Dynamic Ordered Sets with Exponential Search Trees*

Arne Andersson
Computing Science Department
Information Technology, Uppsala University
Box 311, SE - 751 05 Uppsala, Sweden
arnea@csd.uu.se http://www.csd.uu.se/∼arnea

Mikkel Thorup
AT&T Labs–Research
Shannon Laboratory
180 Park Avenue, Florham Park
NJ 07932, USA
mthorup@research.att.com

Abstract

We introduce exponential search trees as a novel technique for converting static polynomial space search structures for ordered sets into fully-dynamic linear space data structures.

This leads to an *optimal* bound of $O(\sqrt{\log n/\log\log n})$ for searching and updating a dynamic set X of n integer keys in linear space. Searching X for an integer y means finding the maximum key in X which is smaller than or equal to y. This problem is equivalent to the standard text book problem of maintaining an ordered set.

The best previous deterministic linear space bound was $O(\log n/\log\log n)$ due to Fredman and Willard from STOC 1990. No better deterministic search bound was known using polynomial space.

We also get the following worst-case linear space trade-offs between the number n, the word length W, and the maximal key $U < 2^W$: $O(\min\{\log\log n + \log n/\log W,\ \log\log n \cdot \frac{\log\log U}{\log\log\log U}\})$. These trade-offs are, however, not likely to be optimal.

Our results are generalized to finger searching and string searching, providing optimal results for both in terms of n.

*This paper combines results presented by the authors at the 37th FOCS 1996 [2], the 32nd STOC 2000 [5], and the 12th SODA 2001 [6]

1 Introduction

1.1 The Textbook Problem

Maintaining a dynamic ordered set is a fundamental textbook problem (see, e.g., [13, Part III]). Starting from an empty set X, the basic operations are:

Insert (X, x) Add x to X where x is a pointer to a key.

Delete (X, x) Remove x from X, here x is a pointer to a key in X.

Search (X, y) Returns a pointer to a key in X with the same value as y, or return a null pointer if there is no such key in X.

Predecessor/Successor (X, x) Given that x points at a key in X, return a pointer to the next smaller/larger key in X (or a null pointer if no such key exists).

Minimum/Maximum (X) Return a pointer to the smallest/largest key in X (or a null pointer if X is empty).

For keys that can only be accessed via comparisons, all of the above operations can be supported in $O(\log n)$ time[1], which is best possible.

However, on computers, integers and floating point numbers are the most common ordered data types. For such data types, represented as lexicographically ordered, we can apply classical non-comparison based techniques such as radix sort and hashing. Historically, radix sort dates back at least to 1929 [12] and hashing dates back at least to 1956 [15], whereas the focus on general comparison based methods only date back to 1959 [16].

In this paper, we consider the above basic data types of integers and floating point numbers. *Our main result is that we can support all the above operations in* $O(\sqrt{\log n / \log\log n})$ *worst-case time, and this common bound is best possible.* We achieve this by introducing a new kind of search trees, called *exponential search trees*, illustrated in Figure 1.1.

The lower bound follows from a result of Beame and Fich [7]. It shows that even if we just want to support insert and predecessor operations in polynomial space, one of these two operations have a worst-case bound of $\Omega(\sqrt{\log n / \log\log n})$, matching our common upper bound. We note that one can find better bounds and trade-offs for some of the individual operations. Indeed, we will support min, max, predecessor, successor, and delete operations in constant time, and only do insert and search in $\Theta(\sqrt{\log n / \log\log n})$ time.

It is also worth noticing that if we just want to consider an incremental dictionary supporting insert and search operations, then our $O(\sqrt{\log n / \log\log n})$ search time is the best known with $n^{o(1)}$ insert time.

[1] We use the convention that logarithms are base 2 unless otherwise stated. Also, n is the number of stored keys

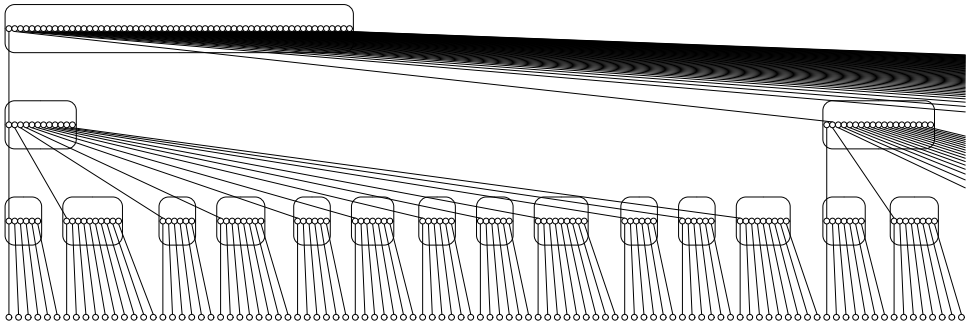

Figure 1.1: An exponential search tree. Shallow tree with degrees increasing doubly-exponentially towards the root.

1.2 Extending the search operation

For an ordered set X, it is common to consider an extended version of searching:

Search (X, y) Returns a pointer to the largest key in X which is smaller than or equal to y, or null if y is smaller than any key in X.

Thus, if the key is not there, we do not just return a null pointer. It is for this extended search that we provide our $O(\sqrt{\log n / \log\log n})$ upper bound. It is also for this extended search operation that Beame and Fich [7] proved their $\Omega(\sqrt{\log n / \log\log n})$ lower bound. Their lower bound holds even if the set X is static with an arbitrary representation in polynomial space. To see that this implies a lower bound for insert or predecessor, we solve the static predecessor problem as follows. First we insert all the keys of X to create a representation of X. To search for y in X, we insert y and ask for its predecessor. The lower bound of Beame and Fich implies that the insert and predecessor operation together takes $\Omega(\sqrt{\log n / \log\log n})$ time in the worst-case, hence that at least one of the operations has a worst-case lower bound of $\Omega(\sqrt{\log n / \log\log n})$.

In the rest of this paper, *search* refers to the extended version whereas the primitive version, returning null if the key is not there, is referred to as a *look-up*.

We will always maintain a sorted doubly-linked list with the stored keys and a distinguished head and tail. With this list we support successor, predecessor, min, and max operations in constant time. Then insertion subsumes a search identifying the key after which the new key is to be inserted. Similarly, if we want to delete a key by value, rather than by a pointer to a key, a search, or look-up, identifies the key to be deleted, if any.

1.3 Model of computation

Our algorithms run on a RAM that reflects what we can program in standard imperative programming languages such as C [23]. The memory is divided into addressable words of length W. Addresses are themselves contained in words, so $W \geq \log n$. Moreover, we have a constant number of registers, each with capacity for one word. The basic instructions are:

conditional jumps, direct and indirect addressing for loading and storing words in registers. Moreover we have some computational instructions, such as comparisons, addition, and multiplication, for manipulating words in registers. We are only considering instructions available in C [23], and refer to these as *standard* instructions. For contrast, on the low level, different processors have quite different computational instructions. Our RAM algorithms can be implemented in C-code which in turns can be compiled to run on all these processors.

The space complexity of our RAM algorithms is the maximal memory address used, and the time complexity is the number of instructions performed. All keys are assumed to be integers represented as binary strings, each fitting in one word (the extension to multi-word keys is treated in section 7). One important feature of the RAM is that it can use keys to compute addresses, as opposed to comparison-based models of computation. This feature is commonly used in practice, for example in bucket or radix sort.

The restriction to integers is not as severe as it may seem. Floating point numbers, for example, are ordered correctly, simply by perceiving their bit-string representation as representing an integer. Another example of the power of integer ordering is fractions of two one-word integers. Here we get the right ordering if we carry out the division with floating point numbers with $2W$ bits of precision, and then just perceive the result as an integer. These examples illustrate how integer ordering can capture many seemingly different orderings.

1.4 History

At STOC'90, Fredman and Willard [18] surpassed the comparison-based lower bounds for integer sorting and searching. Their key result was an $O(\log n/\log\log n)$ amortized bound for deterministic dynamic searching in linear space. They also showed that the $O(\log n/\log\log n)$ bound could be replaced by an $O(\sqrt{\log n})$ bound if we allow randomization or space unbounded in terms of n. Here time bounds for dynamic searching include both searching and updates. These fast dynamic search bounds gave corresponding $o(n\log n)$ bounds for sorting. Fredman and Willard asked the fundamental question: *how fast can we search [integers on a RAM]?*

In 1992, Willard [37, Lemma 3.3] showed that Fredman and Willard's construction could be de-amortized so as to get the same worst-case bounds for all operations.

In 1996, Andersson [2] introduced *exponential search trees* as a general amortized technique reducing the problem of searching a dynamic set in linear space to the problem of creating a search structure for a static set in polynomial time and space. The search time for the static set essentially becomes the amortized search time in the dynamic set. From Fredman and Willard [18], he essentially got a static search structure with $O(\sqrt{\log n})$ search time, and thus he obtained an $O(\sqrt{\log n})$ amortized time bound for dynamic searching in linear space.

In 1999, Beame and Fich showed that $\Theta(\sqrt{\log n/\log\log n})$ is the exact complexity of searching a static set using polynomial space and preprocessing time [7]. Using the above exponential search trees, they got an $\Theta(\sqrt{\log n/\log\log n})$ amortized cost for dynamic searching in linear space.

Finally, in 2000, Andersson and Thorup [5] developed a worst-case version of exponential search trees, giving an *optimal* $O(\sqrt{\log n/\log\log n})$ worst-case time bound for dynamic searching. This article is the combined journal version of [2, 5, 6]. In particular it describes the above mentioned exponential search trees, both the simple amortized ones from [2] and the worst-case ones from [5].

1.5 Bounds in terms of the word length and the maximal key

Besides the above mentioned bounds in terms of n, we get the following worst-case linear space trade-offs between the number n, the word length W, and the maximal key $U < 2^W$: $O(\min\{\log\log n + \log n/\log W,\ \log\log n \cdot \frac{\log\log U}{\log\log\log U}\})$. The last bound should be compared with van Emde Boas' bound of $O(\log\log U)$ [34, 35] that requires randomization (hashing) in order to achieve linear space [26].

1.6 AC^0 operations

As an additional challenge, Fredman and Willard [18] asked how quickly we can search on a RAM if all the computational instructions are AC^0 operations. A computational instruction is an AC^0 operation if it is computable by an $W^{O(1)}$-sized constant depth circuit with $O(W)$ input and output bits. In the circuit we may have negation, and-gates, and or-gates with unbounded fan-in. Addition, shift, and bit-wise Boolean operations are all AC^0 operations. On the other hand, multiplication is not. Fredman and Willard's own techniques [18] were heavily based on multiplication, but, as shown in [4] they can be implemented with AC^0 operations if we allow some non-standard operations that are not part of the usual instruction set. However, as mentioned previously, here we only allow standard operations, defined as RAM operations available in C [23].

Our $O(\sqrt{\log n/\log\log n})$ search structure is strongly based on multiplication. So far, even if we allow amortization and randomization, no search structure using standard AC^0 operations has been presented using polynomial space and $o(\log n)$ time, not even for the static case. Without requirements of polynomial space, Andersson [1] has presented a deterministic worst-case bound of $O(\sqrt{\log n})$. In this paper, we will present a linear space worst-case time bound of $O((\log n)^{3/4+o(1)})$ using only standard AC^0 operations, thus surpassing the $O(\log n)$ bound even in this restricted case.

1.7 Finger searching and finger updates

By *finger search* we mean that we can have a "finger" pointing at a stored key x when searching for a key y. Here a finger is just a reference returned to the user when x is inserted or searched for. The goal is to do better if the number q of stored keys between x and y is small.

We also have *finger updates*. For a finger insertion, we are given a finger to the key after which the new key is to be inserted. To implement a regular (non-finger) insertion of x, we

can first search x and then use the returned pointer as the finger. The finger delete is just a regular delete as defined above, i.e. we are given a pointer to the key to be deleted.

In the comparison-based model of computation Dietz and Raman [14] have provided optimal bounds, supporting finger searches in $O(\log q)$ time while supporting finger updates in constant time. Brodal et al. [9] managed to match these results on a pointer machine.

In this paper we present optimal bounds on the RAM; namely $O(\sqrt{\log q/\log\log q})$ for finger search with constant time finger updates. Also, we present the first finger search bounds that are efficient in terms of the absolute distance $|y-x|$ between x and y.

1.8 String searching

We will also consider the case of string searching where each key may be distributed over multiple words. Strings are then ordered lexicographically. One may instead be interested in variable length multiple-word integers where integers with more words are considered larger. However, by prefixing each integer with its length, we reduce this case to lexicographic string searching.

Generalizing search data structures for string searching is nontrivial even in the simpler comparison-based setting. The first efficient solution was presented by Mehlhorn [25, §III]. While the classical method requires weight-balanced search structures, our approach contains a direct reduction to any unweighted search structure. With inspiration from [2, 5, 7, 25] we show that if the longest common prefix between a key y and the stored keys has ℓ words, we can search y in $O(\ell + \sqrt{\log n/\log\log n})$ time, where n is the current number of keys. Updates can be done within the same time bound. Assuming that we can address the stored keys, our extra space bound is $O(n)$.

The above search bound is optimal, for consider an instance of the 1-word dynamic search problem, and give all keys a common prefix of ℓ words. To complete a search we both need to check the prefix in $O(\ell)$ time, and to perform the 1-word search, which takes $\Omega(\ell + \sqrt{\log n/\log\log n})$ [7].

Note that one may think of the strings as divided into characters much smaller than words. However, if we only deal with one such character at the time, we are not exploiting the full power of the computer at hand.

1.9 Techniques and main theorems

Our main technical contribution is to introduce exponential search trees providing a general reduction from the problem of maintaining a worst-case dynamic linear spaced structure to the simpler problem of constructing static search structure in polynomial time and space. For example, the polynomial construction time allows us to construct a dictionary deterministically with look-ups in constant time. Thus we can avoid the use of randomized hashing in, e.g., a van Emde Boas' style data structure [34, 18, 26]. The reduction is captured by the following theorem:

Theorem 1 *Suppose a static search structure on d integer keys can be constructed in $O(d^{k-1})$, $k \geq 2$, time and space so that it supports searches in $S(d)$ time. We can then construct a dynamic linear space search structure that with n integer keys supports insert, delete, and searches in time $T(n)$ where*

$$T(n) \leq T(n^{1-1/k}) + O(S(n)). \tag{1}$$

The reduction itself uses only standard AC^0 operations.

