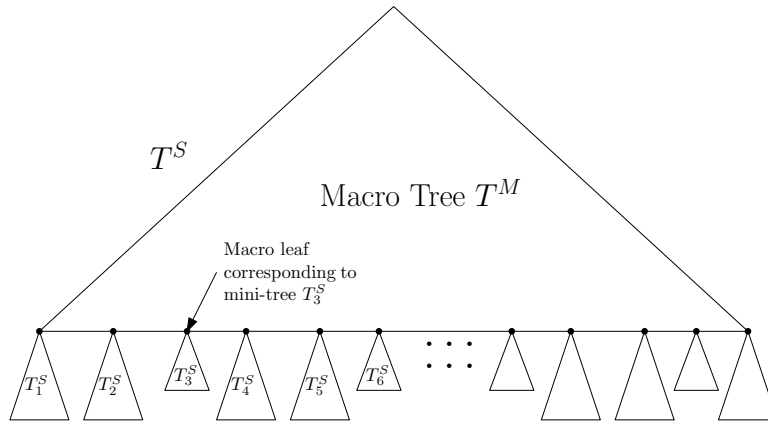
Figure 2 The partition of T^S into macro tree T^M and mini-trees T_j^s. The leaves of T^m are the roots of the T_j^s. All the T_j^s have between $\log^2 n$ and $2\log^2 n$ pseudoleaves. T^M and all of the T_j^s are maintained as dynamic almost-optimal trees for their sets of leaves using the technique of Section 5.

We maintain the almost-optimal tree T as a subset of T^s using the approach of Section 4. An internal node ε_i is an internal node of T^s of height $\lfloor \log(w_i') \rfloor$ such that all leaves in its subtree are associated to an element e_i. Using the same calculations as in Section 4 the depth of ε_i is then at most $\log \frac{W}{w_i'} + 4$ (and not 3 because the calculation is using $\bar{\tau}$ and not τ.)

After an update of T^s, some nodes of T^s (and, hence, some nodes of T) can be moved. If all leaves of a moved internal node u are associated to e_j, we also update the internal node ε_j, if necessary. Suppose that a node u was moved by one position to the left and the node u' to the right of u was also moved by one position to the left. If all leaf descendants of u are associated with an element e_i and all leaf descendants of u' are associated with some $e_j \neq e_i$, then we may have to update ε_i. If ε_i is an ancestor of u, we find the immediate left neighbor ε_i' of ε_i. Since there are $2w_i'$ leaves associated to e_i and the height of ε_i is $\log(w_i')$, all leaf descendants of ε_i' are associated to e_i. Hence we can set $\varepsilon_i := \varepsilon_i'$. We can find the ε_i and ε_i' for every moved node u in $O(\log n)$ time. At most $O(\log n)$ nodes of T^s are moved during every update [4]; hence, the total update cost is $O(\log^2 n)$.

When the total weight W is increased by a factor 2 or when the total number of elements is increased by a factor 2, we update the value of $\tau = \frac{W}{n}$, compute the new values w_i' and as noted, re-build the tree from scratch. The amortized cost of rebuilding T^s from scratch is $O(1)$ per step since the balanced tree can be built in linear time. When we re-build the tree T^s, we use the new value of $k = \log n$.

▶ **Lemma 4.** *We can implement a binary search tree so that access to an element and an insertion of a new element are supported in $O(\log^2 n)$ amortized time. If an element e_i was accessed w_i times over a sequence of W operations, then the depth of the leaf holding e_i does not exceed $\min(\log(W/w_i), \log n) + O(1)$.*

6 Faster Updates

We can reduce the update time by grouping pseudo-leaves in the tree T^s. All pseudo-leaves are divided into $\Theta(n/\log^2 n)$ groups so that each group contains at least $\log^2 n$ and at most $2\log^2 n$ pseudo-leaves.

