PREPRINT. Proc. STOC 2000

Tight(er) Worst-case Bounds on Dynamic Searching and Priority Queues

Arne Andersson
Computing Science Department
Information Technology, Uppsala University
Box 311, SE - 751 05 Uppsala, Sweeden
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.edu

ABSTRACT

We introduce a novel technique for converting static polynomial space search structures for ordered sets into fully-dynamic linear space data structures. Based on this we present optimal bounds for dynamic integer searching, including finger search, and exponentially improved bounds for priority queues.

1. INTRODUCTION

New technical contributions

In this article, we study the worst-case complexity of some of the most basic problems in data structures: dynamic integer searching, including finger search, and priority queues. For these problems, we improve the worst-case complexity dramatically into optimal or near-optimal.

The new results include many technical details; some are novel while others are just hard work. For increased readability, we concentrate our technical presentation on the priority queues, since we feel that these require new and fruitful ideas that can be understood without getting into too many technical details.

On a high level, the general idea on how to get a good worst-case behavior in a dynamic data structure is to ensure that the maintained invariants are "locally" defined. A change at one place should only have local effects. For the searching data structure, a redefinition of exponential search trees [2], combined with much care and bookkeeping, gives the desired de-amortization. However, for priority queues the case is much harder. Here, the type of searching, which is really a batched searching, needs some novel, and very different, techniques.

Short background

At STOC'90, Fredman and Willard broke the comparison-based lower bounds for integer sorting using the features of a RAM modeling what we program in imperative programming languages such as C [8]. Their main result was actually a $O(\log n/\log\log n)$ bound for deterministic searching in linear space. They asked the fundamental question: *how fast can we search [integers on a RAM]?* Since then, much effort has been spent on finding the inherent complexity of fundamental searching and sorting problems. Although much progress have been done, big gaps have remained for *worst-case* complexity with linear or just polynomial space bound.

New Results

Searching.

In a recent break-through [4], Beame and Fich gave tight bounds for static search structures with polynomial space, showing matching upper and lower bounds of $\Theta(\sqrt{\log n/\log\log n})$. Based on a general reduction of Andersson [2], called *exponential search trees*, they obtained a fully dynamic deterministic search structure supporting search, insert, and delete in $O(\sqrt{\log n/\log\log n})$ *amortized* time.

In this paper, we improve the exponential search trees to give worst-case bounds. As a result, we provide a linear space data structure supporting search, insert, and delete in $O(\sqrt{\log n/\log\log n})$ worst-case time per operation, which is optimal even for randomized algorithms. The previous best deterministic worst-case bound for fully-dynamic searching was $O(\log n/\log\log n)$ due to Willard [15, Lemma 2]. With randomized algorithms, the best expected time bound for linear or polynomial space was $O(\sqrt{\log n})$ [8].

Our reduction also imply the following worst-case linear space trade-offs between the number n, the word-length ω, and the maximal key $U < 2^\omega$: $O(\min\{\log\log n + \log n/\log\omega,\ \log\log n \cdot \frac{\log\log U}{\log\log\log U}\})$. The last bound should be compared with van Emde Boas' bound of $O(\log\log U)$ [13; 14] that requires randomization (hashing) in order to achieve linear space [9].

Finger search.

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. Also, we have *finger updates*, where for deletions, one has a finger on the key to be deleted, and for insertions, one has a finger to the key

after which the new key is to be inserted.
In the comparison-based model of computation Dietz and Raman [7] have provided optimal bounds, supporting finger searches in $O(\log q)$ time while supporting finger updates in constant time. On the pointer machine, the exact complexity is not settled yet. Brodal [5] has shown how to support finger searches in $O(\log q)$ time and finger inserts in constant time, but his finger deletes takes $O(\log^* n)$ time.
In this paper we present optimal bounds on the RAM; namely $O(\sqrt{\log q/\log\log q})$ for finger search with constant 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.

