Deletion Without Rebalancing in Multiway Search Trees

SIDDHARTHA SEN, Microsoft Research
ROBERT E. TARJAN, Princeton University and Microsoft Research

Some database systems that use a form of B-tree as the underlying data structure do not do rebalancing on deletion. This means that a bad sequence of deletions can create a very unbalanced tree. Yet such databases perform well in practice. Avoidance of rebalancing on deletion has been justified empirically and by average-case analysis, but to our knowledge no worst-case analysis has been done. We do such an analysis. We show that the tree height remains logarithmic in the number of insertions, independent of the number of deletions. Furthermore the amortized time for an insertion or deletion, excluding the search time, is $O(1)$, and nodes are modified by insertions and deletions with a frequency that is exponentially small in their height. The latter results do not hold for standard B-trees. By adding periodic rebuilding of the tree, we obtain a data structure that is theoretically superior to standard B-trees in many ways. Our results suggest that rebalancing on deletion not only is unnecessary but may be harmful.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; H.2.2 [**Database Management**]: Physical Design—*Access methods*

General Terms: Algorithms, Theory, Design

Additional Key Words and Phrases: B-trees, database access methods, exponential potential function, amortized complexity, I/O model

1. INTRODUCTION

Deletion in balanced search trees [Andersson 1993; Bayer 1971; 1972; Bayer and Mc-Creight 1972; Comer 1979; Guibas and Sedgewick 1978; Haeupler et al. 2009; Jannink 1995; Nievergelt and Reingold 1973; Olivié 1982; Sedgewick 2008] is a problematic operation. First, if items are stored in the internal nodes of the tree, deletion can require swapping the item to be deleted with its predecessor or successor: this moves the deletion position to the bottom of the tree, where the deletion can be done easily. Second, the rebalancing needed to keep the height of the tree (and the worst-case search time) logarithmic is more complicated than that needed for insertion. Indeed, the original paper on AVL trees [Adel'son-Vel'skii and Landis 1962] did not discuss deletion, and many textbooks neglect it. Third, if operations on the search tree occur concurrently, as in many database systems that use some form of B-tree as the underlying data structure, the synchronization necessary to do rebalancing on deletion reduces the available concurrency [Gray and Reuter 1993]. Whereas rebalancing *must* be done on insertion

into a B-tree to guarantee correctness (nodes cannot be over-filled), it is optional on deletion, since a B-tree remains a valid search tree even if it has under-filled nodes.

The first problem with deletion can be overcome by storing the items only in the leaves of the tree, storing keys in the internal nodes to support search. Some authors have called such a form of B-tree a *B+-tree* [Comer 1979], but we shall use the term *B-tree* generically. This takes extra space, but the small space penalty may be worth the benefits. The second and third problems can be addressed by avoiding rebalancing on deletion. But then the tree need no longer have height logarithmic in the number of items. Nevertheless, this method has been used successfully in Berkeley DB [Olson et al. 1999; 2000], which uses B-trees with under-filled nodes, and in other database systems.

Avoiding rebalancing on deletion has been justified empirically [Gray and Reuter 1993; Mohan and Levine 1992; Olson et al. 1999; 2000] and by average-case analysis [Johnson and Shasha 1989; 1993], but to our knowledge no one has studied its worst-case efficiency, perhaps because of the assumption that the worst case, however unlikely, is terrible. Here we undertake such a study. Perhaps surprisingly, our results provide ample theoretical justification for avoiding rebalancing on deletion.

One may wonder how this is possible. It is easy to construct an example showing that the tree height can become arbitrarily large, even if there is only one item left in the tree [Jannink 1995]. Furthermore the idea of deletion without rebalancing has also been used in red-black trees, resulting in unforeseen and unfortunate consequences in at least one application [Sen and Tarjan 2010]. Nevertheless, it is still possible that the height could remain logarithmic in the number of insertions. We show that this is indeed the case. We also show that the amortized restructuring time per insertion or deletion is constant, and that nodes are affected by updates with a frequency exponentially small in their heights. The latter results do not hold for standard B-trees, although they do hold for "weak" B-trees, in which the definition of "under-filled" is relaxed, as proposed by Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] and independently by Maier and Salveter [Maier and Salveter 1981]. Zhang and Hsu [Zhang and Hsu 1989] did an average-case analysis of weak B-trees. We take such relaxation to its logical conclusion and allow nodes that are arbitrarily under-filled, even empty. Our amortized analyses use exponential potential functions, a tool that unifies and simplifies the multilevel credit method of Huddleston and Mehlhorn. Our results hold (with different constants) for either bottom-up or top-down splitting on insertion.