The tree T^s is divided into two components: a macro-tree T^M with $O(n/\log^2 n)$ leaves and $O(n/\log^2 n)$ mini-trees T_j^s. See Fig. 2. Mini-trees correspond to groups of pseudo-leaves:

all pseudo-leaves in the group G_j are stored in a mini-tree T_j^{s}. The j'th leaf of macro-tree T^M is the root of mini-tree T_j^{s}. As before, the almost-optimal tree T is a subtree of T^{s}. An element e_i is assigned to a node ε_i of T, such that the height of ε_i in T^{s} is $\log(w_i')$ (up to an additive constant error) and all leaves in the subtree of ε_i are associated to e_i. T is the subtree of T^{s} induced by nodes ε_i and their ancestors. The division of a tree into macro-trees and mini-trees is a standard data structuring technique; see e.g., [6].

We now find the node ε_i for any element e_i either in a mini-tree or in the macro-tree. Recall that there are $2w_i'$ pseudo-leaves associated to e_i. Let $g = 2\log^2 n$.

First suppose that $w_i' \leq g$; then the pseudo-leaves of e_i are distributed among $O(1)$ subtrees. If all pseudo-leaves are in one subtree T_j^{s}, then T_j^{s} has at least one node u of height $\lfloor \log(w_i') \rfloor$ such that all leaves below u are associated to e_i. If pseudo-leaves of e_i are in two subtrees, T_j^{s} and T_{j+1}^{s}, then either w_i' rightmost pseudo-leaves in T_j^{s} are associated to e_i or w_i' leftmost leaves in T_{j+1}^{s} are associated to e_i. Hence either T_j^{s} or T_{j+1}^{s} contains a node that can be chosen as ε_i. If pseudo-leaves of e_i are distributed among more than two mini-trees, then there is at least one mini-tree T_j^{s} with all pseudo-leaves associated to e_i. In the latter case we can choose the root of T_j^{s} as ε_i.

Now suppose that $kg \leq w_i' < (k+1)g$ for some $k \geq 1$. Then there are at least $2k-1$ mini-trees with all pseudo-leaves associated to e_i. The roots of these mini-trees are macro-leaves ℓ_j, ..., ℓ_{j+2k}. There is at least one node u of height $\lfloor \log k \rfloor$ in the macro-tree, such that all macro-leaves below u are among ℓ_j, ..., ℓ_{j+2k}.

Using Lemma 4, we maintain the mini-tree T_j^{s} for every group G_j. Since each mini-tree has $O(\log^2 n)$ leaves, updates on a mini-tree take $O((\log \log n)^2)$ time. The macro-tree is updated only when a new mini-tree is inserted or a mini-tree is deleted. Hence the cost of updating the macro-tree can be distributed among $O(\log^2 n)$ insertions of pseudo-leaves. Suppose that a new pseudo-leaf corresponding to an element e_i is inserted. As in Section 5 we find the rightmost pseudo-leaf ℓ_i' corresponding to an element e_i. The new pseudo-leaf ℓ_i is inserted into the same mini-tree as ℓ_i' immediately to the right of ℓ_i'. Since every mini-tree has $O(\log^2 n)$ pseudo-leaves, we can insert a new pseudo-leaf in $O((\log \log n)^2)$ time. If the number of pseudo-leaves in T_j^{s} is equal to $2\log^2 n$, we split the mini-tree T_j^{s} into two mini-trees of size $\log^2 n$; then we insert a new macro-leaf into T^{M}. The cost of an insertion into T^{M} is $O(\log^2 n)$. We can also split a mini-tree into two mini-trees in $O(\log^2 n)$ time. Hence the amortized cost of maintaining the macro-tree is $O(1)$.