We then prove the following result on static data structures:

Theorem 2 *In polynomial time and space, we can construct a deterministic data structure over d keys supporting searches in $O(\min\{\sqrt{\log d}, \log\log U, 1 + \frac{\log d}{\log W}\})$ time where W is the word length, and $U < 2^W$ is an upper bound on the largest key. If we restrict ourselves to standard AC^0 operations, we can support searches in $O((\log d)^{3/4+o(1)})$ worst-case time per operation.*

The $\sqrt{\log d}$ and $\log\log U$ bounds above have been further improved by Beame and Fich:

Theorem 3 (Beame and Fich [7]) *In polynomial time and space, we can construct a deterministic data structure over d keys supporting searches in $O(\min\{\sqrt{\log d/\log\log d}, \frac{\log\log U}{\log\log\log U}\})$ time.*

Applying the recursion from Theorem 1, substituting $S(d)$ with (i) the two bounds in Theorem 3, (ii) the last bound in the min-expression in Theorem 2, and (iii) the AC^0 bound from Theorem 2, we immediately get the following four bounds:

Corollary 4 *There is a fully-dynamic deterministic linear space search structure supporting insert, delete, and searches in worst-case time*

$$O\left(\min\left\{\begin{array}{l}\sqrt{\log n/\log\log n}\\ \log\log n \cdot \frac{\log\log U}{\log\log\log U}\\ \log\log n + \frac{\log n}{\log W}\end{array}\right\}\right) \tag{2}$$

where W is the word length, and $U < 2^W$ is an upper bound on the largest key. If we restrict ourselves to standard AC^0 operations, we can support all operations in $O((\log n)^{3/4+o(1)})$ worst-case time per operation.

It follows from the lower bound by Beame and Fich [7] that our $O(\sqrt{\log n/\log\log n})$ bound is optimal.

1.9.1 Finger search

A finger search version of Theorem 1 leads us to the following finger search version of Corollary 4:

Theorem 5 *There is a fully-dynamic deterministic linear space search structure that supports finger updates in constant time, and given a finger to a stored key x, searches a key $y > x$ in time*

$$O\left(\min\left\{\begin{array}{l}\sqrt{\log q/\log\log q}\\ \log\log q\cdot\frac{\log\log(y-x)}{\log\log\log(y-x)}\\ \log\log q+\frac{\log q}{\log W}\end{array}\right\}\right)$$

where q is the number of stored keys between x and y. If we restrict ourselves to AC^0 operations, we still get a bound of $O((\log q)^{3/4+o(1)})$.

1.9.2 String searching

We give a general reduction from string searching to 1-word searching:

Theorem 6 *For the dynamic string searching problem, if the longest common prefix between a key x and the other stored keys has ℓ words, we can insert, delete, and search x in $O(\ell + \sqrt{\log n/\log\log n})$ time, where n is the current number of keys. In addition to the stored keys themselves, our space bound is $O(n)$.*

1.10 Contents

First, in Section 2, we present a simple amortized version of exponential search trees; the main purpose is to make the reader understand the basic mechanisms. Then, in Section 3 we give the worst-case efficient version, which require much more elaborate constructions. In Section 4 we construct the static search structures to be used in the exponential search tree. In Section 5 we reduce the update time to a constant in order to support finger updates. In Section 6, we describe the data structure for finger searching. In Section 7, we describe the data structure for string searching. In Section 8 we give examples of how the techniques of this paper have been applied in other work. Finally, in Section 9, we finish with an open problem.

2 The main ideas and concepts in an amortized setting

Before presenting our worst-case exponential search trees, we here present a simpler amortized version from [2], converting static data structures into fully-dynamic amortized search structures. The basic definitions and concepts of the amortized construction will be assumed for the more technical worst-case construction. As this version of exponential search trees is much simpler to describe than the worst-case version, it should be of high value for the interested reader as it hopefully will provide a good understanding of the main ideas.

2.1 Exponential search trees

An *exponential search tree* is a leaf-oriented multiway search tree where the degrees of the nodes decrease doubly-exponentially down the tree. By *leaf-oriented*, we mean that all keys are stored in the leaves of the tree. Moreover, with each node, we store a *splitter* for navigation: if a key arrives at a node, searching locally among the splitters of the children determines which child it belongs under. Thus, if a child v has splitter s and its successor has splitter s', a key y belongs under v if $y \in [s, s')$. We require that the splitter of an internal node equals the splitter of its leftmost child.

We also maintain a doubly-linked list over the stored keys, providing successor and predecessor pointers as well as maximum and minimum. A search in an exponential search tree may bring us to the successor of the desired key, but if the found key is too large, we just return its predecessor.

In our exponential search trees, the local search at each internal node is performed using a static local search structure, called an *S-structure*. We assume that an S-structure over d keys can be built in $O(d^{k-1})$ time and space and that it supports searches in $S(d)$ time. We define an exponential search tree over n keys recursively:

- The root has degree $\Theta(n^{1/k})$.
- The splitters of the children of the root are stored in a local S-structure with the properties stated above.
- The subtrees are exponential search trees over $\Theta(n^{1-1/k})$ keys.

It immediately follows that searches are supported in time

$$\begin{aligned} T(n) &= O\left(S\left(O(n^{1/k})\right)\right) + T\left(O(n^{1-1/k})\right) \\ &= O\left(S\left(O(n^{1/k})\right)\right) + O\left(S\left(O(n^{(1-1/k)/k})\right)\right) + T\left(O(n^{(1-1/k)^2})\right) \\ &= O\left(S\left(n\right)\right) + T\left(n^{1-1/k}\right). \end{aligned}$$

Above, the first equation follows by applying the recurrence formula to itself. For the second equation, we use that $k > 1$. Then for $n = \omega(1)$, we have $n \geq O(n^{1/k}) \geq O(n^{(1-1/k)/k})$ and $n^{1-1/k} \geq O(n^{(1-1/k)^2})$. For $n = O(1)$, we trivially have $T(n) = O(1) = O\left(S\left(n\right)\right)$.

Next we argue that the space of an exponential search tree is linear. Let n be the total number of keys, and let n_i be the number of keys in the ith subtree of the root. Then $n_i = \Theta(n^{1-1/k})$ and $\sum_i n_i = n$. The root has degree $d = \Theta(n^{1/k})$ and it uses space $O\left(d^{k-1}\right) = O\left(n^{1-1/k}\right)$. Hence the total space $C(n)$ satisfies the recurrence:

$$\begin{aligned} C(n) &= O(n^{1-1/k}) + \sum_i C(n_i) \quad \text{where } \sum_i n_i = n \text{ and } \forall i : n_i = O(n^{1-1/k}) \\ \Rightarrow \qquad C(n) &= O(n). \end{aligned}$$

Since $O(d^{k-1})$ bounds not only the space but also the construction time for the S-structure at a degree d node, the same argument gives that we can construct an exponential search tree over n keys in linear time.

2.2 Updates

Recall that an update is implemented as a search, as described above, followed by a finger update. A finger delete essentially just removes a leaf. However, if the leaf was the first child of its parent, its splitter has to be transferred to the new first child. For a finger insert of a key y, we get a reference (finger) to the key x after which y is to be inserted. We then also have to consider the successor z of x. Let s be the splitter of z. If $y < s$, we place y after x under the parent of x, and let y be its own splitter. If $y \geq s$, we place y before z under the parent of z, and give y splitter s and make z its own splitter.

The method for maintaining splitters is illustrated in Figure 2.2. In (a) we see a leaf-oriented tree-like structure with splitters. Each internal node has the same splitter as its leftmost child. In (b) we see the tree after deleting leaf 30; we transform the splitter 30 to the right neighbor. Note that the splitter 30 remains in the tree although the key 30 has been deleted. Finally (c), when inserting 35, we will be given a reference to the nearest smaller keys, that is 20. Since 35 is larger than the splitter of the successor, we place 35 under the successor's parent and adjust the splitters accordingly.

Balance is maintained in a standard fashion by global and partial rebuilding. By the weight, $|t|$, of a (sub-)tree t we mean the number of leaves in t. By the weight, $|v|$, of a node v, we mean the weight of the tree rooted at v. When a subtree gets too heavy, by a factor of 2, we split it in two, and if it gets too light, by a factor of 2, we join it with its neighbor. By the analysis above, constructing a new subtree rooted at the node v takes $O(|v|)$ time. In addition, we need to update the S-structure at v's parent u, in order to reflect the adding or removing of a key in u's list of child splitters. Since u has $\Theta(|u|^{1/k})$ children, the construction time for u's S-structure is $O((|u|^{1/k})^{k-1}) = O(|u|^{1-1/k})$. By definition, this time is $O(|v|)$. We conclude that we can reconstruct the subtrees and update the parent's S-structure in time linear in the weight of the subtrees.

Exceeding the weight constraints requires that a constant fraction of the keys in a subtree have been inserted and deleted since the subtree was constructed with proper weight. Thus, the reconstruction cost is an amortized constant per key inserted or deleted from a tree. Since the depth of an exponential search tree is $O(\log \log n)$, the update cost, excluding the search cost for finding out were to update, is $O(\log \log n)$ amortized. This completes our sketchy description of amortized exponential search trees.

3 Worst-case exponential search trees

The goal of this section is to prove the statement of Theorem 1:

> *Suppose a static search structure on d integer keys can be constructed in $O(d^{k-1})$, $k \geq 2$, time and space so that it supports searches in $S(d)$ time. We can then construct a dynamic linear space search structure that with n integer keys supports insert, delete, and searches in time $T(n)$ where $T(n) \leq T(n^{1-1/k}) + O(S(n))$. The reduction itself uses only standard AC^0 operations.*

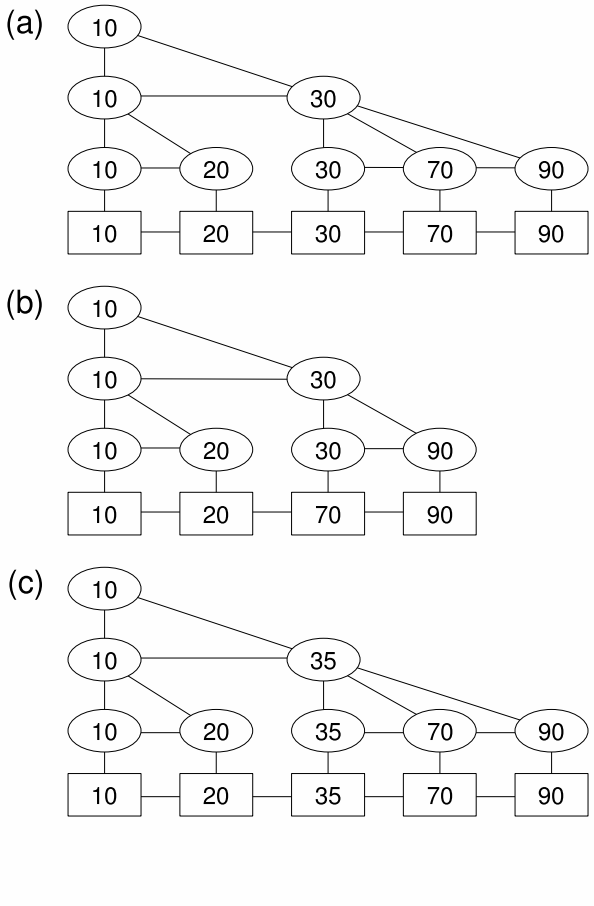

Figure 2.2: Maintaining splitters

In order to get from the amortized bounds above to worst-case bounds, we need a new type of data structure. Instead of a data structure where we occasionally rebuild entire subtrees, we need a multiway tree which is something more in the style of a standard B-tree, where balance is maintained by locally joining and splitting nodes. By locally we mean that the joining and splitting is done just by joining and splitting the children sequences. This type of data structure is for example used by Willard [37] to obtain a worst-case version of fusion trees.

One problem with the previous definition of exponential search trees is that the criteria for when subtrees are too large or too small depend on their parents. If two subtrees are joined, the resulting subtree is larger, and according to our recursive definition, this may imply that all of the children simultaneously become too small, so they have to be joined, etc. To avoid such cascading effects of joins and splits, we redefine the exponential search tree as follows:

Definition 7 *In an exponential search tree all leaves are on the same depth, and we define the height or level of a node to be the unique distance from the node to the leaves descending from it. For a non-root node v at height $i > 0$, the weight (number of descending leaves) is $|v| = \Theta(n_i)$ where $n_i = \alpha^{(1+1/(k-1))^i}$ and $\alpha = \Theta(1)$. If the root has height h, its weight is $O(n_h)$.*

With the exception of the root, Definition 7 follows our previous definition of exponential search trees, that is, if v is a non-root node, it has $\Theta(|v|^{1/k})$ children, each of weight $\Theta(|v|^{1-1/k})$.

In the following, we will use explicit constants

Our main challenge is now to rebuild S-structures in the background so that they remain sufficiently updated as nodes get joined and split. In principle, this is a standard task (see e.g. [38]). Yet it is a highly delicate puzzle which is typically either not done (e.g. Fredman and Willard [18] only claimed amortized bounds for their original fusion trees), or done with rather incomplete sketches (e.g. Willard [37] only presents a 2-page sketch of his de-amortization of fusion trees). Furthermore, our exponential search trees pose a new complication; namely that when we join or split, we have to rebuild not only the S-structures of the nodes being joined or split, but also the S-structure of their parent. For contrast, when Willard [37] de-amortizes fusion trees, he actually uses the "atomic heaps" from [19] as S-structures, and these atomic heaps support insert and delete in constant time. Hence, when nodes get joined or split, he can just delete or insert the splitter between them directly in the S-structure of the parent, without having to rebuild it.

In this section, we will present a general quotable theorem about rebuilding, thus making proper de-amortization much easier for future research.