Priority queues.

Thorup [12] has generalized the above mentioned exponential search trees to convert his static multi-search structures into priority queues, yielding a linear space priority queue supporting find-min in constant time, and insert and delete in $O((\log\log n)^2)$ *amortized* time per operation. Here, again we improve the conversion so as to yield a worst-case bound of $O((\log\log n)^2)$ time per insert or delete. The previous best worst-case bound for polynomial space was $O(\log n/\log\log n)$ [15], even if we ignore general deletions and only support insert and extract-min.
For randomized linear space algorithms, the best bound is $O(\log\log n)$ expected time per insert or extract-min, but for general deletions, the $O(\log\log n)$ bound is only expected amortized [11].

Model of computation.

Our algorithms runs on a RAM, which models what we program in imperative programming languages such as C. The memory is divided into addressable words of length ω. Addresses are themselves contained in words, so $\omega \geq \log n$. 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, and some computational instructions, such as comparisons, addition, and multiplication, for manipulating words in registers. The space complexity 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. 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 2ω bits of precession, and then just perceive the result as an integer. The above examples illustrate how integer ordering can capture many seemingly different orderings that we would naturally be interested in.
Fredman and Willard [8] have 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 $\omega^{O(1)}$-sized constant depth circuit with $O(\omega)$ 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. As it turns out, our $O((\log\log n)^2)$ priority queue uses only standard AC^0 operations. However, 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 AC^0 search structure 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 AC^0 bound of $O((\log n)^{3/4+o(1)})$, thus cracking the $O(\log n)$ bound even in this restricted case.

The exact reductions

Our main results are some general reductions, converting static data structures into dynamic ones. For searching, our 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 $T(n) \leq T(n^{1-1/k}) + O(S(n))$. The reduction itself uses only standard AC^0 operations.*

COROLLARY 2. *There is a fully-dynamic deterministic linear space search structure supporting insert, delete, and searches in worst-case time*

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

where ω is the word length, and $U < 2^\omega$. 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.

PROOF. The first two bounds follows from [4], where Beame and Fich show that for n integer keys, in $d^{O(1)}$ time and space, a static deterministic $O(\min\{\sqrt{\log d/\log\log d}, \log\log U/\log\log\log U\})$-time search structure can be constructed.
The third bound is based on static fusion trees supporting searches in time $O(\frac{\log d}{\log\omega}+1)$ [2; 8].
For the AC^0 bound, we first need to combine a couple of results to get a static data structure. 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. [6] 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 $i = O((\log n)^{1/4+o(1)})$, and the dynamic worst-case time follows from Theorem 1. □

A finger search version of Theorem 1 leads us to the following finger search version of Corollary 2:

THEOREM 3. *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*

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

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 $O((\log q)^{3/4+o(1)})$.

For priority queues, we prove our next main reduction:

THEOREM 4. *Given d keys, suppose a static structure can be constructed in $O(d^{k-1})$, $k > 3$, time and space, supporting any batch of d searches in time $O(d \cdot S(d))$. We can then construct a dynamic priority queue that with n keys supports find-min in constant time and insert and delete in $T(n)$ time where $T(n) \leq T(n^{1-1/k}) + O(S(n))$. The reduction itself uses only standard AC^0 operations*

COROLLARY 5. *There is a dynamic deterministic linear space priority queue supporting find-min in constant time and insert and delete in worst-case time $O((\log\log n)^2)$*

PROOF. Thorup [11; 12] has shown that given a set X of n keys, a static data structure can be constructed in $n^{O(1)}$ time and space so that any batch of n searches in X can be performed in time $O(n \log\log n)$. Now the result follows by application of Theorem 4. □

Contents

In section 2, we briefly review the exponential search trees as presented in [2]. In section 3, we present a worst-case version of exponential search trees. The main step here is the discovery of an alternative definition of the exponential search trees which is locally stable, Subsequently, in section 4, we improve the techniques from [12] to give worst-case bounds for priority queues. This is the main technical challenge in the paper and involves a system of "cascades" for adaptively emptying buffers in an exponential search tree. Finally, in Section 5, we sketch the data structure for finger searching. In this extended abstract, we will skip many technical details, and focus on communicating the main ideas.