Our results suggest that in B-trees rebalancing on deletion is not only unnecessary but possibly harmful, in that it increases update time and complexity while decreasing search time by very little. Our results provide theoretical support for the design decision made in Berkeley DB and other database systems to avoid rebalancing on deletion. In a companion paper [Sen and Tarjan 2010] we present similar results for balanced binary trees. These results require careful design of the deletion method: a natural but bad choice results in the tree height becoming linear in the number of items in the worst case.

The remainder of our paper consists of five sections. In Section 2 we define the *relaxed B-tree*, a form of B-tree in which nodes can be arbitrarily under-filled. We consider trees in which the leaves hold the items and the internal nodes hold keys to guide searches. Our methods give similar results for trees in which the internal nodes hold the items, but deletion becomes more complicated. Relaxed B-trees are essentially those used in Berkeley DB. We describe how to do searches, insertions and deletions in such trees. Insertions use the standard B-tree method of rebalancing by splitting nodes that become too full. Rebalancing can be bottom-up or top-down; we describe both methods. Deletions require only elimination of empty nodes. In Section 3 we analyze

relaxed B-trees. We show that the height, and hence the search time, is $\mathrm{O}(\log_b m/l)$, where m is the total number of insertions and b and l are the maximum number of keys in an internal node and items in a leaf, respectively. This bound is independent of the number of deletions. We also show that an insertion or deletion takes $\mathrm{O}(1)$ amortized time in the I/O model in addition to a search, and that nodes are modified by insertions and deletions with a frequency that is exponentially small in their height. In Section 4 we describe and analyze alternative ways to eliminate empty nodes. In Section 5 we briefly discuss how and when to rebuild the tree. Such rebuilding overcomes two theoretical drawbacks of relaxed B-trees: if the number of deletions approaches the number of insertions, the tree height can become super-logarithmic in the number of items, and the space can become super-linear in the number of items. If rebuilding is done appropriately often, the height remains logarithmic in the number of items, the space remains linear in the number of items, and the amortized rebuilding time is $\mathrm{O}(\epsilon/l)$ per deletion for an arbitrarily small positive ϵ. We conclude in Section 6 with some remarks, including a comparison of our results with those of Huddleston and Mehlhorn and Maier and Salveter.

This paper is a revised, rewritten, and expanded version of a conference paper [Sen and Tarjan 2009]. Our main extension is the new, more eager methods for eliminating empty nodes described in Section 4. These methods further demonstrate the power of exponential potential functions, and provide an additional basis for comparing our work with that of Huddleston and Mehlhorn and Maier and Salveter, which we do in Section 6. This comparison is itself another extension over our conference paper. Our remaining extensions include new bounds on the space utilization of relaxed B-trees and simplifications to existing theorem statements and proofs, both of which appear in Section 3.

2. RELAXED B-TREES

In our discussion of multiway search trees we denote by m, d, n, and h, respectively, the number of insertions, the number of deletions, the current number of items in the tree, and the tree height, which we define as follows: the height of a leaf is zero; the height of an internal node is one plus the maximum of the heights of its children. We assume that the initial tree is empty. In our trees, the structure of internal nodes differs from that of leaves: internal nodes contain keys and pointers to children; leaves contain items (and their keys) but no pointers. The *size* of a node is its number of children if it is internal, its number of items if it is a leaf.