3.1 Coping with interference between processes: the idea of compensation

Consider a process A that traverses a list. When A begins, the length of the list is m and A is given m steps to traverse the list, one element per step. Now, assume that between two

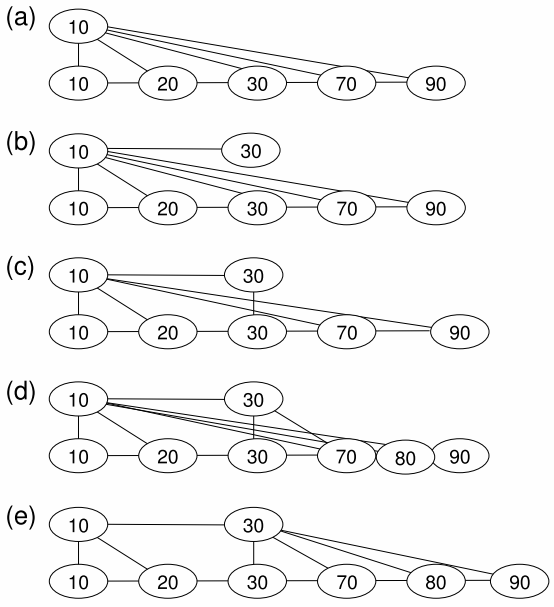

Figure 3.3: Compensation.

steps of A, another process B adds a new element to the list. Then, in order to ensure that A will not need any extra step, we let B *compensate* A by progressing A by one step. This type of compensation will be used frequently in our worst-case techniques to ensure proper scheduling.

We illustrate this compensation technique in Figure 3.3. In (a) we see a small tree with one root and five children, the five children are also linked together in a list. (b) shows the initial step of splitting the root into two nodes; the three rightmost children should be redirected to belong to the new node. We initialize a process A of traversing the list of these three nodes in three steps, changing appropriate pointers. In (c) we have performed one step of the redirection process, updating the node 30. (d) shows what happen when another process B splits the child node 70 into two nodes, 70 and 80, while A is not yet finished. Then, B compensates A by performing one of A's step. Hence, A will still need only two steps to finish (e).

3.2 Join and split with postprocessing

As mentioned, we are going to deal generally with multiway trees where joins and splits cannot be completed in constant time. For the moment, our trees are only described structurally with a children list for each non-leaf node. Then joins and splits can be done in constant time. However, after each join or split, we want to allow for some unspecified *postprocessing*

before the involved nodes can participate in new joins and splits. This postprocessing time will, for example, be used to update parent pointers and S-structures.

The key issue is to schedule the postprocessing, possibly involving reconstruction of static data structures, so that we obtain good worst-case bounds. We do this by dividing each postprocessing into *local update steps* and ensuring that each update only uses a few local update steps at the same time as each postprocessing is given enough steps to complete. If a data structure is changed by some other operation before the postprocessing is finished, the postprocessing will be compensated, and therefore it will always finish within the number of steps allocated.

To be more specific in our structural description of a tree, let u be the predecessor of v in their parent's children list C. A *join* of u and v means that we append the children list of v to that of u so that u adopts all these children from v. Also, we delete v from C. Similarly, a *split* of a node u at its child w means that we add a new node v after u in the children list C of u's parent, that we cut the children list of u just before w, and make the last part the children list of v. Structural splits and joins both take constant time and are viewed as atomic operations. In the postprocessing of a join, the resulting node is not allowed to participate in any joins or splits. In the postprocessing of a split, the resulting nodes are neither allowed to participate directly in a join or split, nor is the parent allowed to split between them.

We are now in the position to present our general theorem on worst-case bounds for joining and splitting with postprocessing (the relation between the constants below and Definition 7 will be clarified in Lemma 11):

Theorem 8 *Given a number series $n_1, n_2, \ldots$, with $n_1 \geq 84$, $n_{i+1} > 19n_i$, we can schedule split and joins to maintain a multiway tree where each non-root node v on height $i > 0$ has weight between $n_i/4$ and n_i. A root node on height $h > 0$ has weight at most n_h and at least 2 children, so the root has weight at least $n_{h-1}/2$. The schedule gives the following properties:*

(i) When a leaf v is inserted or deleted, for each node u on the path from v to the root the schedule uses one local update step contributing to the postprocessing of at most one join or split involving either u or a neighbor of u. In addition to the local updates, the schedule spends constant time on each level.

(ii) For each split or join at level i the schedule ensures that we have $n_i/84$ local update steps available for postprocessing, including one at the time of the split or join.

The above numbers ensure that a node which is neither root nor leaf has at least $(n_i/4)/n_{i-1} = 19/4 > 4$ children. If the root node is split, a new parent root is generated implicitly. Conversely, if the root's children join to a single child, the root is deleted and the single child becomes the new root. The proof of Theorem 8 is rather delicate, and deferred till later. Below we show how to apply Theorem 8 in exponential search trees. As a first simple application of the schedule, we show how to compute parents.

Lemma 9 *Given Theorem 8, the following property can be added to the theorem: the parent of any node can be computed in constant time.*

Proof: With each node we maintain a parent pointer which points to the true parent, except possibly during the postprocessing of a split or join. Split and joins are handled equivalently. Consider the case of a join of u and v into u. During the postprocessing, we will redirect all the parent pointers of the old children of v to point to u. Meanwhile, we will have a forward pointer from v to u so that parent queries from any of these children can be answered in constant time, even if the child still points to v.

Suppose that the join is on level i. Then v could not have more than n_i children. Hence, if we redirect 84 of their parent pointers in each local update step, we will be done by the end of the postprocessing of Theorem 8. The redirections are done in a traversal of the children list, starting from the old first child of v. One technical detail is, however, that we may have join and split in the children sequence. Joins are not a problem, but each splitting will cause the number of children below v to increase by one. Therefore, when a split occurs at one of v's children, that split process will compensate the process of redirecting pointers from v by performing one step of the redirection; this compensation will be part of initializing the split postprocessing. ■

For our exponential search trees, we will use the postprocessing for rebuilding S-structures. We will still keep a high level of generality to facilitate other applications, such as, for example, a worst-case version of the original fusion trees [18].

Corollary 10 *Given a number series $n_0, n_1, n_2, \ldots$, with $n_0 = 1$, $n_1 \geq 84$, $n_{i+1} > 19n_i$, we maintain a multiway tree where each node at height i which is neither the root nor a leaf node has weight between $n_i/4$ and n_i. A root node on height $h > 0$ has weight at most n_h and at least 2 children, so the root has weight at least $n_{h-1}/2$. Suppose an S-structure for a node on height i can be built in $O(n_{i-1}t_i)$ time. We can then maintain S-structures for the whole tree supporting a finger update in $O(h + \sum_{i=1}^{h} t_i)$ time where h is the current height of the tree.*

Proof: In Section 2, we described how a finger update, in constant time, translates into the insertion or deletion of a leaf. We can then apply Theorem 8.

Our basic idea is that we have an ongoing periodic rebuilding of the S-structure at each node v. A period starts by scanning the splitter list of the children of v in $O(n_i/n_{i-1})$ time. It then creates a new S-structure in $O(n_{i-1}t_i)$ time, and finally, in constant time, it replaces the old S-structure with the new S-structure. The whole rebuilding is divided into $n_{i-1}/160$ steps, each taking $O(t_i)$ time.

Now, every time an update contributes to a join or split postprocessing on level $i-1$, we perform one step in the rebuilding of the S-structure of the parent p, which is on level i. Then Theorem 8 ascertains that we perform $n_{i-1}/84$ steps on $S(p)$ during the postprocessing. Hence, we have at least one complete rebuilding of $S(p)$ without the splitter removed by the join or, in case of a split, with the added splitter.

When two neighboring nodes u and v on level $i-1$ join to u, the next rebuilding of $S(u)$ will automatically include the old children of v. The maintenance of the disappearing node v, including rebuilding of $S(v)$ when needed, is continued for all updates belonging under

the old v; this maintenance is ended when the creation of $S(u)$ for the joined node u is completed. Furthermore, each step of rebuilding $S(v)$ for the old v will also compensate the rebuilding of $S(u)$, in this way we make sure that the children of v and u do not experience any delay in the rebuilding of the S-structure of their parents. Note that $S(u)$ is completely rebuilt in $n_{i-2}/84$ steps which is much less than the n_{i-1} steps we have available for the postprocessing.

During the postprocessing of the join, we may, internally, have to forward keys between u and v. More precisely, if a key arrives at v from the S-structure of the parent and $S(u)$ has been updated to take over $S(v)$, the key is transferred to $S(u)$. Conversely, $S(v)$ is still in use; if a key arriving at u is larger than or equal to the splitter of v it is transferred to $S(v)$.

The split is implemented using the same ideas: all updates for the two new neighboring nodes u and v promote both $S(u)$ and $S(v)$. For $S(u)$, we finish the current rebuilding over all the children before doing a rebuild excluding the children going to v. By the end of the latter rebuild, $S(v)$ will also have been completed. If a key arrives at v and $S(v)$ is not ready, we transfer it to $S(u)$. Conversely, if $S(v)$ is ready and a key arriving at u is larger than or equal to the splitter of v, the key is transferred to $S(v)$. ■

Below we establish some simple technical lemmas verifying that Corollary 10 applies to the exponential search trees from Definition 7. The first lemma shows that the number sequences n_i match.

Lemma 11 *With $\alpha = \max\{84^{(k-1)/k}, 19^{(k-1)^2/k}\}$ and $n_i = \alpha^{(1+1/(k-1))^i}$ as in Definition 7, then $n_1 \geq 84$ and $n_{i+1}/n_i \geq 19$ for $i \geq 1$.*

Proof: $n_1 \geq (84^{(k-1)/k})^{1+1/(k-1)} = 84$ and $n_{i+1}/n_i = n_{i+1}^{1/k} \geq n_2^{1/k} \geq \alpha^{(k/(k-1))^2/k} \geq 19$. ■

Next, we show that the S-structures are built fast enough.

Lemma 12 *With Definition 7, creating an S-structure for a node u on level i takes $O(n_{i-1})$ time, and the total cost of a finger update is $O(\log\log n)$.*

Proof: Since u has degree at most $\Theta(n_i^{1/k})$, the creation takes $O((n_i^{1/k})^{k-1}) = O((n_i^{1-1/k}) = O(n_{i-1})$ time. Thus, we get $t_i = O(1)$ in Corollary 10, corresponding to a finger update time of $O(\log\log n)$. ■

Since $S(n) = \Omega(1)$, any time bound derived from Theorem 1 is $\Omega(\log\log n)$, dominating our cost of a finger update.

Next we give a formal proof that the recursion formula of Theorem 1 holds.

Lemma 13 *Assuming that the cost for searching in a node of degree d is $O(S(d))$, the search time for an n key exponential search tree from Definition 7 is bounded by $T(n) \leq T(n^{1-1/k}) + O(S(n))$ for $n = \omega(1)$.*

Proof: Since $n_{i-1} = n_i^{1-1/k}$ and since the degree of a level i node is at most $4n_i^{1/k}$, the search time starting just below the root at level $h-1$ is bounded by $T'(n_{h-1})$ where $n_{h-1} < n$ and $T'(m) \leq T'(m^{1-1/k}) + O(S(4m^{1/k}))$. Moreover, for $m = \omega(1)$, $4m^{1/k} > m$, so $O(S(4m^{1/k})) = O(S(m))$.

The degree of the root is bounded by n, so the search time of the root is at most $S(n)$. Hence our total search time is bounded by $S(n) + T'(n_{h-1}) = O(T(n))$. Finally, the O in $O(T(n))$ is superfluous because of the O in $O(S(n))$. ∎

Finally, we prove that exponential search trees use linear space.

Lemma 14 *The exponential search trees from Definition 7 use linear space.*

Proof: Consider a node v at height i. The number of keys below v is at least $n_i/4$. Since v has degree at most $4n_i^{1/k}$, the space of the S-structure by v is $O((4n_i^{1/k})^{k-1}) = O(n_i^{1-1/k})$. Distributing this space on the keys descending from v, we get $O(n_i^{-1/k})$ space per key.

Conversely, for a given key, the space attributed to the key by its ancestors is $O(\sum_{i=0}^{h} n_i^{-1/k}) = O(1)$. ∎

The above lemmas establish that **Theorem 1 holds if we can prove Theorem 8**.

3.3 Proving Theorem 8: A game of weight balancing

In order to prove Theorem 8, we consider a game on lists of weights. In relation to Theorem 8, each list represents the weights of the children of a node on some fixed level. The purpose of the game is to crystallize what is needed for balancing on each level. Later, in a bottom-up induction, we will apply the game to all levels.

It is worth noticing that the weight balancing protocol used to prove Theorem 8 relies on Lemma 9, which in turn relies on Theorem 8. How this is possible without a circular proof is stated at the end of this section, when describing how to find a path from a leaf to a root. (These dependencies between lemmas and theorems is the main reason for the order in which they are presented.)

3.3.1 A Protocol for a Weight Balancing Game

First we define the game on a single list of non-negative integer weights. The players are *us* against an adversary. Our goal is to maintain balance in the sense that for some parameter b, called the *latency*, we want all weights to be of size $\Theta(b)$. More precisely, for some concrete constants $c_1 < c_2 < c_3 < c_4$, the list is *balanced* if all weights are in the interval $[c_1 b, c_4 b]$. Initially, the list is *neutral* in the sense that all weights are in the smaller interval $[c_2 b, c_3 b]$.

The goal of the adversary is to upset balance, pushing some weight outside $[c_1 b, c_4 b]$. The adversary has the first turn, and when it is his turn, he can change the value of an arbitrary weight by one. He has the restriction that he may not take the sum of all weights in the lists down below $c_2 b$.

After the adversary has changed a weight, we get to work locally on balance. We may join neighboring weights w_1 and w_2 into one $w = w_1 + w_2$, or split a weight w into w_1 and w_2. The adversary decides on the values of w_1 and w_2, but he must fulfill that $w_1 + w_2 = w$ and that $|w_1 - w_2| \leq \Delta b$. Here Δ is called the *split error*.

Each split or join is followed by a postprocessing requiring b steps during which the involved weights may not participate in any other split or join. When it is our turn, we get to perform one such step. Moreover, the step may only be performed *locally* in the sense of being on a postprocessing involving the weight last changed by the adversary, or a neighboring weight.

We will now extend the game to a family of lists. This gives the adversary some alternatives to the previous weight updates. He may add or remove a neutral list, that is, a list with all weights in $[c_2 b, c_3 b]$. He may also cut or concatenate lists.

However, the adversary may not do the cuts arbitrarily. He may not cut between two weights coming out of a split until the split postprocessing is completed. Moreover, we have the power to *tie* some neighboring weights, and the adversary cannot cut ties. We ourselves can only tie and untie the last weight updated by the adversary or one of its neighbors.

An *uncuttable* segment is a segment of weights that the adversary may not cut. Our tying of weights is restricted in that for some constant $c_5 \geq c_4$, we have to make sure that the total weight of any uncuttable segment is at most $c_5 b$. In particular, this implies that no uncuttable segment contains more than $c_5/c_1 = O(1)$ weights.