The total height of a tree does not exceed the height of the macro-tree plus the maximum height of a mini-tree. Since the number of mini-trees is bounded by $\frac{2W'}{(\log^2 n)/2}$, the height of the macro-tree does not exceed $\log(W') - 2\log \log n + 3$. The height of a mini-tree is bounded by $2\log \log n + 1 + O(1)$ because it contains at most $2\log^2 n$ pseudo-leaves. Hence the total height of our tree does not exceed $\log(W') + 4$. We already showed that the height of a sub-tree rooted at the node ε_i is $\lfloor \log(w_i') \rfloor$; hence the depth of ε_i in T is at most $\log(W'/w_i') + O(1)$.

▶ **Lemma 5.** *We can implement a binary search tree so that access to an element and an insertion of a new element are supported in $O((\log \log n)^2)$ amortized time. If an element e_i was accessed w_i times over a sequence of W operations, then the depth of the leaf holding e_i does not exceed $\min(\log(W/w_i), \log n) + O(1)$.*

The result of Lemma 5 can be further improved by bootstrapping. For any integer $f \geq 1$ the following statement can be proved.

▶ **Lemma 6.** *Suppose there exists a binary search tree T^f, such that (1) the depth of a leaf holding an element e_i in T^f does not exceed $\min(\log(W/w_i), \log n) + O(1) + O(f)$ (2) the amortized cost of updating T^f after an element access or an insertion is $O((\log^{(f)} n)^2)$.*

Then there is a binary search tree T^{f+1}, such that (1) the depth of a leaf holding an element e_i in T^{f+1} does not exceed $\min(\log(W/w_i), \log n) + O(1) + O(f+1)$ (2) the amortized cost of updating T^{f+1} after an element access or an insertion is $O((\log^{(f+1)} n)^2)$.

Proof. We divide the tree T^{s} into the macro-tree and mini-trees in the same way as in the proof of Lemma 5. Every mini-tree is implemented using the tree T^f. Hence each mini-tree can be updated in $O((\log^{(f)}(\log n))^2) = O((\log^{(f+1)} n)^2)$ time. The amortized cost of maintaining the macro-tree is $O(1)$. Hence the total amortized cost of updates is $O((\log^{(f+1)} n)^2)$.

Suppose that ε_i is stored in the macro-tree. The depth of a node ε_i in the macro-tree is bounded by $\log(\min(W'/w_i', n)) + O(1)$. Now suppose that ε_i is stored in some mini-tree. The depth of ε_i in the mini-tree is bounded by $\log(\min(W_g'/w_i', n_i)) + O(f) + O(1)$, where W_g' is the total sum of all quantized weights in the mini-tree and n_g is the total number of elements in the subtree. By the same argument as in Lemma 5, the depth of ε_i in T is bounded by $\log(\min(W'/w_i', n)) + O(f+1) + O(1)$. ◀

Our main result is obtained when we apply Lemma 6 $f+1$ times for a parameter $f \geq 0$.

▶ **Theorem 7.** *For any $f \geq 1$ there exists a binary search tree T^f, such that the depth of a leaf holding an element e_i in T^f does not exceed $\min(\log(W/w_i), \log n) + O(f)$ and the amortized cost of updating T^f after an element access or an insertion is $O(\log^{(f)} n + f)$.*

We remark that when we insert a new element e_i, we need to update the search path for one leaf. This may incur an additional cost of $\log n + O(1)$ operations.

Our data structure can also support two symmetric operations. We can decrement the weight of an element and delete an element of weight 1. These operations can be implemented in the same way as incrementing the weight of an element and an insertion of a new element.

7   Worst-Case Updates

Our construction can be modified to support updates with worst-case time guarantees. We start by showing how the data structure from Section 5 can be changed. We run several processes in the background; these processes adapt the tree structure to the changing value of the parameter τ and maintain the correct number of pseudo-leaves for each element e_i. The value of τ is changed every time the total weight W for the number of elements is changed by a constant factor (described below). Two background processes guarantee that the value of τ used in T^{s} is within a constant factor of its current value. Moreover pseudo-leaves are stored in a k-neighbor tree data structure, but the parameter $k = \Theta(\log n)$ must be changed when the number of elements is increased or decreased by too much. We run another process that modifies the tree when the parameter k needs to be changed.