We define our trees using two parameters. A *relaxed B-tree of type* b, ℓ consists of an ordered tree whose leaves all have the same depth, each of whose internal nodes has size between 1 and b inclusive, and each of whose leaves has size between 1 and ℓ inclusive. A node is *full* if it is internal of size b or a leaf of size ℓ. Each item has a distinct key selected from a totally ordered universe. (If item keys are not distinct, we break ties by item identifier.) Increasing key order corresponds to left-to-right leaf order: if leaf x is to the left of leaf y, all items in x have keys smaller than those of all items in y. To facilitate searching, each internal node of size j contains $j-1$ keys in increasing order, alternating with pointers to its j children; if a pointer to node y immediately precedes (follows) key κ, then all items in the subtree rooted at y have key no greater than (greater than) κ. We allow $j = 1$; an internal node of size one and no keys is *empty*. We allow a node to become temporarily over-filled during an insertion: an over-filled internal node is of size $b + 1$ and contains b keys, an over-filled leaf has size $\ell + 1$. We call a leaf empty if it contains no items. We allow a leaf to become temporarily empty during a deletion.

We measure the time of an operation by counting the number of nodes examined or modified. This is equivalent to the search time in the standard I/O complexity

model [Aggarwal and Vitter 1988], in which each I/O operation transmits B units of data between external memory (where the B-tree is stored) and main memory, and any operations on data in main memory are free. All of our results carry over to this model provided three conditions are met: (1) The entire B-tree is stored in external memory, except for the contents of $O(1)$ nodes during certain operations. Our results do not change materially if the root, or the top few levels of the tree, are stored in main memory. (2) The values of b and l are small enough that a node, either a leaf or an internal node, can fit into a single block of external memory, so that reading or writing a node from/to external memory takes one I/O. (3) Main memory can hold the contents of $\Omega(1)$ nodes. Generally we think of b and ℓ as large integer constants, but we obtain meaningful results for any $\ell \geq 1$ and $b \geq 3$ if insertion rebalancing is bottom-up, any $\ell \geq 1$ and $b \geq 4$ if insertion rebalancing is top-down.

To search for the item (if any) with a given key κ in a relaxed B-tree, start at the root and repeat the following *search step* until reaching a leaf: in the current node x, find the smallest key no less than κ and replace x by the child indicated by the pointer immediately preceding κ; if there is no such key, replace x by the rightmost child of x. Upon reaching a leaf, check whether any of its items has the desired key. The time for a search is $h + 1$.

Insertion in a relaxed B-tree is identical to insertion in a standard B-tree with items in the leaves. If the tree is empty, merely create a new leaf containing the item to be inserted. Otherwise, do a search for the key of the item. Upon reaching a leaf, insert the new item into the leaf. If this over-fills the leaf, split it into two leaves, as follows: find its $\lceil (\ell+1)/2 \rceil$-th smallest key κ; create one leaf containing the $\lceil (\ell+1)/2) \rceil = \lfloor \ell/2 \rfloor + 1$ items with keys no less than κ and another leaf containing the remaining $\lfloor (\ell+1)/2 \rfloor = \lceil \ell/2 \rceil$ items; in the parent, replace the pointer to the original leaf by pointers to the two new leaves, separated by a copy of κ. If this over-fills the parent, split it too, but slightly differently: find its $\lceil (b+1)/2 \rceil$-th smallest key κ; put the $\lfloor b/2 \rfloor$ keys smaller than κ and the child pointers preceding κ into a new node; put the $\lceil b/2 \rceil$ keys larger than κ and the child pointers following κ into another new node; in the parent, replace the pointer to the old node by two pointers to the new nodes, separated by κ. That is, promote κ to the parent instead of copying it. After such a split, walk up the path toward the root, splitting over-filled nodes in the same way, until a split does not over-fill a node or the root splits. If the root splits, create a new root containing pointers to the two new nodes created by the split, separated by the promoted or copied key. (The old root could be an internal node or a leaf.)

An alternative way to do an insertion is to split full nodes top-down as the search proceeds, rather than splitting over-filled nodes bottom-up after the search. To do an insertion in this way, do a search on the key of the new item, but if the current node of the search is full, find its $\lceil b/2 \rceil$-th smallest key κ, split the remaining keys and the child pointers between two nodes, containing the keys less than κ and the child pointers preceding κ, and the keys greater than κ and the child pointers following κ, respectively; in the parent, replace the pointer to the old child by pointers to the two new nodes, separated by κ. (The parent cannot be full; if it were, it would have split previously.) If the root splits, create a new root as in the bottom-up method. Continue the search from the appropriate one of the two nodes formed by the split. On reaching a leaf, insert the new item into this leaf and split the leaf if it is over-filled. (Again, the parent cannot be full.)