A *protocol* for the balancing game is a winning strategy against any adversary. Also, to be a protocol it should be efficient identifying its moves in constant time.

Proposition 15 *For any latency b, split error Δ, and number μ, we have a protocol for the balancing game with $c_1 = \mu b$, $c_2 = (\mu + 3)b + 1$, $c_3 = (2\mu + \Delta + 9)b - 1$, $c_4 = (3\mu + \Delta + 14)b$, and $c_5 = (5\mu + \Delta + 20)b$. In particular, for $\Delta = 7$ and $\mu = 21$, we assume that neutral lists have weights strictly between $24b$ and $58b$, and that the total weight of any list is more than $24b$. Then we guarantee that all weights stay between $21b$ and $84b$ and that the maximum uncuttable segment is of size at most $132b$.*

The rest of this section is devoted to the proof of Proposition 15. First we will present a protocol. Next we will prove that the protocol is the winning strategy of Proposition 15.

We say a weight is *involved* if it is involved in a split or join postprocessing; otherwise it is *free*. Above we mentioned tying of weights preventing the adversary to cut between them. Our protocol will use tying as follows. If a free weight v has an involved neighbor w, the protocol may tie v to w. As long as v is free and tied to w, we will do a step of the postprocessing of w whenever we get an update of v. Then v cannot change much before the current postprocessing involving w is finished. When w is finished, it is free to join v, or to start something with the neighbor on the other side.

Let $s = 2\mu + \Delta + 9$ and $m = \mu + 3$. Our protocol is defined by the following rules:

(a) If a free weight gets up to sb, we start splitting it.

(b) If a free weight v gets down to mb and has a free neighbor w, we join v and w, untying w from any other neighbor.

(c) If a free weight v gets down to mb and has no free neighbors, we tie it to any one of its neighbors. If v later gets back up above mb, it is untied again.

(d) When we finish a join postprocessing, if the resulting weight is $\geq sb$, we immediately start splitting it. If a neighbor was tied to the joined weight, the tie is transferred to the nearest weight resulting from the split.

If the weight v resulting from the join is $< sb$ and v is tied by a neighbor, we join with the neighbor that tied v first.

(e) At the end of a split postprocessing, if any of the resulting weights are tied by a neighbor, it joins with that neighbor. Note here that since resulting weights are not tied to each other, there cannot be a conflict.

Note that our protocol is independent of legal cuts and concatenations by the adversary, except in (c) which requires that a free weight getting down to $(\mu + 3)b$ has at least one neighbor. This is, however, ensured by the condition from Proposition 15 that each list has total weight strictly larger than $(\mu + 3)b$.

Lemma 16

(i) Each free weight is between μb and $sb = (2\mu + \Delta + 9)b$.

(ii) The weight in a join postprocessing is between $(m + \mu - 1)b = (2\mu + 2)b > mb$ and $(s + m + 1)b = (3\mu + \Delta + 13)b > sb$.

(iii) In a split postprocessing, the total weight is at most $(3\mu + \Delta + 14))b$ and the split weights are between $((s - \Delta)/2 - 1)b = (\mu + 3.5)b > mb$ and $((s + m + 1 + \Delta)/2 + 1)b = (1.5\mu + \Delta + 7.5)b < sb$.

Proof: First we prove some simple claims.

Claim 16A *If (i), (ii), and (iii) are true when a join postprocessing starts, then (ii) will remain satisfied for that join postprocessing.*

Proof: For the upper bound note that when the join is started, none of the involved weights can be above sb, for then we would have split it. Also, a join has to be initiated by a weight of size at most mb, so when we start the total weight is at most $(s + m)b$, and during the postprocessing, it can increase by at most b.

For the lower bound, both weights have to be at least μb. Also, the join is either initiated as in (b) by a weight of size mb, or by a tied weight coming out from a join or split, which by (ii) and (iii) is of size at least mb, so we start with a total of at least $(\mu + m)b$, and we loose at most b in the postprocessing. □

Claim 16B *If (i), (ii), and (iii) are true when a split postprocessing starts, then (iii) will remain satisfied for that split postprocessing.*

PROOF: For the lower bound, we know that a split is only initiated for a weight of size at least sb. Also, during the postprocessing, we can loose at most b, and since the maximal difference between the two weights is Δb, the smaller is of size at least $(s-1-\Delta)/2b$.

For the upper bound, the maximal weight we can start with is one coming out from a join, which by (ii) is at most $(s+m+1)b$. Hence the largest weight resulting from the split is at most $((s+m+1+\Delta)/2)b$. Both of these numbers can grow by at most b during the split postprocessing. □

We will now complete the proof of Lemma 16 by showing that there cannot be a first violation of (i) given that (ii) and (iii) have not already been violated. The only way we can possibly get a weight above sb is one coming out from a join as in (ii), but then by (d) it is immediately split, so it doesn't become free.

To show by contradiction that we cannot get a weight below μb, let w the first weight getting down below μb keys. When w was originally created by (ii) and (iii), it was of size $> mb$, so to get down to μb, there must have been a last time where it got down to mb. It then tied itself to an involved neighboring weight w'. If w' is involved in a split, we know that when w' is done, the half nearest w will immediately start joining with w as in (e). However, if w' is involved in a join, when done, the resulting weight may start joining with a weight w'' on the other side. In that case, however, w is the first weight to tie to the new join. Hence, when the new join is done, either w starts joining with the result, or the result get split and then w will join with the nearest weight coming out from the split. In the worst case, w will have to wait for two joins and one split to complete before it gets joined, and hence it can loose at most $3b = (m-\mu)b$ while waiting to get joined. ■

Proof of Proposition 15 By Lemma 16, all weights remain between μb and $(3\mu+\Delta+13)b$. Concerning the maximal size of an uncuttable segment, the maximal total weight involved in split or join is $(m+s+2)b$, and by (b) we can have a weight of size at most mb tied from either side, adding up to a total of $(3m+s+2)b = (5\mu+\Delta+20)b$. **This completes the proof of Proposition 15.**

3.3.2 Applying the protocol

We now want to apply our protocol in order to prove Theorem 8:

> *Given a number series $n_1, n_2, \ldots$, with $n_1 \geq 84$, $n_{i+1} > 19n_i$, we can schedule split and joins to maintain a multiway tree where each non-root node v on height $i > 0$ has weight between $n_i/4$ and n_i. A root node on height $h > 0$ has weight at most n_h and at least 2 children, so the root has weight at least $n_{h-1}/2$. The schedule gives the following properties:*
>
> *(i) When a leaf v is inserted or deleted, for each node u on the path from v to the root the schedule uses one local update step contributing to the postprocessing of at most one join or split involving either u or a neighbor of u. In addition to the local updates, the schedule spends constant time on each level.*

(ii) For each split or join at level i the schedule ensures that we have $n_i/84$ local update steps available for postprocessing, including one at the time of the split or join.

For each level $i < h$, the nodes are partitioned into children lists of nodes on level $i+1$. We maintain these lists using the scheduling of Proposition 15, with latency $b_i = n_i/84$, split error $\Delta = 7$, $\mu = 21$. This gives weights between $21b_i = n_i/4$ and $84b_i = n_i$, as required. We need to ensure that the children list of a node on level i can be cut so that the halves differ by at most Δb_i. For $i = 1$, this is trivial, in that the children list is just a list of leaves that can be cut anywhere, that is, we are OK even with $\Delta = 1$. For $i > 1$, inductively, we may assume that we have the required difference of Δb_{i-1} on level below, and then, using Proposition 15, we can cut the list on level i with a difference of $132b_{i-1}$. However, $b_i \geq 19b_{i-1}$, so $132b_{i-1} \leq 7b_i$, as required.

Having dealt with each individual level, three unresolved problems remain:

- To implement splits in constant time, how do we find a good place to cut the children list?
- How does the protocol apply as the height of the tree changes?
- How do we actually find the nodes on the path from the leaf v to the root?

Splitting in constant time For each node v on level i, our goal is to maintain a good cut child in the sense that when cutting at that child, the lists will not differ by more than Δb_i. We will always maintain the sum of the weights of the children preceding the cut child, and comparing that with the weight of v tells us if it is at good balance. If an update makes the preceding weight too large, we move to the next possible cut child to the right, and conversely, if it gets too small, we move the cut child to the left. A possible cut is always at most 4 children away, so the above shifts only take constant time. Similarly, if the cut child stops being cutable, we move in the direction that gives us the best balance.

When a new list is created by a join or a split, we need to find a new good cut child. To our advantage, we know that we have at least b_i update steps before the cut child is needed. We can therefore start by making the cut child the rightmost child, and every time we receive an update step for the join, we move to the right, stopping when we are in balance. Since the children list is of length $O(n_i/n_{i-1})$, we only need to move a constant number of children to the right in each update step in order to ensure balance before the postprocessing is ended.

Changing the height A minimal tree has a root on height 1, possibly with 0 children. If the root is on height h, we only apply the protocol when it has weight at least $21b_h$, splitting it when the protocol tells us to do so. Note that there is no cascading effect, for before the split, the root has weight at most $84b_h$, and this is the weight of the new root at height $h+1$. However $b_h \leq b_{h+1}/19$, so it will take many updates before the new root reaches the weight $21b_{i+1}$. The S-structure and pointers of the new root are created during the postprocessing of the split of the old root. Conversely, we only loose a root at height

$h+1$ when it has two children that get joined into one child. The cleaning up after the old root, i.e. the removal of its S-structure and a constant number of pointers, is done in the postprocessing of the join of its children. The new root starts with weight at least $21b_h$, so it has at least $21b_h/84b_{h-1} \geq 19/4 > 4$ children. Hence it will survive long enough to pay for its construction.

Finding the nodes on the path from the leaf v to the root The obvious way to find the nodes on the path from the leaf v to the root is to use parent pointers, which according to Lemma 9 can be computed in constant time. Thus, we can prove Theorem 8 from Lemma 9. The only problem is that we used the schedule of Theorem 8 to prove Lemma 9. To break the circle, consider the first time the statement of Theorem 8 or of Lemma 9 is violated. If the first mistake is a mistaken parent computation, then we know that the scheduling and weight balancing of Theorem 8 has not yet been violated, but then our proof of Lemma 9 based on Theorem 8 is valid, contradicting the mistaken parent computation. Conversely, if the first mistake is in Theorem 8, we know that all parents computed so far were correct, hence that our proof of Theorem 8 is correct. Thus there cannot be a first mistake, so **we conclude that both Theorem 8 and Lemma 9 are correct**.

4 Static search structures

In this section, we will prove Theorem 2:

> *In polynomial time and space, we can construct a deterministic data structure over d keys supporting searches in $O(\min\{\sqrt{\log d}, \log\log U, 1+\frac{\log d}{\log W}\})$ time where W is the word length, and $U < 2^W$ is an upper bound on the largest key. If we restrict ourselves to standard AC^0 operations, we can support searches in $O((\log d)^{3/4+o(1)})$ worst-case time per operation.*

To get the final bounds in Corollary 4, we actually need to improve the first bound in the min-expression to $O(\sqrt{\log n/\log\log n})$ and the second bound to $O(\log\log U/\log\log\log U)$. However, the improvement is by Beame and Fich [7]. We present our bounds here because (i) they are simpler (ii) the improvement by Beame and Fich is based on our results.

4.1 An improvement of fusion trees

Using our terminology, the central part of the fusion tree is a static data structure with the following properties:

Lemma 17 *(Fredman and Willard) For any d, $d = O\left(W^{1/6}\right)$, A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbor queries in $O(1)$ worst-case time.*

Fredman and Willard used this static data structure to implement a B-tree where only the upper levels in the tree contain B-tree nodes, all having the same degree (within a constant factor). At the lower levels, traditional (i.e. comparison-based) weight-balanced trees were used. The amortized cost of searches and updates is $O(\log n/\log d + \log d)$ for any $d = O\left(W^{1/6}\right)$. The first term corresponds to the number of B-tree levels and the second term corresponds to the height of the weight-balanced trees.

Using an exponential search tree instead of the Fredman/Willard structure, we avoid the need for weight-balanced trees at the bottom at the same time as we improve the complexity for large word sizes.

Lemma 18 *A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbor queries in $O\left(\frac{\log d}{\log W} + 1\right)$ worst-case time.*

Proof: We just construct a static B-tree where each node has the largest possible degree according to Lemma 17. That is, it has a degree of $\min\left(d, W^{1/6}\right)$. This tree satisfies the conditions of the lemma. ■

Corollary 19 *There is a data structure occupying linear space for which the worst-case cost of a search and update is $O\left(\frac{\log n}{\log W} + \log\log n\right)$*

Proof: Let $T(n)$ be the worst-case cost. Combining Theorem 1 and Lemma 18 gives that

$$T(n) = O\left(\frac{\log n}{\log W} + 1 + T\left(n^{4/5}\right)\right).$$

■

4.2 Tries and perfect hashing

In a binary trie, a node at depth i corresponds to an i-bit prefix of one (or more) of the keys stored in the trie. Suppose we could access a node by its prefix in constant time by means of a hash table, i.e. without traversing the path down to the node. Then, we could find a key x, or x's nearest neighbor, in $O(\log\log U)$ time by a binary search for the node corresponding to x's longest matching prefix. Recall here that U is a bound on the largest key so $\log U$ bounds the maximal key length which may be smaller than the word length W. At each step of the binary search, we look in the hash table for the node corresponding to a prefix of x; if the node is there we try with a longer prefix, otherwise we try with a shorter one.

The idea of a binary search for a matching prefix is the basic principle of the van Emde Boas tree [34, 35, 36]. However, a van Emde Boas tree is not just a plain binary trie represented as above. One problem is the space requirements; a plain binary trie storing d keys may contain as much as $\Theta(d\log U)$ nodes. In a van Emde Boas tree, the number of nodes is decreased to $O(d)$ by careful optimization.

In our application $\Theta(d\log U)$ nodes can be allowed. Therefore, to keep things simple, we use a plain binary trie.

Lemma 20 *A static data structure containing d keys and supporting neighbor queries in $O(\log\log U)$ worst-case time can be constructed in $O(d^4)$ time and space. The implementation can be done without division.*

Proof: We study two cases.

Case 1: $\log U > d^{1/3}$. Since $W \geq \log U$, Lemma 18 gives constant query cost.