2. EXPONENTIAL SEARCH TREES

We will now briefly review exponential search trees, 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.

First, by *searching* a key y in a key set X we mean finding the largest key $x \leq y$ in X. 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 internal node, we store a *min-key* for navigation: if a key arrives at a node, a local search among the min-keys of the children determines which child it belongs under.

In our exponential search trees, the local search at each internal node is performed using a static search structure, called an *S-structure*, at each internal node. 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 min-keys 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 $T(n) = O\left(S\left(O(n^{1/k})\right)\right) + T\left(O(n^{1-1/k})\right)$, which is essentially the time bound we are aiming at.

An exponential search tree over n keys takes linear space. The space of the S-structure at a node of degree d is $O\left(d^{k-1}\right)$, and the total space $C(n)$ is essentially given by

$$\begin{aligned} C(n) &= O((n^{1/k})^{k-1}) + n^{1/k} \cdot C(n^{1-1/k}) \\ \Rightarrow C(n) &= O(n). \end{aligned}$$

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 recursive argument gives that we can construct an exponential search tree over n keys in linear time.

Below, 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.

Balance is maintained in a standard fashion by global and partial rebuilding. 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 merge it with its neighbor. Constructing a new subtree rooted at the node t takes $O(|t|)$ time. In addition, we need to update the S-structure at t's parent v, in order to reflect the adding or removing of a key v's list of child keys. Since v has $\Theta(|v|^{1/k})$ children, the construction time for v's S-structure is $O((|v|^{1/k})^{k-1}) = O(|v|^{1-1/k})$. By definition, this time is $O(|t|)$. 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 size constraints requires that a constant fraction of the keys in a subtree have been inserted and deleted since the subtree was constructed with the right size. 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 time.

This completes our sketchy description of Andersson's exponential search trees [2].

3. WORST-CASE

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 something more in the style of a regular B-tree, where balance is maintained by locally merging and splitting nodes. By locally we mean that the merging and splitting is done just by merging and splitting the children sequences. This type of data structure is for example used by Willard [15] to obtain a worst-case version of fusion trees. The problem with our definition above is that the criteria

for when subtrees are too large or too small are not locally defined. If two subtrees are merged, 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 merged, etc.
Furthermore, since we need to rebuild the S-structure of a merged or split node, merging and splitting has to be scheduled carefully. Even changing the degree of a node by 1, which happens at the parent of merged or split nodes, requires an expensive update in the S-structure. (In Willard's de-amortization of fusion trees [15], an update in a parent was done by an update in an atomic heap, wish only takes constant time.)
In summary, in order to obtain worst-case bounds, we need (i) essential changes in the definition of exponential search trees, making the definition local in the sense that merges or splits do neither affect the parent nor the children of the involved nodes; and (ii) new maintenance algorithms.

The new exponential search trees.

We will use the following bottom-up definition, where the exact constants depend on algorithmic details:

DEFINITION 6. *In an exponential search tree all leaves are on the same depth, and we define the height 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 the number of leaves below v is between $n_i/10$ and n_i, where $n_i = a^{(1-1/k)^{-i}}$ and $a = 60^{k-1} = O(1)$. If the root has height h, we require $n_h \geq n \geq n_{h-1}$, but it is not necessary that $n \geq n_h/10$.*

Note for the n_i that $n_{i-1} = n_i^{1-1/k}$ for $i > 0$, as in the previous recursive definition. Also, our degrees are asymptotically preserved by the weight requirements:

OBSERVATION 7. *The degree of a node on level $i > 0$ can vary by a factor 100 between $(n_i/10)/n_{i-1} = n_i^{1/k}/10$ and $n_i/(n_{i-1}/10) = 10n_i^{1/k}$.*

Since the degree of a node is $\Theta(n_i^{1/k})$ we will sometimes use the notation $d_i = n_i^{1/k}$.
As desired, we have now achieved that merges or splits do neither affect the weight requirements at the parent nor at the children of the involved nodes.
As for the original definition, the following can be shown:

LEMMA 8. *Exponential search trees use linear space, and the search time for an n key exponential search tree is bounded by $T(n) \leq T(n^{1-1/k}) + O(S(n))$.*

Maintaining balance.

We let the split and merge operations, including rebuilding of S-structures, be run as background processes. At each update, we follow a search path down the tree of length $O(\log\log n)$. At each node on the search path, we perform a *local update step*. For each local update step, we spend constant time progressing merge and split processes. Thereby the worst-case restructuring cost per update becomes $O(\log\log n)$. Below we present a sketch of the basic dynamics. It should be noted that we are actually slowing down the merges and splits so as to make sure that the S-structure at the parent can be kept up to date.

General scheduling principle. We allocate $O(n_i)$ time for a merge or split of siblings on level i. This will be divided over $n_i/16$ constant time operations, of which we do one for every local update step at the involved siblings. With a good scheduling, which is too complicated for this extended abstract, this suffices to preserve the weights between $n_i/10$ and n_i, as required by Definition 6.

Rebuilding S-structures. The S-structure at each node is kept up to date by an ongoing cyclic reconstruction. The children being merged and split are causing the need for reconstruction. In a node with degree d, a reconstruction period starts by scanning the current children for their min-keys in $O(d)$ time, then a new S-structure for these keys is constructed in $O(d^{k-1})$ time, the old S-structure is replaced with the new one in constant time, and finally, the old S-structure is destructed in $O(d^{k-1})$ time. By Observation 7, if the node is on level i, $d \leq 10n_i^{1/k}$, which gives a total of $O((n_i^{1/k})^{k-1}) = O(n_{i-1})$ time. We can therefore complete a cycle at a level i node over $n_{i-1}/48$ local update steps, spending constant time per step. This bound will be used in the description of merge and split below.

Merge and split. Consider a merge of two neighboring siblings v and w on level i with parent u. In connection with the local update step at v or w triggering the merge, we remove the min-key from w so that it is not there the next time we reconstruct the S-structure at u. The new node vw takes over the min-key from v, the children lists of v and w are concatenated, and the construction of an S-structure for vw is started. Afterwards, we just wait for the $n_i/16$ local updates to v or w before we declare their merge complete, so as complete on schedule.

An S-structure at u without w's removed min-key will be completed within the next $2 \cdot n_i/48 < n_i/16$ local update steps at u. Since any update below v or w has come through u, such an S-structure will complete before our merge of v and w is declared complete. Also the S-structure for the new merged node, will complete within $n_{i-1}/48 \ll n_i/16$ local update steps.

During the merge, we need to keep a copy of w's min-key, for if the S-structure at u finishes before the S-structure for the merged node vw, then keys arriving to vw have to be distributed between the current S-structures for v and w.

Splitting of a node v is very similar, but with the added complication that we first scan the min-keys and weights at the children to find a min-key that splits the key below v in two approximately equally sized parts.

Above we have ignored some important scheduling problems. In particular, if a node v starts getting light, we would like to merge it with a neighbor, but both neighbors may themselves currently be engaged in merging or splitting processes. In this situation, to avoid that v gets too light, as we delete keys below v, we need to help the neighbors finish their processes, and further request them to not engage in other processes without v. However, nodes in merge and split processes may receive such requests from neighbors on either side, so we need to worry about fairness. All these scheduling issues are postponed to the journal version of this extended abstract. This completes our sketchy proof of Theorem 1.