In order to avoid floors and ceilings in various expressions and to avoid some case analyses, we define $c = \lceil \ell/2 \rceil$ and $c' = \lfloor \ell/2 \rfloor + 1$; these are the sizes of the two nodes into which a leaf splits. We also define $a = \lceil b/2 \rceil$ and $a' = \lfloor b/2 \rfloor + 1$ if splitting is bottom-up, $a = \lfloor b/2 \rfloor$ and $a' = \lceil b/2 \rceil$ if splitting is top-down; these are the sizes of the nodes into which an internal node splits. If ℓ is odd, $c' = c$; otherwise, $c' = c + 1$. If b is
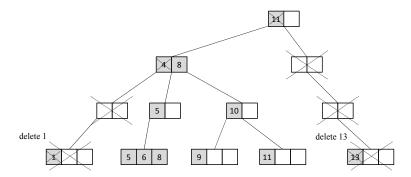
Fig. 1. Deletion in a relaxed B-tree with $b = \ell = 3$. Deleting item 1 and the empty leaf containing it causes deletion of one more empty node and key 4, shown crossed out. Deleting item 13 and the empty leaf containing it causes deletion of two more empty nodes and key 11, leaving the root empty (but not deleted).

odd and splitting is bottom-up, or b is even and splitting is top-down, $a' = a$; otherwise, $a' = a + 1$. We assume $a \geq 2$, which is equivalent to $b \geq 3$ if splitting is bottom-up, $b \geq 4$ if top-down.

To delete an item from a relaxed B-tree, find the leaf containing it and delete the item from the node. If the leaf is now empty, delete it, as well as the pointer from its parent and one of the keys next to this pointer (either key will do if there are two). If before the deletion the parent was empty, delete the parent as well, and walk back up the path toward the root deleting each empty node along with the pointer to it until reaching a non-empty node or the root. If the node reached is non-empty, delete from it a key that is missing a neighboring pointer (such a missing pointer previously pointed to a now-deleted empty child); otherwise (the node reached is the root, and it is empty), delete the node: the entire tree is now empty. (See Figure 1.)

The insertion methods we have described are the standard bottom-up and top-down insertion methods for B-trees with items in the leaves, but our deletion method is not standard. The standard deletion method refills each "under-filled" node by fusing it with a sibling (the inverse of splitting) or borrowing one or more keys or items from a sibling. (Borrowing is equivalent to fusing and then resplitting, possibly in an unbalanced way.) In standard B-trees a leaf or internal node is defined to be under-filled if its size is less than c or a, respectively. With this definition, deletion with refilling guarantees that the tree height is at most $\log_a(n/c)+1$. Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] and independently Maier and Salveter [Maier and Salveter 1981] relaxed the definition of "under-filled," but (crucially) they did not allow empty nodes. They obtained improved amortized bounds for updates, but their approach does not simplify the deletion algorithm, and it increases the worst-case tree height relative to standard B-trees. Allowing empty nodes considerably simplifies deletion, since no refilling is necessary. It also gives good theoretical bounds, as we shall see. We compare our bounds with those of Huddleston and Mehlhorn and Maier and Salveter in Section 6, after we have derived them.

We conclude this section by discussing the implementation of restructuring. The restructuring process needs access to the affected nodes on the search path. There are several ways to provide such access, as we have discussed previously [Haeupler et al. 2014]. One is to add parent pointers to the tree. This uses $O(1)$ extra space per node, but it increases the cost of splitting an internal node because the parent pointers of at least half of its children must be updated. Since each child node must be accessed to update its parent pointer, this increases the restructuring cost of insertions by a factor

of $O(a)$. We conclude that maintaining parent pointers in a B-tree is generally a bad idea.

Instead of storing parent pointers, we can store the search path, either as a stack of pointers in main memory (if main memory has space to hold $h + 1$ pointers to nodes), or by temporarily reversing child pointers along the search path (making them into parent pointers), or by storing temporary parent pointers in the nodes along the search path (if the nodes have space for an extra pointer).