Let W_0 and n_0 denote the total weight and the number of elements at some time t_0. Let $\tau_0 = W_0/n_0$ and let the *delayed weight* of an element e_i be defined as $\overline{w}_i = \lceil w_i/\tau_0 \rceil$. We maintain the invariant that w_i' differs from $\overline{w}_i$ by at most a constant factor. In the worst-case construction delayed weights $\overline{w}_i$ are used instead of w_i', i.e., an element e_i is assigned $\overline{w}_i$ pseudo-leaves. Our re-building processes guarantee that $W_0 \leq W \leq (4/3)W_0$ and $n_0 \leq n \leq (4/3)n_0$. Therefore $\tau = (W/n) \leq (4/3)\tau_0$ and $\tau \geq (3/4)\tau_0$. For any element e_i, $\overline{w}_i = \frac{w_i}{\tau_0} \leq (4/3)w_i'$ and $\overline{w}_i \geq (3/4)w_i'$. Thus we have $\frac{\overline{W}}{\overline{w}_i} \leq (16/9)\frac{W'}{w_i'}$ where $\overline{W} = \sum_i \overline{w}_i$ and $\log(\overline{W}/\overline{w}_i) < \log(W'/w_i') + 1 \leq \log(W/w_i) + 2$.

We move among three re-building processes. Each process is executed in the background during at most $n/18$ insertions or accesses. The first process updates the value of W_0. If
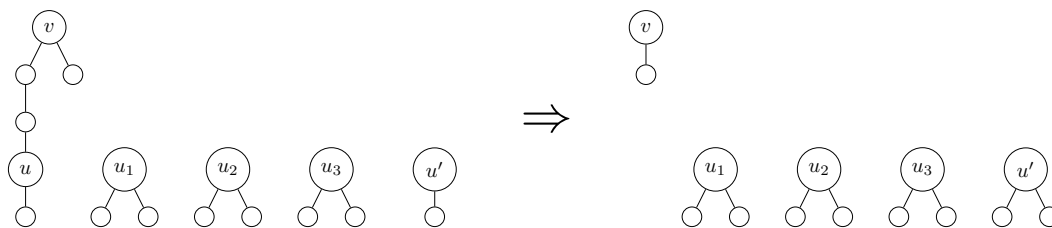
Figure 3 Example of procedure $\textsc{Consolidate}(u, u')$. Left: nodes u and u' have one child. Right: node u and its ancestors, up to a node v that has two children, are removed. Children of u_1, u_2, u_3 are shifted one position to the right. Only relevant nodes and their children are shown.

$W \geq (7/6)W_0$, we set $W_1 = W$, $n_1 = n$, and compute $\tau_1 = W_1/n_1$. For every e_i, we compute the new value of $\overline{w}_i = w_i/\tau_1$ and update the tree T^{s} by removing some pseudo-leaves if necessary. When the number of pseudo-leaves for all elements is adjusted in this way, we set $W_0 = W_1$ and $n_0 = n_1$. The second process updates the value of n_0. If $n \geq (7/6)n_0$, we also compute the new $\tau_1 = W_1/n_1$ for $W_1 = W$ and $n_1 = n$. Then for every element e_i we set $\overline{w}_i = w_i/\tau_1$ and update the tree T^{s}. The tree always contains $O(n)$ leaves. Every time when we access an element or insert a new element, our background process inserts or removes $O(1)$ pseudo-leaves. We can choose the constant in such a way that adjusting the value of τ_0 is distributed among $n/18$ update or access operations. Suppose that $W \geq (7/6)W_0$ or $n \geq (7/6)n_0$; the value of τ_0 will be adjusted after at most $n/6$ operations. Hence $W \leq (4/3)W_0$ and $n \leq (4/3)n_0$ at any time.