4. PRIORITY QUEUES

In this section, we de-amortize the exponential search tree based priority queues of Thorup [12]. This de-amortization contains the most intriguing technical problems addressed in this paper. To phrase the problem, we first have to briefly review the $O((\log\log n)^2)$ amortized priority queue from [12]. The $\Omega(\sqrt{\log n/\log\log n})$ lower bound of Beame and Fich [4] for searching, addresses the problem of placing one key among n keys. However, in [12] it is shown that we can preprocess n keys for multi-searching so that we can place n other keys between them in $O(\log\log n)$ time per key.

We use a multi-search structure at each node in the exponential search tree. Therefore, we cannot just take update keys directly down to the leaves. At a node with degree d, we need to accumulate $\Theta(d)$ update keys in a buffer before we send them through the multi-search structure so as to distribute them between the children. As a result, when an update key finally reaches a leaf, we have spent a total of $O((\log\log n)^2)$ time on it.

A special case occurs along the tree's leftmost path. For a priority queue, we need to have the minimum key available at any time. Therefore the smallest keys are not kept in buffers. For each node v on the leftmost path, we avoid putting update keys belonging to the leftmost child in the buffer. When an update key arrives at v, we check in constant time if it can go to the leftmost child of v. If so, it goes there immediately. Otherwise, it is put in the buffer at v.

Turning the ideas above into a priority queue with $O((\log\log n)^2)$ amortized cost was non-trivial. The worst-case solution presented here is quite different in flavor, and the new definition of exponential search trees allows for different techniques to be used. The main challenge in adding buffers to our worst-case exponential search trees is that merges of nodes can cause buffers to overflow, and if they overflow, there is no simple way of getting rid of the overflowing keys. A single update can cause many buffers to overflow. Furthermore, if we just try to send overflowing keys down to the children, then the children's buffers may start overflowing etc. We will resolve this problem by introducing a novel cascading technique.

Preliminary investigations.

Our goal is to mimic the construction from the previous section. First, we make some preliminary definitions and observations bringing the buffer system into the right format.

We specify that a buffer at level i may contain at most $d_i = n_i^{1/k}$ update keys. Each buffer has three rooms: an input room, a multi-search room, and an output room. The multi-search room contains an S-structure capable of placing $d_i/3$ keys between the children in $O(S(d_i))$ time. In each room we have capacity for $d_i/3$ update keys. In a basic *exchange* a key arrives in the input room. Then we do $O(S(d_i))$ work in the multi-search room, and finally, we pull out a key from the output room. The work in the multi-search room is scheduled so that $d_i/3$ key exchanges get a batch of $d_i/3$ keys through the S-structure. Hence, we have a period of $d_i/3$ exchanges, starting with the input room empty, a new batch of $d_i/3$ keys in the multi-search room, and the output room full. After $d_i/3$ key ex changes, the input room is full, the multi-search is completed, and the output room empty. Then, in constant time, the input keys are moved to the multi-search room as a new batch while the processed batch is moved to the output room.

When a key is inserted or deleted, it is first sent as an update key to the root. When an update key arrives at a node, it is exchanged in the local buffer. For the update key coming out of the buffer we know which child it belongs under, and it is then sent to that child. In this way, an update will cause a chain of exchanges on some "search path" down the tree, typically ending with some update key arriving at a leaf where it is inserted or deleted. Each time such a chain passes a node at level i, we have a local update step spending $O(S(d_i))$ time on processing the node's buffer. Hence, the entire cost of the update satisfies $T(n) = T(n^{1-1/k}) + O(S(n))$, as stated in Theorem 4.