Case 2: $\log U \leq d^{1/3}$. In $O(d \log U) = o(d^2)$ time and space we construct a binary trie of height $\log U$ containing all d keys. Each key is stored at the bottom of a path of length $\log U$ and the keys are linked together. In order to support neighbor queries, each unary node contains a neighbor pointer to the next (or previous) leaf according to the inorder traversal.

To allow fast access to an arbitrary node, we store all nodes in a perfect hash table such that each node of depth i is represented by the i bits on the path down to the node. Since the paths are of different length, we use $\log U$ hash tables, one for each path length. Each hash table contains at most d nodes. The algorithm by Fredman, Komlos, and Szemeredi [17] constructs a hash table of d keys in $O(d^3 \log U)$ time. The algorithm uses division, this can be avoided by simulating each division in $O(\log U)$ time. With this extra cost, and since we use $\log U$ tables, the total construction time is $O\left(d^3 \log^3 U\right) = O(d^4)$ while the space is $O(d \log U) = o(d^2)$.

With this data structure, we can search for a key x in $O(\log\log U)$ time by a binary search for the node corresponding to x's longest matching prefix. This search either ends at the bottom of the trie or at a unary node, from which we find the closest neighboring leaf by following the node's neighbor pointer.

During a search, evaluation of the hash function requires integer division. However, as pointed out by Knuth [24], division with some precomputed constant p may essentially be replaced by multiplication with $1/p$. Having computed $r = \lfloor 2^{\log U}/p \rfloor$ once in $O(\log U)$ time, we can compute x DIV p as $\lfloor xr/2^{\log U} \rfloor$ where the last division is just a right shift $\log U$ positions. Since $\lfloor x/p \rfloor - 1 < \lfloor xr/2^{\log U} \rfloor \leq \lfloor x/p \rfloor$ we can compute the correct value of x DIV p by an additional test. Once we can compute DIV, we can also compute MOD. ■

An alternative method for perfect hashing without division is the one developed by Raman [30]. Not only does this algorithm avoid division, it is also asymptotically faster, $O(d^2 \log U)$.

Corollary 21 *There is a data structure occupying linear space for which the worst-case cost of a search and the amortized cost of an update is $O(\log\log U \log\log n)$.*

Proof: Let $T(n)$ be the worst-case search cost. Combining Lemmas 1 and 20 gives $T(n) = O(\log\log U) + T\left(n^{4/5}\right)$. ■

4.3 Finishing the proof of Theorem 2

If we combine Lemmas 18 and 20, we can in polynomial time construct a dictionary over d keys supporting searches in time $S(d)$, where

$$S(n) = O\left(\min\left(1 + \frac{\log n}{\log W}, \log\log U\right)\right) \tag{3}$$

Furthermore, balancing the two parts of the min-expression gives

$$S(n) = O\left(\sqrt{\log n}\right).$$

To get AC^0 bound in Theorem 2, we combine some known results. From Andersson's packed B-trees [1], it follows that if in polynomial time and space, we build a static AC^0 dictionary with membership queries in time t, then in polynomial time and space, we can build a static search structure with operation time $O(\min_i\{it + \log n/i\})$. In addition, Brodnik et.al. [10] have shown that such a static dictionary, using only standard AC^0 operations, can be built with membership queries in time $t = O((\log n)^{1/2+o(1)})$. We get the desired static search time by setting $i = O((\log n)^{1/4+o(1)})$. **This completes the proof of Theorem 2, hence of Corollary 4**.

4.4 Two additional notes on searching

Firstly, we give the first deterministic polynomial-time (in n) algorithm for constructing a linear space static dictionary with $O(1)$ worst-case access cost (cf. perfect hashing).

As mentioned earlier, a linear space data structure that supports member queries (neighbor queries are not supported) in constant time can be constructed at a worst-case cost $O(n^2W)$ without division [30]. We show that the dependency of word size can be removed.

Proposition 22 *A linear space static data structure supporting member queries at a worst case cost of $O(1)$ can be constructed in $O(n^{2+\epsilon})$ worst-case time. Both construction and searching can be done without division.*

Proof: W.l.o.g we assume that $\epsilon < 1/6$.

Since Raman has shown that a perfect hash function can be constructed in $O(n^2W)$ time without division) [30], we are done for $n \geq W^{1/\epsilon}$.

If, on the other hand, $n < W^{1/\epsilon}$, we construct a static tree of fusion tree nodes with degree $O\left(n^{1/3}\right)$. This degree is possible since $\epsilon < 1/6$. The height of this tree is $O(1)$, the cost of constructing a node is $O\left(n^{4/3}\right)$ and the total number of nodes is $O\left(n^{2/3}\right)$. Thus, the total construction cost for the tree is $O(n^2)$.

It remains to show that the space taken by the fusion tree nodes is $O(n)$. According to Fredman and Willard, a fusion tree node of degree d requires $\Theta(d^2)$ space. This space is occupied by a lookup table where each entry contains a rank between 0 and d. A space of $\Theta(d^2)$ is small enough for the original fusion tree as well as for our exponential search tree. However, in order to prove this proposition, we need to reduce the space taken by a fusion tree node from $\Theta(d^2)$ to $\Theta(d)$. Fortunately, this reduction is straightforward. We note that a number between 0 and d can be stored in $\log d$ bits. Thus, since $d < W^{1/6}$, the total number of bits occupied by the lookup table is $O(d^2 \log d) = O(W)$. This packing of numbers is done cost efficiently by standard techniques.

We conclude that instead of $\Theta(d^2)$, the space taken by the lookup table in a fusion tree node is $O(1)$ ($O(d)$ would have been good enough). Therefore, the space occupied by a fusion

tree node can be made linear in its degree, which implies that the entire data structure uses linear space. ∎

Secondly, we show how to adapt our data structure according to a natural measure. An indication of how "hard" it is to search for a key is how large part of it must be read in order to distinguish it from the other keys. We say that this part is the key's *distinguishing prefix*. (In Section 4.2 we used the term longest matching prefix for essentially the same entity.) For W-bit keys, the longest possible distinguishing prefix is of length W. Typically, if the input is nicely distributed, the average length of the distinguishing prefixes is $O(\log n)$.

As stated in Proposition 23, we can search faster when a key has a short distinguishing prefix.

Proposition 23 *There exist a linear-space data structure for which the worst-case cost of a search and the amortized cost of an update is $O(\log b \log\log n)$ where $b \leq W$ is the length of the query key's distinguishing prefix, i.e. the prefix that needs to be inspected in order to distinguish it from each of the other stored keys.*

Proof: We use exactly the same data structure as in Corollary 21, with the same restructuring cost of $O(\log\log n)$ per update. The only difference is that we change the search algorithm from the proof of Lemma 20. Applying an idea of Chen and Reif [11], we replace the binary search for the longest matching (distinguishing) prefix by an exponential-and-binary search. Then, at each node in the exponential search tree, the search cost will decrease from $O(\log W)$ to $O(\log b)$ for a key with a distinguishing prefix of length b. ∎

5 Constant time finger updates

In this section, we will generally show how to reduce the finger update time of $O(\log\log n)$ from Lemma 12 to a constant. The $O(\log\log n)$ bound stems from the fact that when we insert or delete a leaf, we use a local update step for each level above the leaf. Now, however, we only want to use a constant number of local update steps in connection with each leaf update. The price is that we have less local update steps available for the postprocessing of joins and splits. More precisely, we will prove the following analogue to the general balancing in Theorem 8:

Theorem 24 *Given a number series $n_1, n_2, \ldots$, with $n_1 \geq 84$, $23n_i < n_{i+1} < n_i^2$ for $i \geq 1$, we can schedule split and joins to maintain a multiway tree where each non-root node v on height $i > 0$ has weight between $n_i/4$ and n_i. A root node on height $h > 0$ has weight at most n_h and at least 2 children, so the root has weight at least $n_{h-1}/2$. The schedule gives the following properties:*

(i) When a leaf v is inserted or deleted, the schedule uses a constant number of local update steps. The additional time used by the schedule is constant.

(ii) For each split or join at level $i > 1$ the schedule ensures that we have at least $\sqrt{n_i}$ local update steps available for postprocessing, including one in connection with the split or join itself. For level 1, we have n_1 local update steps for the postprocessing.

As we shall see in Section 6, the $\sqrt{n_i}$ local update steps suffice for the maintenance of S-structures.

It is important to note that the number series of Theorem 24 is more restrictive than that of Theorem 8, hence that Theorem 8 works for any number series from Theorem 24.

We will use essentially the same schedule for join and splits as in the proof of Theorem 8. The schedule of Theorem 8 ascertains that we during the postprocessing of a split or join on level i have at least $n_i/84$ leaf updates below the involved nodes or their neighbors.

Theorem 8 performs a local update step on the split or join postprocessing in connection with each of these leaf updates. For Theorem 24, we need to be far more economical with the local update steps. On the other hand, we only need to make $\sqrt{n_i} \ll n_i/84$ local update steps available for the postprocessing.

As for Theorem 8, we have

Lemma 25 *Given Theorem 24, the following property can be added to the theorem: the parent of any node can be computed in constant time.*

Proof: We use exactly the same construction as for Lemma 9. The critical point is that we for the postprocessing have a number of updates which is proportional to the number of children of a node. This is trivially the case for level 1, and for higher levels i, the number of children is at most $n_i/(n_{i-1}/4) = O(\sqrt{n_i})$. ■

As in the proof of Theorem 8, we will actually use the parent computation of Lemma 25 in the proof of Theorem 24. As argued at the end of Section 3.3.2 this does not lead to a circularity.

Level 1 is exceptional, in that we need n_1 local update steps for the split and join postprocessing. This is trivially obtained if we with each leaf update make 84 local updates on any postprocessing involving or tied to the parent. For any other level $i > 1$, we need $\sqrt{n_i}$ local update steps, which is obtained below using a combination of techniques. We will use a tabulation technique for the lower levels of the exponential search tree, and a scheduling idea of Levcopoulos and Overmars [27] for the upper levels.

5.1 Constant update cost for small trees on the lower levels

In this subsection, we will consider small trees induced by lower levels of the multiway tree from Theorem 24.

One possibility for obtaining constant update cost for search structures containing a few keys would have been to use atomic heaps [19]. However, here we aim at a solution using only AC^0 operations. We will use tabulation. A tabulation technique for finger updates was also used by Dietz and Raman [14]. They achieved constant finger update time and $O(\log q)$ finger search time, for q intermediate keys, in the comparison based model of computation. However, their approach has a lower bound of $\Omega(\log q/\log\log q)$ as it involves ranks [20], and would prevent us from obtaining our $O(\sqrt{\log q/\log\log q})$ bound. Furthermore, our target is the general schedule for multiway trees in Theorem 24 which is not restricted to search applications.

Below we present a schedule satisfying the conditions of Theorem 24 except that we need tables for an efficient implementation.

5.1.1 A system of marking and unmarking

Our basic goal is to do only a constant number of local update steps in connection with each leaf update. More precisely, when we insert or delete a leaf u, we will do 600 local update steps. The place for these local update steps is determined based on a system of marking and unmarking nodes. To place a local update step from a leaf u, we find its nearest unmarked ancestor v. We then unmark all nodes on the path from u to v and mark v. The local update step will be used by any postprocessing involving v or a neighbor that v is tied to.

Lemma 26 *Let v be a level i node. If there are m markings from leaves below v, then v gets marked at least $(m - 2n_i)/2^{i+1}$ times.*

Proof: We use a potential function argument. The potential of a marked node is 0 while the potential of an unmarked node on level i is 2^i. The sub-potential of v is the sum of the potential of all nodes descending from or equal to v. Then, if an update below v does not unmark v, it decreases the sub-potential by 1. On the other hand, if we unmark v, we also unmark all nodes on a path from a leaf to v, so we increase the potential by $2^{i+1} - 1$. When nodes are joined and split, the involved nodes are all marked so the potential is not increased. The potential is always non-negative. Further, its maximum is achieved if all nodes are unmarked. Since all nodes have degree at least 4, if all nodes are unmarked, the leaves carry more than half of the potential. On the other hand, the number of leaves below v is at most n_i, so the maximum potential is less than $2n_i$. It follows that the number of times v gets unmarked is more than $(m - 2n_i)/2^{i+1}$. Hence v gets marked at least $(m - 2n_i)/2^{i+1}$ times during m local updates from leaves below v. ■

5.1.2 A restricted weight balancing game

Lemma 26 is only useful if we have many markings below a single node. The markings stem from leaf updates below that node. For the postprocessing of a split or join, the schedule from the previous section guaranteed that we would have $n_i/84$ leaf updates below the involved nodes and their tied neighbors. However, the tied neighbors could change many times during the postprocessing, and then we cannot guarantee many leaf updates below any single node.

To guarantee many leaf updates below a single node, we make a slight restriction of the rules of the weight balancing game from Section 3.3.1. Currently, the b step postprocessing of a join or split may result from any updates to the involved weights and their neighbors. We now require that the b steps either all come from updates to the involved weights, or all come from updates to a single tied neighbor. In particular this means that we do not count steps from a tied neighbor if the neighbor gets untied before the postprocessing has finished. The lemma below states that the this restriction does not affect the validity of Proposition 15.

Lemma 27 *Proposition 15 remains valid for the restricted weight balancing game where the b postprocessing steps for a join or split have to be provided either from updates to the involved weights or from updates to a single tied neighbor.*

Proof: The protocol is identical to that of Proposition 15. The proof is also unchanged. The argument that free weights do not change too much only used that if a free weight v is tied to a weight involved in a postprocessing, then that postprocessing will finish after at most b updates to v. The argument that weights involved in a postprocessing do not change too much only used that the postprocessing finishes after at most b updates to the involved weights. Both of these properties are preserved in the modified game. ■

Since Proposition 15 remains true, we can use this restricted weight balancing game for the scheduling of join and splits in Theorem 8. This allows us to pin the $n_i/84$ leaf updates below a level i split or join postprocessing as follows.

Lemma 28 *For each split or join at level i, our schedule ensures that we have at least $n_i/84$ leaf updates either below the involved nodes, or below a single tied neighbor.*

The next lemma concludes that our revised schedule satisfies Theorem 24 except that we lack an efficient implementation.

Lemma 29 *We get at least $n_i/2^i > \sqrt{n_i}$ local update steps for a split or join postprocessing on level i.*

Proof: Consider a split or join postprocessing on level i. From Lemma 28 we known that we have at least $n_i/84$ leaf updates either below the involved nodes, or below a single tied neighbor. Each leaf update results in 600 node markings, and each time we mark one of the above nodes, we get a local update step for the postprocessing.