The third background process updates the parameter k in the k-neighbor tree. We set $k_0 = 2(\log(W_0) + 1)$ and maintain a k_0-neighbor tree on pseudo-leaves. When the number of leaves in T^{s} is increased by factor 2, we start the process of adjusting k. Internal nodes on every level of the tree are divided into pieces, so that every piece consists of $k_0 + 2$ consecutive nodes. We process pieces on the same level in the left-to-right order. Since T^{s} is already a k_0-neighbor tree, the distance between any two 1-nodes (a 1-node is a node with one child) is at least $k_0 + 1$. Hence each piece contains at most two 1-nodes. If there are two 1-nodes in the same piece P, then they are the leftmost and the rightmost nodes in P. In this case, we execute the procedure $\textsc{Consolidate}(u, u')$, where u and u' are the 1-nodes in P. This procedure, that will be described below, removes the node u and adds one additional child to u'. If P contains one 1-node, then we examine the preceding piece P'. If P' also contains a 1-node and the distance between the 1-nodes in P and P' is equal to $k_0 + 2$, we start the procedure $\textsc{Consolidate}(u', u)$, where u is the 1-node in P and u' is the 1-node in the slide that precedes P. After all pieces on a tree level are processed, every piece contains at most one 1-node and the distance between 1-nodes is at least $k + 2$. We will show below that $\textsc{Consolidate}$ requires $O(k \log n)$ move operations and can be executed in $O(k \log^2 n) = O(\log^3 n)$ time. Hence the third background process needs $O((n/k) \log^3 n) = O(n \log^2 n)$ time. Since an update takes $O(\log^2 n)$ time, we can distribute the third process among $n/18$ tree updates or accesses.

It remains to describe the procedure $\textsc{Consolidate}(u, u')$. $\textsc{Consolidate}(u, u')$ considers the children of nodes u, u', and the children of all nodes between u and u'. Every such node is moved by one position to the left. As a result, the node has no children and all other considered nodes have two children. Next, let v be the lowest ancestor of u that has two children. The node u and all its ancestors that are below v have no leaf descendants now. We remove the node u and all nodes between v and u. Now the node v is a 1-node. If v is the root node, then we remove v. Otherwise, we check whether v has a neighbor v', such

that the distance between v and v' does not exceed $k_0 + 1$ and v' is a 1-node. If v' exists, we recursively call the procedure CONSOLIDATE(v, v') (respectively (CONSOLIDATE(v', v)). There is at most one recursive call of our procedure per tree level. Our procedure shifts $O(k_0)$ nodes by one position to the right and recursively calls itself on some higher tree level; every time when some node in T^{s} is shifted, we may have to move some leaf ε_i of T. Hence the total time of CONSOLIDATE(u, u') is $O(k_0 \log^2 n) = O(\log^3 n)$.

▶ **Lemma 8.** *We can implement a binary search tree so that access to an element and an insertion of a new element are supported in $O(\log^2 n)$ time. If an element e_i was accessed w_i times over a sequence of W operations, then the depth of the leaf holding e_i does not exceed* $\min(\log(W/w_i), \log n) + O(1)$.

7.1   Fast Updates

Now we show how the data structure from Section 6 can be changed to support updates in worst-case time. As in Section 6 the tree T^{s} is divided into the macro-tree and mini-trees. Each mini-tree contains $O(\log^3 n)$ pseudo-leaves.

A new pseudo-leaf is inserted into a mini-tree; the cost of an insertion is $O((\log \log n)^2)$ time by Lemma 8. We run an additional background process that maintains the sizes of mini-trees. During each iteration we identify the largest mini-tree T_l among all subtrees of size at least $(7/4) \log^3 n$. We split T_l into two mini-trees of almost-equal size. We also identify the smallest mini-tree T_k of size at most $(3/4) \log^3 n$; we merge T_k with one of its direct neighbors (i.e., with the mini-tree immediately to the left or immediately to the right of T_k). If the resulting mini-tree is larger than , then we split it into two almost-equal parts. We show in the full version [15] how a mini-tree can be split into two almost-equal parts or merged with another mini-tree in less than $O(\log^2 n)$ time. When we split or merge two mini-trees, we also have to perform $O(1)$ updates on the macro-tree. The cost of updates is $O(\log^2 n)$, hence each iteration takes $O(\log^2 n (\log \log n)^2)$ time. By Theorem 5 from [10], we can organize our background process so that each mini-tree has no more than $2 \log^3 n$ and no less than $\log^3 n/2$ pseudo-leaves.