For merge, split, and cyclic reconstruction of S-structures, we wish to use the same scheduling technique as for our worst-case search structure in the previous section. An obvious concern is that we cannot upgrade an S-structure while some keys are using it for a multi-search. Thus, when done with a new S-structure, we may have to wait for $d_i/3$ exchanges to finish a batch of keys in the buffer's multi-search room before we can actually replace the old S-structure in the buffer. Also, it takes another $d_i/3$ exchanges before we start getting keys in the output room based on the new S-structure. Thus, for a merge or split on level i, it now takes an extra $2d_{i+1}/3$ local update steps for both the S-structure and the buffer at the parent to integrate a removed or inserted child min-key. However, for $k \geq 3$,

$$d_{i+1} = n_{i+1}^{1/k} = n_i/n_{i+1}^{1-2/k} \leq n_i/n_1^{1/3} < n_i/60$$

which implies that the number of extra steps is small compared to the length of a restructuring period (cf. Section 3). Hence, waiting for the buffers do not cause any essential changes in our bookkeeping.

Splits are implemented as described in the last section, but for merges we run into problems because the merged buffer may contain too many keys.

Main challenge: overflowing buffers

When two nodes are merged, their combined buffer may become too large for the buffer capacity at a single node, and we have a buffer overflow. Unfortunately, any local update step can be part of a merge creating a buffer overflow, so when an update brings us down the tree, we may create an overflow at every single level. From a worst-case perspective, it is very tricky to get rid of these buffer overflows, for if we try to send an overflowing update key further down the tree, then it may again lead to a chain of overflows.

The most obvious solution to this problem might seem to be to redefine the buffer capacities so that merges do not cause buffer overflows. For example, we could let the buffer capacity be linear in the actual degree of a node. Then, when merging two nodes, the degrees and buffer capacities would grow together. However, the overflow problem still pops up: if we merge two nodes, the parent's buffer may suddenly become too large.

Another natural idea would be to let the buffer size at a node v be bounded by a function f of its weight $|v|$ with $f(|v|) = \Theta(|v|^{1/k})$. However, then when merging two nodes we could get a buffer overflow because $2f(|v|) > f(2|v|)$.
Returning to our definition of the capacity as d_i, when two nodes get merged, we may get a buffer overflow of d_i update keys. Our merge is not complete before we have gotten rid of this overflow, and for our accounting to work, this has to happen within the next $n_i/16$ local update steps.

Cascades.

The emptying of buffers are made as carefully scheduled background processes, called *cascades*. From a local perspective, decrementing a buffer means sending one key down to a child. This, in turn, will have the effect that the buffer of that child is incremented. To avoid building up large buffers, we use essentially the following local strategy: each time a node receives one update key from above, it sends two keys further down. In this way, sending a key down to a child generates a cascade where each involved node decrements its buffer. The idea is that a node which is being merged performs a number of cascades in order to decrease its buffer size. In order for this to work, we need to ensure (i) that a node gets time to get rid of two keys before it receives another key from above, and (ii) that the cascades are fast enough to remove all buffer overflow before a merge is completed.

Implementing one cascade.

A cascade is done during a depth-first-style traversal of the tree. Each cascade has a *root*, and a root v has a *cascade pointer* $p(v)$, pointing at the current position of the traversal. Each involved node has a counter, which is 0, 1, or 2. Informally, the counter represents how many keys have been sent down from the node during the current cascade in which the node is involved. When a node u is not the root of a cascade, $p(u)$ is nil, and when u is not involved in a cascade, its counter is 0.
When the cascade starts at v, $p(v) = v$ while v's counter is 0. A *cascade step* at v is performed in the following way. Let u be the node pointed at by $p(v)$. We have two cases:

Case 1. u's counter is 0 or 1.

u has not yet sent down two keys. First we increment the counter. If the buffer's output room is non-empty, we take one key from the buffer, send the key to the proper child and let $p(v)$ point at that child. In this way we proceed downwards. (If the output buffer was empty, we just let $p(v)$ remain pointing at u.)

Case 2. u's counter is 2.

In this case, the cascade is done at u. We set the counter to 0 and let $p(v)$ point at u's parent, in this way backing up the tree. An exception occurs when $u = v$, in which case the entire cascade is done.