Consider first the simple case where we get $n_i/84$ leaf updates below a single node v which is either a merged node, or the single tied neighbor. We then get $600 \cdot n_i/84$ markings from below v. By Lemma 26, this leads to $(600 \cdot n_i/84 - 2n_i)/2^{i+1} > n_i/2^i$ markings of v.

Suppose instead that we get $n_i/84$ leaf updates below two nodes v_1 and v_2 from a split. Let m_1 and m_2 be the markings from below v_1 and v_2, respectively. Then $m_1 + m_2 \geq 600 \cdot n_i/84$. By Lemma 26, the total number of markings of v_1 and v_2 is at least

$$(m_1 - 2n_i)/2^{i+1} + (m_2 - 2n_i)/2^{i+1} = (m_1 + m_2 - 4n_i)/2^{i+1} \geq (600 \cdot n_i/84 - 4n_i)/2^{i+1} > n_i/2^i.$$

Since each of these markings leads to a local update step on the postprocessing, we get at least $n_i/2^i$ local update steps for the postprocessing. Finally $n_i/2^i > \sqrt{n_i}$ because $n_i > 23^i$, as stated in Theorem 24. ■

5.1.3 A tabulation based implementation

The above schedule, with the marking and unmarking of nodes to determine the local update steps, could easily be implemented in time proportional to the height of the tree, which is $O(\log \log n)$. However, to get down to constant time, we will use a special compact representation of small trees with up to m nodes where $m = O(\sqrt{\log n})$. Here we think of n as a fixed capacity for the total number of stored keys. (While the number of actual keys change by a factor of 2, we can build a data structure with new capacity in the background.)

Consider an exponential search tree E with at most m nodes, i.e. E is one of the small trees at the bottom of the main tree. With every node, we associate a unique index below m, which is given to the node when it is first created by a split. Indices are recycled when nodes disappear in connection with joins. We will have a table of size m that maps indices into the nodes in E. Conversely, with each node in E, we will store its index. In connection with an update, tables will help us find the index to the node to be marked, and then the table give us the corresponding node.

Together with the tree E, we store a bitstring τ_E representing the topology of E. More precisely, τ_E represents the depth first search traversal of E where 1 means go down and 0 means go up. Hence, τ_E has length $2m - 2$. Moreover, we have a table μ_E that maps depth first search numbers of nodes into indices. Also, we have a table γ_E that for every node tells if it is marked. We call $\alpha_E = (\tau_E, \mu_E, \gamma_E)$ the signature of E. Note that we have $\leq 2^{2m} \times m^m \times 2^m \times O(m) = m^{O(m)}$ different signatures.

To facilitate updates in trees, we use one global transformation table. For each of the signatures, we tabulate what to do in connection with each possible leaf update. More precisely, for each possible leaf delete, the transformation table takes a signature of the tree and the index of the leaf to be deleted and produces the signature of the tree without the leaf. Thus when deleting a leaf, we first find its associated index so that we can use the table to look up the new signature. Similarly, for a leaf insert, we know the index of a preceding sibling, or the parent if the leaf is to be a first child. The transformation table produces both the new signature and the index of the new leaf. This index is stored with the new leaf. Also, the leaf is stored with the index in the table mapping indices to nodes.

For the local updates, the transformation table takes a signature and the index of a leaf to do the local update from. The table produces the index of the node to be marked, hence at which to do a local update. If a split or join is to be done, the table tells the indices of the involved nodes. For a split, this includes the child at which to split the children sequence. Also, it includes the index of the new node. Finally, the transformation table produces the signature of the resulting tree.

The entire transformation table can easily be constructed in $m^{O(m)} = o(n)$ time and space. Furthermore, each signature occupies $O(\log n)$ bits, so we can update the signature of a tree in constant time.

In Figure 5.4 we give an illustration of the construction with small trees, represented as signatures. The upper part shows the main parts of the entire data structure; a large exponential search tree will small tress at the bottom, one transformation table taking care of updates in all of the bottom trees. The lower part illustrates the structure of a signature.

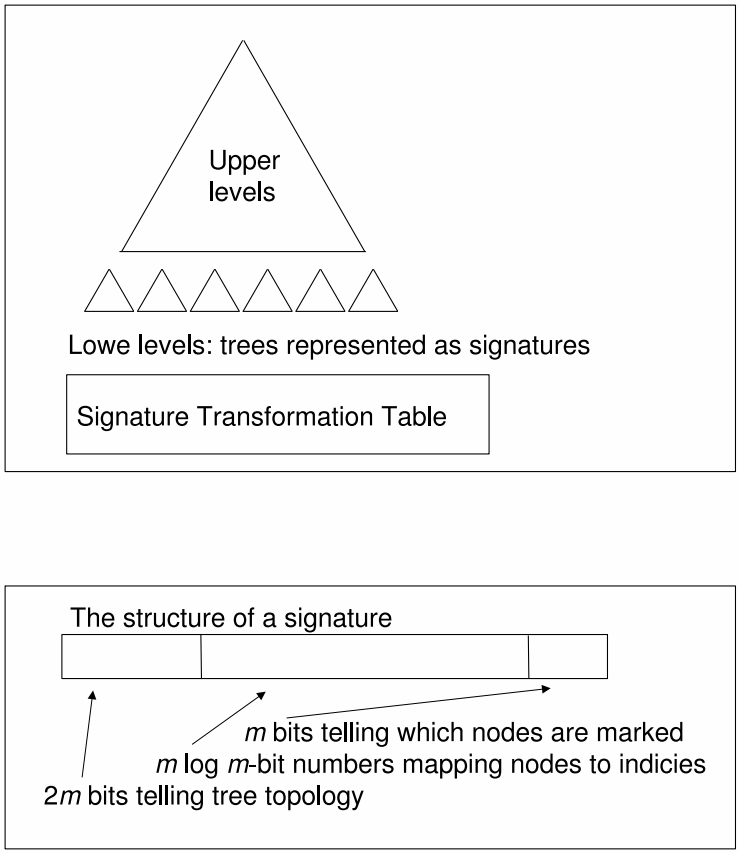

Figure 5.4: Illustration of compact representation of small trees.

Let a be such that $n_a \leq \sqrt{\log n} < n_{a+1}$ and set $m = n_a$. We may assume that $a > 1$, for otherwise n is a constant, and then the whole problem is trivial. We are going to use the above tabulation to deal with levels $0, ..., a$ of the multiway tree of Theorem 24. (Note that if $n_1 > \sqrt{\log n}$, $a = 0$, and the construction with small trees is not needed.) With each of the level a nodes, we store the signature of the descending subtree as well as the table mapping indices to nodes. Also, with each leaf, we store an ancestor pointer to its level a ancestor. Then, when a leaf is added, it copies the ancestor pointer of one of its siblings. Via these ancestor pointers, we get the signature of the tree that is being updated.

A new issue that arises is when level a nodes u and v get joined into u. For this case, we temporarily allow indices up to $2m - 1$, and add m to the indices of nodes descending from v. A table takes the signatures of the subtrees of u and v and produce the signature for the joined tree with these new indices. Also, we place a forward pointer from v to u, so that

nodes below v can find their new ancestor in constant time. To get the index of a node, we take its current ancestor pointer. If it points to a node with a forward pointer, we add m to the stored index. Conversely, given an index, if it is not less than m, this tells us that we should use the old table from v, though subtracting m from the index.

During the postprocessing of the join, we will traverse the subtree that descended from v. We move each node w to u, redirecting the ancestor pointers to u and give w a new unique index below m. Such an index exists because the total size of the tree after the join is at most m. The indexing is done using a table that suggests the index and the resulting new signature. The node is then inserted in the table at u mapping indices below m to nodes. Since we use the same general schedule as that in Theorem 8, we know that we have $n_a/84$ updates below the join before the join needs to be completed. In that time, we can make a post traversal of all the at most n_a descendants of the join, assigning new indices and updating parent pointers. We only deal with a constant number of descendants at the time. For the traversal, we can use a depth first traversal, implemented locally as follows. At each point in time, we are at some node w, going up or down. We start going down at that first child of v from when the join was made. If we are going down, we move w to its first child. If we are going up and there is a sibling to the left, we go to that sibling, going down from there. If we are going up and there is no sibling to the left, we go up to the parent. At each node, we check if it has already been moved to u by checking if the ancestor pointer points to u. If we are about to join or split the traversal node w, we first move w away a constant number of steps in the above traversal. This takes constant time, and does not affect the time bounds for join and split.

A level a split of u into u and v is essentially symmetric but simpler in that we do not need to change the indices. In the traversal of the new subtree under v, we only need to redirect the ancestor pointers to v and to build the table mapping indices to nodes in the new subtree.

The traversals for level a join and split postprocessings take constant time for each descending leaf update. In the next subsection, we are going to do corresponding traversals for two other distinguished levels.

Including the new tables for index pairs, all tables are constructed in $m^{O(m)} = o(n)$ time and space. With them, we implement the schedule of Theorem 24 for levels $i = 0, .., a$ using constant time and a constant number of local update steps per leaf update, yet providing at least $\sqrt{n_i}$ local updates for the postprocessing of each join or split.

5.2 The upper levels

We are now going to implement the schedule of Theorem 24 on levels $a+1$ and above. Recall that $n_{a+1} > \sqrt{\log n}$ which we may assume is larger than any concrete constant.

5.2.1 A counter game with small increments and big decrements

To place local updates at levels $a + 1$ and above, we are going to use the following variant of a lemma of Overmars and Levcopoulos [27]:

Lemma 30 *Given p counters, all starting at zero, and an adversary incrementing these counters arbitrarily. Every time the adversary has made q increments, the increments being by one at the time, we subtract q from some largest counter, or set it to zero if it is below q. Then the largest possible counter value is $\Theta(q \log p)$.*

In the original lemma from [27], instead of subtracting q from a largest counter, they split it into two counters of equal size. That does not imply our case, so we need our own proof, which also happens to be much shorter.

Proof: We want to show that the maximal number of counters larger than $2iq$ is at most $p/2^i$. The proof is by induction. Obviously, the statement is true for $i = 0$, so consider $i > 0$. Consider a time t where the number of counters larger than $p/2^i$ is maximized, and let t^- be the last time before t at which the largest counter was $(2i-1)q$.

We consider it one step to add 1 to q counters, and subtract q from a largest counter. Obviously, at the end of the day, we can at most do q better in total.

The basic observation is that between t^- and t, no change can increase the sum of the counter excesses above $(2i-2)q$, for whenever we subtract q it is from a counter which is above $(2i-1)q$. However, at time t^-, by induction, we had only $p/2^{i-1}$ counters above $(2i-2)q$, and each had an excess of at most q. To get to $2iq$, a counter needs twice this excess, and since the total excess can only go down, this can happen for at most half the counters. ■

For the implementation of Lemma 30, we have

Lemma 31 *Spending constant time per counter increment in Lemma 30, the largest counter to be reduced can be found in constant time.*

Proof: We simply maintain a doubly linked sorted list of counter values, and with each value we have a bucket containing the counters with that value. When a counter c is increased from x to $x+1$, we check the value x' after x in the value list. If $x' > x+1$, we insert $x+1$ into the value list with an associated bucket. We now move c to the bucket of $x+1$, removing x if its bucket gets empty. Decrements by one can be handled symmetrically. Thus, when a largest counter a has been picked, during the next k increments, we can decrement a by one. ■

5.2.2 Converting counter decrements to local updates

We are going to use the above counter game in two bands, one on levels $a+1, ..., b$ where b is such that $n_b \leq (\log n)^{\log\log n} < n_{b+1}$, and one levels $b+1$ and up. First, we consider levels $a+1, ..., b$.

To describe the basic idea, for simplicity, we temporarily assume that there are no joins or splits. Set $q = b-a$. For $i = a+1, ..., b$, during $\Omega(n_i)$ leaf updates below a node v on level i, we will get $\Omega(n_i/q)$ local updates at v. Since $n_{i+1} > 19n_i$, $q < \log_{19}(n_b/n_a) < (\log\log n)^2$. On the other hand, $n_i \geq n_{a+1} > \sqrt{\log n}$, so $q = o(\sqrt{n_i})$.

Each level $a+1$ node v has a counter that is incremented every time we have a leaf update below v. In the degenerate case where $a = 0$, we always make a local update at v so as to get enough updates on level 1 as required by Theorem 24. We make an independent schedule for the subtree descending from each level b node u. Once for every q updates below u, we pick a descending level i node with the largest counter, do a local update at v, and subtract q from the counter. During the next $q-1$ leaf updates below u, we follow the path up from v to u, doing a local update at each node on the way.

A largest counter below u is maintained as described in Lemma 31. The number of counters below u is at most $p = n_b/(n_a/4)$, so by Lemma 30, the maximal counter value is $O(q \log p) = O((\log n_b)^2) = O((\log \log n)^4)$.

Now, for $i = a+1, ..., b$, consider a level i node w. The maximal number of counters below w is $n_i/(4n_{a+1})$, so their total value is at most

$$O((n_i/n_{a+1})(\log \log n)^4) = O((n_i/\sqrt{\log n})(\log \log n)^4) = o(n_i).$$

Each update below w adds one to this number. Moreover, we do a local update at w every time we subtract q from one of the descending counters, possibly excluding the very last subtraction if we have not passed w on the path up to u. Consequently, during $r = \Omega(n_i)$ leaf updates below w, the number of local updates at w is at least

$$(r - o(n_i) - q)/q = \Omega(n_i/q) = \omega(\sqrt{n_i}) > \sqrt{n_i}.$$

5.2.3 Maintaining approximate weights

Next, we show how to maintain approximate weights. For the nodes v on level $a+1$, we assume we know the exact weight W_v. For nodes w on levels $i = a+1, ..., b$, we have an approximate weight $\widehat{W}_w$ which is a multiple of q. When the counter of a level $a+1$ node v is picked, we change $\widehat{W}_v$ by $-q$, or $+q$, or leave it unchanged, whatever brings us closest to W_v. As we move up from v to u during the next $q-1$ updates, at each node w, we change $\widehat{W}_v$ accordingly.