▶ **Lemma 9.** *We can implement a binary search tree so that access to an element and an insertion of a new element are supported in $O((\log \log n)^2)$ amortized time. If an element e_i was accessed w_i times over a sequence of W operations, then the depth of the leaf holding e_i does not exceed* $\min(\log(W/w_i), \log n) + O(1)$.

We can recursively apply Lemma 9 in the same way as described in Section 6. To obtain the main result of this paper with worst-case guarantees, we apply Lemma 9 $k + 1$ times for a parameter $k > 1$.

▶ **Theorem 10.** *For any $k \geq 1$ there exists a binary search tree T^k, such that the depth of a leaf holding an element e_i in T^k does not exceed $\min(\log(W/w_i), \log n) + O(k)$ and the cost of updating T^k after an element access or an insertion is $O(\log^{(k)} n + k)$.*

────── **References** ──────

1   Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of informa-tion. *Soviet Mathematics – Doklady*, 3:1259–1262, 1962.

2   Rudolf Ahlswede and Ingo Wegner. *Search Problems.* John Wiley and Sons, Chichester, 1987.

3   Brian Allen. On the costs of optimal and near-optimal binary search trees. *Acta Informatica*, 18:255–263, 1982.

4   Arne Andersson. Optimal bounds on the dictionary problem. In *Proc. International Symposium on Optimal Algorithms*, pages 106–114, 1989.

5   Arne Andersson, Rolf Fagerberg, and Kim S. Larsen. Balanced binary search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2004.

6   Arne Andersson and Tony W. Lai. Comparison-efficient and write-optimal searching and sorting. In *(Proc. 2nd International Symposium on Algorithms (ISA '91)*, pages 273–282, 1991.

7   Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972.

8   Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th Annual European Symposium on Algorithms (ESA 2002)*, pages 152–164, 2002.

9   Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 1503–1522, 2017.

10   Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC 1987)*, pages 365–372. ACM, 1987.

11   Rolf Fagerberg. Binary search trees: How low can you go? In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT '96)*, pages 428–439, 1996.

12   Travis Gagie. Dynamic shannon coding. In *Proc. 12th Annual European Symposium on Algorithms (ESA 2004)*, pages 359–370, 2004.

13   Adriano M. Garsia and Michelle L. Wachs. A New Algorithm for Minimum Cost Binary Trees. *SIAM Journal on Computing*, 6(4):622–642, 1977.

14   E.N. Gilbert and E.F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

15   Mordecai Golin, John Iacono, Stefan Langerman, J. Ian Munro, and Yakov Nekrich. Dynamic trees with almost-optimal access cost. `arXiv:1806.10498`.

16   Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21, 1978.

17   T. C. Hu, Lawrence L Larmore, and J David Morgenthaler. Optimal Integer Alphabetic Trees in Linear Time. In *13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669, pages 226–237, 2005.

18   T. C. Hu and A. C. Tucker. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

19   Maria Klawe and Brendan Mumey. Upper and Lower Bounds on Constructing Alphabetic Binary Trees. *SIAM Journal on Discrete Mathematics*, 8(4):638–651, 1995.

20   D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

21   Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

22   Hermann A. Maurer, Thomas Ottmann, and Hans-Werner Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, 1976.

23   Kurt Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6(2):235–239, 1977.

24   Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

25   Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

26 J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 1987(4):825–845, 1987.

27 Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Inf. Comput.*, 97(2):150–204, 1992.

28 R.W. Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, may 1991.