LEMMA 9. *Assume that only one cascade takes place at a time and consider a cascade made at node v at height i. Then, during the cascade, at most $3 \cdot 2^i$ update keys have passed v, each one causing a merge step at v.*

PROOF. From the description of a cascade step, it follows that the cascade performs a traversal of some binary part of the tree in a depth-first manner. From each node, the cascade proceeds downwards at most two times, after which the cascade backs up the tree. (The two steps down the tree may be to the same child, so it is not a strict depth-first traversal.)
Since each node in the cascade sends down at most two keys, the total number of visited nodes (including duplicate visits) is at most 2^i. At each node we make three cascade steps, hence the cascade at v is finished within $3 \cdot 2^i$ steps. □

Asynchronous win-win.

Next, we consider the situation with asynchronous ongoing cascades. Assume that when performing a cascade step rooted at v, we find that $p(v)$ points at the root u of some other ongoing cascade. Then, a *win-win-takeover* will take place. We just set $p(v) \leftarrow p(u)$ and set $p(u) \leftarrow$ nil, in this way marking that u is no longer the root of any cascade. We then perform one cascade step at the (new) node pointed at by $p(v)$.
It turns out that this takeover is beneficial for both involved cascades, and we get the following lemma.

LEMMA 10. *Between the occasion when a node v becomes the root of a cascade, and the next occasion when v is not part of a cascade (its own cascade or a takeover), at most $3 \cdot 2^i$ update keys have passed v (causing the same number of merge steps).*

PROOF. We have three cases:

Case 1, v does not take part in any win-win takeover.

The lemma follows immediately from Lemma 9.

Case 2, v's cascade is taken over from a node w above.

Compared with the situation when v is allowed to finish its own cascade, the win-win-takeover will have the following effect on v:

- v will receive one key instead of zero. This does not affect the number of steps to finish the cascade. Since v can get rid of 2 keys during the cascade, the cascade still serves it purpose.
- Since w is an ancestor of v, each update passing v will also pass w. Hence, the takeover can only increase the frequency of the cascade steps.

Case 3, v takes over a cascade at node u.

Then, the a part of the cascade done with root u is a direct saving from the perspective of v. □

LEMMA 11. *Let v be a node at height i whose buffer contains more than d_i keys. Then, after v has taken part in a cascade, either as a root or not, v's buffer size has decreased by at least one.*

PROOF. Since v's buffer contains more than d_i keys, there will be enough keys in the output part of the buffer so that two keys can be sent down by the cascade. On the other hand, at most one key is added from above by the cascade. Non-cascade updates occurring during the cascade will leave the buffer size unchanged. □

From the description above, we note that a cascade step takes $O(S(n_i))$ time.
Finally, we show that the cascading technique has the desired effect of keeping buffers small enough.

LEMMA 12. *After a merge is completed at a node v at height i, v's buffer will have size at most d_i.*

PROOF. The total buffer size of the two merged nodes is at most $2d_i$, so since the merge takes $n_i/16$ steps, we need to get rid of at most d_i keys during $n_i/16$ merge steps.
From Lemmas 10 and 11 we know that the repeated cascading gets rid of one key for every $3 \cdot 2^i$th merge step. Hence we need to show that $n_i/(16 \cdot d_i) > 3 \cdot 2^i$. This can be shown by calculations beyond this extended abstract. □

Clearing the leftmost path

In order to support find-min in constant time, we need to make sure that no buffer on the leftmost path contains an update key belonging to its leftmost child. Therefore, we make the following simple modification of handling update keys that arrive during insert, delete, or cascade along the leftmost path. When an update key arrives at a node on the path, we first check by one comparison if it belongs to the leftmost child. If so, we send it down directly.
The property of having no keys from the leftmost path in buffers is immediately maintained during a split. Problems arises when a node v on the path is merged with its neighbor w. Now, keys in the buffer of the parent u will be illegal if they belonged to w.
We need to get rid of these illegal keys. First, as soon as the merge has started, all update keys arriving at u heading for the new merged node vw will be sent down immediately without entering the buffer at u. In addition, we scan all update keys in the buffer at the parent u from when the merge started, identifying all those that belonged under vw, and then we use cascades to get rid of them. It can be shown that even these extra cascades do not jeopardize the basic scheduling of worst-case exponential search trees.
This completes our sketchy proof of Theorem 4. Details are deferred to the journal version.