We will now argue that for any node w on level $i = a+1, ..., b$, the absolute error in our approximate weight $\widehat{W}_w$ is $o(n_i)$. The error in $\widehat{W}_w$ is at most the sum of the counters below w plus q for each counter, and we have already seen above that this value is $o(n_i)$. It follows that

$$W_w = (1 \pm o(1))\widehat{W}_w.$$

This error is small enough that we can use the approximate weights for the scheduling of split and joins. More precisely, in the analysis, we rounded at various points, and the rounding left room for errors below a constant fraction.

Our approximate weights are multiples of q and change by at most q each time, so in our weight balancing games, we can simply use the approximate weights divided by q.

One slight problem with the approximate weights is that our schedule only guarantees that $\widehat{W}_w \in [n_i/4, n_i]$, but we need $W_w \in [n_i/4, n_i]$. We solve this by a slight change in the parameters of our balancing game from Proposition 15. We increase μ from 21 to 26 but

keep $\Delta = 7$. Then we get weights between $\mu b = 26b$ and $(3\mu + \Delta + 14)b = 99b$. Including the division by q, we will use $b_i = n_i/(100q)$ (for a moment ignoring a rounding error). Then we end up with $\widehat{W}_w \in [\frac{26}{100}n_i, \frac{99}{100}n_i]$. Since $W_w = (1 \pm o(1))\widehat{W}_w$, this implies $W_w \in [n_i/4, n_i]$ even if we include the rounding error setting $b_i = \lfloor n_i/(100q) \rfloor$.

The above change of parameters implies that we need a slightly larger growth in the n_is than we had before. That is why we in Theorem 24 require $n_{i+1} > 23n_i$ whereas we in Theorem 8 only required $n_{i+1} > 19n_i$. More precisely, we now get an uncuttable segment of size $(5\mu + \Delta + 20)b_i = 157b_i$. As described in Section 3.3.2, this has to be smaller than the split error of the next level which is $\Delta b_{i+1} = 7b_{i+1} > 161b_i$, and this increased growth is sufficient.

5.2.4 Implementation details

We are now ready to describe the details of the schedule as nodes get joined and split. From the last subsection, we have ancestor pointers to level a, and via a table we can also get the exact weight. From this, we can easily get ancestor pointers and exact weights on level $a+1$. On level $a + 1$, we can then run the join and split schedule from Section 3.3.2.

For level $i = a + 2, ..., b$, we use the approximate weights both for the nodes and for the children. When we get a local update at a node w, we know that $\widehat{W}_w$ has just been updated and that it equals the sum of the weights of the children, so we do have local consistency in the approximate weights. We then use the new approximate weight in the schedule of Proposition 15 to check if w is to be joined or split or tied to some other join or split postprocessing. The local update step is applied to any join or split postprocessing neighboring w.

Finally, we use the traversal technique from the last subsection to maintain ancestor pointers to level b nodes. This means that we use constant time on level b in connection with each leaf update. In connection with a join or split postprocessing on level b, this time also suffice to join or split the priority queue over counters below the postprocessed nodes. This completes our maintenance of levels $a + 1, ..., b$.

5.2.5 The top levels

For the levels above b, we use the same technique as we did for levels $a + 1, ..., b$. Direct pointers from level a nodes to their level b ancestors are easily maintained. Then we can access the level b ancestor of a leaf update in constant time, so we can maintain exact weights for the level b nodes. One simplification is that we have only one tree induced by levels above b. Consequently, we have only one priority queue over all counters on level b. The numbers, however, are a bit different. This time, the number q' of levels is $\log_{19}(n/n_b) < \log n$. For $i > b$, $n_i > (\log n)^{\log \log n}$, so $q' = o(\sqrt{n_i})$.

We have one priority queue over all counters on level b, of which there are at most $p' = n/(n_{b+1}/4)$, so by Lemma 30, the maximal counter value is $O(q' \log p') = O(\log n(\log \log n)^2)$.

Now, for $i > b$, consider a level i node w. The maximal number of counters below w is

$n_i/(4n_{b+1})$, so their total value is at most

$$O((n_i/n_{b+1}) \log n (\log\log n)^2) = O((n_i/\log n^{\log\log n}) \log n (\log\log n)^2) = o(n_i).$$

With the above changes in numbers, we use the same technique for levels above b as we used for levels $a+1, ..., b$. **This completes the proof of Theorem 24**

Corollary 32 *Given a number series $n_0, n_1, n_2 <, \ldots,$ with $n_0 = 1$, $n_1 \geq 84$, $n_i^2 > n_{i+1} > 23n_i$, we maintain a multiway tree where each node on height i which is neither the root nor a leaf node has weight between $n_i/4$ and n_i. If an S-structure for a node on height i can be built in $O(\sqrt{n_{i-1}})$ time, or $O(n_1)$ time for level 1, we can maintain S-structures for the whole tree in constant time per finger update.*

Proof: We use the same proof as the one we used to prove Corollary 10 from Theorem 8. ■

6 Finger search

For finger searching we have a finger pointing at a key x while searching for another key y, and let q be the number of keys between x and y. W.l.o.g. we assume $y > x$. In its traditional formulation, the idea of finger search is that we should be able to find y quickly if q is small. Here, we also consider another possibility: the search should be fast if $y - x$ is small. Compared with the data structure for plain searching, we need some modifications to support finger search and updates efficiently. The overall goal of this section is to prove the statement of Theorem 5:

> *There is a fully-dynamic deterministic linear space search structure that supports finger updates in constant time, and given a finger to a stored key x, searches a key $y > x$ in time*
>
> $$O\left(\min\left\{\begin{array}{l}\sqrt{\log q/\log\log q} \\ \log\log q \cdot \frac{\log\log(y-x)}{\log\log\log(y-x)} \\ \log\log q + \frac{\log q}{\log W}\end{array}\right\}\right)$$
>
> *where q is the number of stored keys between x and y. If we restrict ourselves to AC^0 operations, we still get a bound of $O((\log q)^{3/4+o(1)})$.*

In the previous section, we showed how to implement the finger updates in constant time. It remains to make the finger searches fast. For simplicity, we will always assume that the fingered key x is smaller than the key y sought for. The other case can be handled by a "mirror" data structure where each key x is replaced by $U - x$, where U is the largest possible key.

The following finger search analog of Theorem 1 is obtained using the same kind of methods as for pointer based finger search structures, i.e. by the use of horizontal links.

Theorem 33 *Suppose a static search structure on d integer keys can be constructed in $O(d^{(k-1)/2})$, $k \geq 2$, time and space so given a finger to a stored key x, we can search a key $y > x$ in time $S(d, y-x)$. We can then construct a dynamic linear space search structure that with n integer keys supports finger updates in time constant time and finger searches in time $T(q, y-x)$ where q is the number of stored keys between x and y and $T(n, y-x) \leq T(n^{1-1/k}, y-x) + O(S(n, y-x))$. Here S is supposed to be non-decreasing in both arguments. The reduction itself uses only standard AC^0 operations.*

Proof: We use an exponential search tree where on each level we have horizontal links between neighboring nodes. It is trivial to modify join and split to leave horizontal pointers between neighboring nodes on the same level.

A level i node has $O(n_i/n_{i-1}) = O(n_i^{1/k})$ children, so, by assumption, its S-structure is built in time $O(n_i^{(k-1)/(2k)}) = O(\sqrt{n_{i-1}})$. Hence we can apply Corollary 32, and maintain S-structures at all nodes in constant time per finger update.

To search for $y > x$, given a finger to x, we first traverse the path up the tree from the leaf containing x. At each level, we examine the current node and its right neighbor until a node v is found that contains y. Here the right neighbor is found in constant time using the horizontal links between neighbors. As we shall see later, the node v has the advantage that its largest possible degree is closely related to q.

Let u be the child of v containing x and let x' be the separator immediately to the right of u. Then, $x \leq x' \leq y$, and if we start our search from x', we will find the child w where y belongs in $S(d, y-x') \leq S(d, y-x)$ time, where d is the degree of v.

We now search down from w for y. At each visited node, the left splitter x' satisfies $x \leq x' \leq y$ so we start our search from the left splitter.

We are now going to argue that the search time is $T(q, y-x) \leq T(q^{1-1/k, y-x}) + O(S(q, y-x))$, as stated in the theorem. Let i be the level of the node v. Let u be the level $i-1$ ancestor of the leaf containing x, and let u' be the right neighbor of u. By definition of v, y does not belong to u', and hence all keys below u' are between x and y. It follows that $q \geq n(u') \geq n_{i-1}/10$. Now, the recursive search bound follows using the argument from the proof of Lemma 13. ∎

Note in the above theorem, that it does not matter whether the static search structure supports efficient finger search in terms of the number d of intermediate keys. For example, the static search bound of $O(\sqrt{\log n / \log\log n})$ from [7] immediately implies a dynamic finger search bound of $O(\sqrt{\log q / \log\log q})$ where q is the number of stored keys between the fingered key x and the sought key y. However, if we want efficiency in terms of $y-x$, we need the following result.

Lemma 34 *A data structure storing a set X of d keys from a universe of size U can be constructed in $d^{O(1)}$ time and space such that given a finger to stored key $x \in X$, we search a key $y > x$ in time $O(\log\log(y-x) / \log\log\log(y-x))$.*

Proof: Beame and Fich [7] have shown that a polynomial space search structure can be constructed with search time $O(\min\{\sqrt{\log n / \log\log n}, \log\log U / \log\log U\})$, where n is the

number of keys and $U = 2^W$ is the size of the universe they are drawn from. As a start, we will have one such structure over our d keys. This gives us a search time of $O(\sqrt{\log d / \log\log d})$. Hence we are done if $\log\log(y-x)/\log\log\log(y-x) = \Omega(\sqrt{\log d/\log\log d})$, and this is the case if $y - x \geq 2^d$.

Now, for each key $x \in X$, and for $i = 0, ..., \log\log d$, we will have a search structure $S_{x,i}$ over the keys in the range $[x, x + 2^{2^{2^i}})$, with search time $O(\log\log 2^{2^{2^i}} / \log\log\log 2^{2^{2^i}}) = O(2^i/i)$. Then to find $y < x + 2^d$, we look in $S_{x,\lceil\log\log\log(y-x)\rceil}$. Now, $2^{2^{2^{\lceil\log\log\log(y-x)\rceil}}} < (y-x)^{\log(y-x)}$, the search time is $O(\log\log(y-x)^{\log(y-x)}/\log\log\log(y-x)^{\log(y-x)}) = O(\log\log(y-x)/\log\log\log(y-x))$.

It should be noted that it is not a problem to find the appropriate $S_{x,i}$. Even if for each x, we store the $S_{x,i}$ as a linked list together with the upper limit value of $x + 2^{2^{2^i}}$, we can get to the appropriate $S_{x,i}$ by starting at $S_{x,0}$ and moving to larger $S_{x,i}$ until $y < x + 2^{2^{2^i}}$. This takes $O(\log\log\log(y-x)) = o(\log\log(y-x)/\log\log\log(y-x))$ steps.

Finally, concerning space and construction time, since we only have $O(\log\log d)$ search structures for each of the d elements in X, polynomiality follows from polynomiality of the search structure of Beame and Fich. ■

Proof of Theorem 5: The result follows directly from the reduction of Theorem 33 together with the static search structures in Theorem 2, Theorem 3, and Lemma 34. ■

7 String searching

In this section, we prove Theorem 6:

> *For the dynamic string searching problem, if the longest common prefix between a key x and the other stored keys has ℓ words, we can insert, delete, and search x in $O(\ell + \sqrt{\log n/\log\log n})$ time, where n is the current number of keys. In addition to the stored keys themselves, our space bound is $O(n)$.*

7.1 The standard trie solution

As a basic component, we use a standard trie over the strings where the characters are 1-word integers [25, §III]. For now, we use space proportional to the total size of the strings. For technical reasons, we assume that each string ends with a special character $\perp$, hence that no string is a prefix of any other string. A trie over a set S of strings is the rooted tree whose nodes identify prefixes of strings in S. For a string α and a 1-word character a, the node αa has parent α and is labeled a. The root is not labeled, so αa is the labels encountered on the path from the root to the node αa. Our trie is ordered in the sense that the children of a node are ordered according to their labels. We use standard path (Patricia) compression, so paths of unary nodes are stored implicitly by pointers to the stored strings. Hence the trie data structure is really concentrated on the $O(n)$ branching nodes. At a branching node α, the children have the same labels as in the explicit trie, but a label a brings us to a child

β with prefix αa. Here β is the shortest branching node with αa as a prefix. The technical issues involved in the path compression are standard, and will mostly be ignored below.

By storing appropriate pointers, the problem of searching a string x among the stored strings S reduces to (1) finding the longest common prefix α between x and the strings in S, and (2) searching the next 1-word character of x among the labels of the children of the trie node α. In a static implementation, we would use a dictionary in each node, which would allow us to spend constant time at each visited node during step (1). Then, by keeping a search structure from Corollary 4 at each branching node, we perform step (2) in $O(\sqrt{\log n / \log\log n})$ time, which is fine.

However, in a dynamic setting we cannot use dictionaries in each node over all children since we cannot update linear spaced dictionaries efficiently in the worst case. Instead, we will sometimes allow step (1) to spend more than constant time in each visited node. This is fine as long as the total time spent in step (1) does not exceed the total bound aimed at.

7.2 Efficient traversal down a trie

Consider a trie node α. Our new idea is to only use the constant time dictionaries for some of the children of a node, called the dictionary children. The dictionary children will have many leaf descendants so they are guaranteed to remain for a long time. We can therefore afford to store them in a constant time dictionary which is rebuilt slowly in the background. In addition, α will have a dynamic search structure from Corollary 4 over the labels of all its children, including the dictionary children. Now suppose a search arrives at α and the next character is a. Then we first consult the dictionary. If a is in the dictionary, we get to the child labeled a in constant time, just as in the static step (1) above. We call this case (1a). If a is not in the dictionary, we use the search structure at α. If the label a is found by the search structure, it brings us to the child labeled a. We call this case (1b). Otherwise there was no child of α labeled a. We call this case (2). In case (2) the search structure places the query string between the children of α as in the static step (2) which marks the end of the search.

If we ignore case (1b), searching a string down the trie takes $O(\ell + \sqrt{\log n / \log\log n})$ total time, just as in the static case. The total time in case (1b) will be bounded exploiting that it takes us to a non-dictionary child with much fewer leaf descendants, as defined more precisely below.

We will use Proposition 22 to construct the dictionary over the dictionary children. This dictionary uses linear space and is constructed in $O(n^{k-1})$ time for any fixed $k > 3$.

For a node α with m descending leaves, we will guarantee that any child αb with more than $m^{1-1/k}$ descending leaves is in the dictionary at α. When we search the next label a, we first check the dictionary, and only if a is not found do we check the dynamic search structure from Corollary 4. There are at most m labels in the dynamic search structure, so the search for a takes $O(\sqrt{\log m / \log\log m})$ time. In case (1b) where a is found by the search structure as a label of a child αa, we know that αa has at most $m^{1-1/k}$ descending

leaves. The total cost of case (1b) is therefore bounded by $T(n)$, where

$$\begin{aligned} T(m) &\leq O(\sqrt{\log m/\log\log m}) + T(m^{1-1/k}) \\ &= O(\sqrt{\log m/\log\log m}). \end{aligned}$$

This completes the proof that the total time spent on case (1b) is bounded by $O(\sqrt{\log n/\log\log n})$, hence that the total time needed to search a string in the trie is $O(\sqrt{\log n/\log\log n} + \ell)$.

7.3 Trie updates

We will now show how to update the dictionaries and search structures in our trie. Our target is to match the search time of $O(\sqrt{\log n/\log\log n} + \ell)$. The search path to an update is the path from the root to the leaf inserted or to the leaf to be deleted. The search path itself is found by the above search. The only search structure to be updated is the parent of the leaf, and by Corollary 4, this update takes $O(\sqrt{\log n/\log\log n})$ time. To update the dictionaries, we are going to spend constant time at each node in the search path. This is equivalent to saying that each node α gets to spend constant time during each descending leaf update. If we can satisfy this time constraint, the total update time is bounded by $O(\sqrt{\log n/\log\log n} + \ell)$, as desired.

Suppose the node α has m leaf descendants. Our challenge is to make sure that its dictionary contains the label of any child β with more than $m^{1-1/k}$ leaf descendants. For now we view m as fixed. Nevertheless, we maintain a counter $m(\alpha)$ for the exact number of leaf descendants of α, and use this number to decide if α should be in the dictionary of its parent. Similarly, we know the exact number $m(\beta)$ of leaf descendants of each child β of α. The period of α spans $m^{1-1/k}/4$ leaf updates below α.

To keep track of the dictionary children of α, we maintain an unordered doubly-linked "heavy" list of all children with more than $m^{1-1/k}/2$ descending leaves. When a child passes this threshold, it enters or leaves the heavy list in constant time. In a period, we first scan the heavy list, and then build a dictionary over the labels of the scanned children. When the new dictionary is completed, it replaces the previous one in constant time. A scanned heavy child will continue to have at least $m^{1-1/k}/4$ descending leaves, so we collect at most $m/(m^{1-1/k}/4) = O(m^{1/k})$ children. Constructing a dictionary over the labels takes $O(m^{1/k\cdot(k-1)}) = O(m^{1-1/k})$ time. The whole scan and rebuild can thus be completed spending only constant time during each of the $m^{1-1/k}/4$ updates below α during a period. Then no child can have more than $m^{1-1/k}$ descending leaves without having its label in the current heavy dictionary; for to get $m^{1-1/k}$ descending leaves, it must have had $m^{1-1/k}/2$ descending leaves for at least $m^{1-1/k}/2$ updates. This must include one full period where it got scanned in the heavy list and placed in the new dictionary.

Complicating things a bit, we will now remove the assumption that the number m of leaf descendants of α is constant. Let $m(\alpha)$ denote the exact number of leaf descendants, as maintained by simple counters updated on the search path of an update. We are going to shorten the above period length of α to $m_0^{1-1/k}/5$ where m_0 is the value of $m(\alpha)$ at the

beginning of different kind of period, called the "long" period, of length $m_0/10$. We refer to the old period as the short period. During a long period, we have $m(\alpha) = (1 \pm 1/10)m_0$.

A child β will aim to be in the heavy list if $m(\beta) > \frac{2}{5}m_0^{1-1/k}$. Whenever the search path of an update passes β and $m(\beta)$ changes, we compare it with m_0, entering or withdrawing β from α's heavy list if appropriate. This would work if it was not for changes to $m(\alpha)$. However, during the long period, we run through all the children β of α, and compare $m(\beta)$ with m_0, updating the heavy list if needed. This work of the long period is done in constant time per update below α.

The short period is unchanged in that we first scan the heavy list, and then build a dictionary over the scanned children. However, this time the work is spread over $m_0^{1-1/k}/5$ leaf updates below α. We know that a child β's membership was checked at least in the last period with some $m_0' \geq m_0/(1+1/10)$, and that it then had at least $\frac{2}{5}{m'}_0^{1-1/k} > \frac{2}{5}m_0^{1-1/k}/(1+1/10) = \frac{2}{5}m_0^{1-1/k}/(1+1/10) = \frac{4}{11}m_0^{1-1/k}$ leaf descendants. During the short period, it looses at most $m_0^{1-1/k}/5$ descendants, so it has $\Omega(m_0^{1-1/k})$ leaf descendants throughout the short period. Hence we scan $O(m_0^{1/k})$ children for the new dictionary, which is built in $O(m_0^{1/k\cdot(k-1)}) = O(m_0^{1-1/k})$ time. As above we conclude the whole scan and rebuild can be done spending constant time during each of the $m_0^{1-1/k}/5$ updates below α spanned by a short period.

Finally we have to verify that we honor our guarantee that any child β of α with $m(\beta) > m(\alpha)^{1-1/k}$ is in the current dictionary. We know that $m(\alpha) \geq (1-1/10)m_0$, so we want to show that $m(\beta) \leq \frac{9}{10}m_0$. If β is not in the dictionary, it is because it was not in the heavy list during the last scan of the short period. This means that we had $m(\beta) \leq \frac{2}{5}m_0^{1-1/k}$ last time we checked before that scan. Since a check is performed every time $m(\beta)$ is changed, only m_0 can have changed since then; namely from the m_0' of the previous period. Since $m_0 \leq \frac{9}{10}m_0'$, we conclude that we had $m(\beta) \leq \frac{2}{5}(\frac{10}{9}m_0)^{1-1/k} \leq \frac{4}{9}m_0^{1-1/k}$ when the scan of the last short period passed by. Since then, we had at most $2m_0^{1-1/k}/5$ updates below α, limiting the increase of $m(\beta)$. Hence the maximal value of $m(\beta)$ is $(\frac{4}{9}+\frac{2}{5})m_0^{1-1/k} < \frac{9}{10}m_0^{1-1/k}$, as desired. This completes the proof that we honor our guarantees for the dictionaries spending only constant time per node on the search path, hence that our total update time is $O(\sqrt{\log n/\log\log n} + \ell)$.

7.4 Space

Since both the dictionaries from Proposition 22 and the dynamic search structures from Corollary 4 use linear space, and since a path compressed trie only uses space at the branching nodes, we conclude that our total space bound is $O(n)$. **This completes the proof of Theorem 6**.

8 Other applications of our techniques

In this section we discuss how the techniques presented in this paper have been applied in other contexts.

Variants of exponential search trees have been instrumental in many of the previous strongest results on deterministic linear integer space sorting and priority queues [2, 31, 5, 21]. Here a priority queue is a dynamic set for which we maintain the minimum element. When first introduced by Andersson [2], they provided the then strongest time bounds of $O(\sqrt{\log n})$ for priority queues and $O(n\sqrt{\log n})$ for sorting. As noted by Thorup in [31], we can surpass the $\Omega(\sqrt{\log d/\log\log d})$ lower bound for static polynomial space searching in a set of size d if instead of processing one search at the time, we process a batch of d searches. Thorup got the time per key in the batch down to $O(\log\log d)$. In order to exploit this, Thorup developed an exponential priority queue tree where the update time was bounded by (1), but with $S(n)$ being the per key cost of batched searching. Thus he got priority queues with an update time of $O((\log\log n)^2)$ and hence sorting in $O(n(\log\log n)^2)$ time. Thorup's original construction was amortized, but a worst-case construction was later presented by Andersson and Thorup [5]. More advanced static structures for batched searching where later developed by Han [21] who also increased the batch size to d^2. He then ended up with a priority queue update time $O((\log\log n)(\log\log\log n))$ and sorting in $O(n(\log\log n)(\log\log\log n))$ time. However, exponential search trees are not used in Han's recent deterministic $O(n\log\log n)$ time sorting in linear space [22] or in Thorup's [32] corresponding priority queue with $O(\log\log n)$ update time. Since (1) cannot give bounds below $O(\log\log n)$ per key, it seems that the role of exponential search trees is played out in the context of integer sorting and priority queues.

Bender, Cole, and Raman [8] have used the techniques to derive worst-case efficient cache-oblivious algorithms for several data structure problem. This nicely highlights that the exponential search trees themselves are not restricted to integer domains. It just happens that our applications in this paper are for integers.

Patrascu [28] has very recently used exponential search trees to get the first sublogarithmic query time for planar point location.

Theorem 8 provides a general tool for maintaining balance in multiway trees. These kind of techniques have been used before, but they have never been described in an such a general independent quotable way. By using our theorems, many proofs of dynamization can be simplified, and in particular, we can avoid the standard hand-waving, claiming without proper proof that amortized constructions can be deamortized. The second author [32] has recently used our Proposition 15 in a general reduction from priority queue to sorting, providing a priority queue whose update cost is the per key cost of sorting. Also, he [33] has recently used Theorem 8 in a space efficient solution to dynamic stabbing, i.e., the problem of maintaining a dynamic set of intervals where the query is to find an interval containing a given point. This codes problems like method look-up in object oriented programming and IP classification for firewalls on the Internet. The solution has query time $O(k)$, update time $O(n^{1/k})$, and uses linear space. Previous solutions used space $O(n^{1+1/k})$. The solution does not involve any search structure, so it is important that Theorem 8 has a general format not specialized to search applications.

9 Concluding remarks

We have introduced exponential search trees and used them to obtain optimal bounds for deterministic dynamic searching on the RAM, both for regular searching and for finger and string searching. Some interesting open problems remain.

Bounds in terms of the universe What is the best linear space deterministic dynamic search time measured in terms of the universe size U, that is, when all keys are in $[U]$?

The current $O(\sqrt{\log n/\log\log n})$ bound is only optimal in terms of the number n of keys. From [29] we have a lower bound of $\Omega(\log\log U)$. Using randomized hashing, this lower bound is matched by van Emde Boas' data structure [26, 34, 35]. Can this be derandomized? In Corollary 4, we provided a search time of $O(\log\log n\cdot\frac{\log\log U}{\log\log\log U})$. This bound also holds when n is the number of distinct values, so in terms of U, we get a upper bound of $O(\frac{(\log\log U)^2}{\log\log\log U})$. This upper bound is almost the square of the lower bound.

AC^0 RAM Another interesting open problem is to find the complexity for searching with standard, or even non-standard, AC^0 operations? Andersson et.al. [3], have shown that even if we allow non-standard AC^0 operations, the exact complexity of membership queries is $\Theta(\sqrt{\log n/\log\log n})$. This contrast the situation at the RAM, where we can get down to constant time for membership queries. Interestingly, $\Theta(\sqrt{\log/\log\log n})$ is also the RAM lower bound for searching, so the question is whether or not it is possible to do the $\Theta(\sqrt{\log n/\log\log n})$ searching using AC^0 operations only.

Acknowledgments

We thank a referee from *J. ACM* for many good suggestions for improving the presentation of this paper.

References

[1] A. Andersson. Sublogarithmic searching without multiplications. In *Proc.* 36*th FOCS*, pages 655–663, 1995.

[2] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc.* 37*th FOCS*, pages 135–141, 1996.

[3] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *Proc.* 37*th FOCS*, pages 441–450, 1996.

[4] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.

[5] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32th STOC*, pages 335–342, 2000.

[6] A. Andersson and M. Thorup. Dynamic string searching. In *Proc. 12th SODA*, pages 307–308, 2001.

[7] P. Beame and F. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. System Sci.*, 65(1):38–72, 2002. Announced at STOC'99.

[8] M. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th ICALP*, pages 195–207, 2002.

[9] G. S. Brodal, G. Legogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *J. Comput. System Sci.*, 67(2):381–418, 2003. Announced at STOC'02.

[10] A. Brodnik, P. B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In *Proc. 5th WADS, LNCS 1272*, pages 426–439, 1997.

[11] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proc. 34th FOCS*, pages 104–112, 1993.

[12] L. J. Comrie. The hollerith and powers tabulating machines. *Trans. Office Machinary Users' Assoc., Ltd*, pages 25–37, 1929-30.

[13] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, McGraw-Hill, 2nd edition, 2001. ISBN 0-262-03293-7, 0-07-013151-1.

[14] P.F. Dietz and R. Raman. A constant update time finger search tree. *Inform. Process. Lett.*, 52:147–154, 1994.

[15] A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

[16] L. R. Ford and S. M. Johnson. A tournament problem. *Amer. Math. Monthly*, 66(5):387–389, 1959.

[17] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[18] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.

[19] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994.

[20] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

[21] Y. Han. Improved fast integer sorting in linear space. *Inform. Comput.*, 170(8):81–94, 2001. Announced at STACS'00 and SODA'01.

[22] Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Alg.*, 50(1):96–105, 2004. Announced at STOC'02.

[23] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988.

[24] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973. ISBN 0-201-03803-X.

[25] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984. ISBN 3-540-13302-X.

[26] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Inform. Process. Lett.*, 35(4):183–189, 1990.

[27] M. H. Overmars and C. Levcopoulos. A balanced search tree with O(1) worst-case update time. *Acta Inform.*, 26:269–277, 1988.

[28] M. Pătraşcu. Planar point location in sublogarithmic time. In *Proc. 47th FOCS*, pages 325–332, 2006.

[29] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th STOC*, pages 232–240, 2006.

[30] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proc. 4th ESA, LNCS 1136*, pages 121–137, 1996.

[31] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. 9th SODA*, pages 550–555, 1998.

[32] M. Thorup. Equivalence between priority queues and sorting. In *Proc. 43nd FOCS*, pages 125–134, 2002.

[33] M. Thorup. Space efficient dynamic stabbing with fast queries. In *Proc. 35th STOC*, pages 649–658, 2003.

[34] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6(3):80–82, 1977.

[35] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[36] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Proc. Lett.*, 17:81–84, 1983.

[37] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000. Announced at SODA'92.

[38] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32(3):597–617, 1985.