5. FINGER SEARCH

Recall that 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.
First, our new version of exponential search trees is well suited for connecting neighboring nodes by horizontal links in the classical way. This allows for the type of tree traversal which is useful for finger searching. Traversing the tree bottom-up from x until we find a node v such that either v or its neighbor contains y, and then searching down again, we immediately get the search complexity in terms of q as stated in Theorem 3.
In order to obtain the bound in terms of $y - x$, we need a more elaborate construction. Essentially, we create a new kind of S-structure where we store several search structures in each node of the tree. For each key in a node, we store a sequence of search structures, representing larger and larger intervals. The sizes of the intervals are growing triply-exponentially, and it can be show that, for a node of degree d, $O(d \log\log d)$ search structures are sufficient (we only need to consider rather small intervals, for if $y - x$ gets too large, the bound in terms of q will suffice). The total space taken by these search structures can be shown to be polynomial in the degree d, and since the exponential search tree can handle any polynomially sized S-structure, the construction works out.
In order to have efficient finger updates, we need to decrease the restructuring cost in exponential search trees from $O(\log\log n)$ to a constant. By changing the parameters, demanding that a static search structure over d keys can be built in time $O(d^{k-2})$ instead of $O(d^{k-1})$, we can decrease the period for merges and splits (cf. Section 3) from $O(n_i)$ to $O((n_i/n_{i-1})^{k-2}) = o(n_i/2^i)$ steps. This implies that we can reduce the number of local update steps needed by a factor 2^i and yet preserve that each step takes constant time.
Implementing this with good worst-case bounds requires careful scheduling in order to find at which nodes to make the local update steps. We have found such a protocol. However, while all our other general problem reduction techniques can be implemented on a pointer machine, the computing of this protocol requires tabulation. If we could find a way to implement our protocol on a pointer machine, we would solve another long standing open problem: how to perform constant time finger updates on a pointer machine. The needed tables are space-consuming. Fortunately, they are small enough for trees of size $\Theta(\log\log n)$. Therefore, we divide the exponential search tree into two layers. At the lower layer, we store small trees and use the tabulated scheduling to achieve constant update cost. At the top level, we have a tree containing $\Theta(n/\log\log n)$ keys. Using a lemma by Levcopoulos and Overmars [10], we can schedule the merging and splitting of small trees so that there is always $\Theta(\log\log n)$ updates between each merge or split causing an update in the top tree. In this way, we will always be able to finish the processing of an update in the top tree before the next split/merge calls for a new insert/delete at the top tree.
This finishes our sketch of finger search and updates.

6. AN OPEN PROBLEM

It is an interesting open problem what is the right 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 member-ship queries is $\Theta(\sqrt{\log / \log\log n})$. 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 potentially, it is possible to do the $\Theta(\sqrt{\log / \log\log n})$ searching using AC^0 operations only.

7. 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] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31^{st} STOC*, pages 295–304, 1999.

[5] G. Brodal. Finger search trees with constant insertion time. In *Proc. 9^{th} SODA*, pages 540–549, 1998.

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

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

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

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

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

[11] M. Thorup. Randomized sorting in $O(n\log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. In *Proc. 8^{th} SODA*, pages 352–359, 1997.

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

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

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

[15] D.E. Willard. Applications of the fusion tree method to computational geometry and searching. In *Proc. 3^{rd} SODA*, pages 386–395, 1992.