A third approach is to maintain a *safe node* during the search. This node is the topmost node that will be affected by restructuring. Metzger [Metzger 1975] and Samadi [Samadi 1976] used safe nodes to limit the amount of locking in a concurrent B-tree. Assume that all accesses proceed from the root, so that locking a node x prevents access by other processes to the entire subtree rooted at x. As an insertion search proceeds, it needs to maintain a lock only on the bottommost non-full node, which is the safe node. When the search encounters a new non-full node x, it locks x and unlocks the old safe node: any node splitting caused by the insertion will not propagate above x. A similar idea applies to deletions, except the safe node is the bottommost non-empty node. In either an insertion or a deletion, once the search reaches the bottom of the tree, we do appropriately modified restructuring steps top-down starting from the safe node. This method needs only $O(1)$ extra space, but it incurs additional overhead during the search and during the restructuring, to maintain the safe node and to determine the next node on the search path, respectively. Its advantages are that it can avoid the need for parent pointers or a stack to do restructuring and it provides the minimum context needed for locking if searches and updates are concurrent (during an insertion or deletion, lock each new safe node and unlock the old one). For insertions, one can reduce the size of this context to $O(1)$ by splitting full nodes proactively, as we have described above. (Indeed, the top-down insertion method we described advances the safe node in every search step, avoiding the need for a safe node altogether.) For deletions, reducing the context to $O(1)$ requires a more complicated method for eliminating empty nodes, as we discuss in Section 4.

3. ANALYSIS OF RELAXED B-TREES

To estimate the cost of insertions, we bound the number of splits as a function of the height: in addition to the search, an insertion takes $O(1)$ time plus $O(1)$ time per split. To estimate the cost of deletions, we bound the number of nodes deleted as a function of the height: in addition to the search, a deletion takes $O(1)$ time plus $O(1)$ time per node deleted.

We use the potential method of amortized analysis [Tarjan 1985]. To each state of the data structure we assign a non-negative *potential*, zero for an empty structure. We define the *amortized cost* of an operation to be its actual cost plus the net increase in potential it causes. Then for any sequence of operations starting with an empty structure, the sum of the actual costs is at most the sum of the amortized costs.

We use exponential potential functions similar to those we have used to analyze balanced binary trees [Haeupler et al. 2009; Sen and Tarjan 2010; Haeupler et al. 2014]. Our method unifies and simplifies the multilevel credit method of Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982]. Each node has a non-negative potential that is a function of its height and its size. The potential of a tree is the sum of the potentials of its nodes.

A nice feature of deletion without refilling is that it does not increase any node size. This allows us to analyze insertions and deletions separately. To bound the tree height and the cost of insertions, we define the potential of a leaf of size j to be $\max\{0, j - c'\}/c$ and that of an internal node of size j and height h to be $\max\{0, j - a'\}a^{h-1}$. (Since a'

depends on whether splitting is bottom-up or top-down, so does the potential of an internal node.)

A deletion cannot increase the potential. Ignoring the effect of splits, an insertion increases the potential by at most $1/c$, by adding one item to a leaf. Suppose an over-filled leaf splits. Before the split, the leaf contains $c + c'$ items and potential 1. The split produces two new leaves of sizes c and c', respectively, both of potential zero, and increases the potential of the parent by at most 1. Thus the split does not increase the potential.

A similar analysis shows that a split of a node of height $h > 0$ does not increase the potential. Before the split, the node has size $a + a'$ and potential a^h; the split produces two nodes of sizes a and a', respectively, both of potential zero, and increases the potential of the parent by at most a^h, for a net potential increase of at most zero.

There is one situation in which the potential drops drastically: a split of the root. The new root produced by the split has size 2 and potential zero, since $a' \geq 2$. If the new root has height h, the split that created it decreases the potential by a^{h-1}. Since the potential increases by at most $1/c$ per insertion, for a total of m/c over all insertions, the tree height h satisfies $a^{h-1} \leq m/c$. This gives the following theorem:

THEOREM 3.1. *A relaxed B-tree has height at most* $\log_a(m/c) + 1$.

This is our most important theorem. The bound in Theorem 3.1 is the same as that of standard B-trees except that m, the number of insertions, replaces n, the current number of items in the tree. As long as the number of insertions is linear in the number of items (only a fixed fraction less than one of the items have been deleted), the height of a relaxed B-tree is within an additive constant of that of a standard B-tree built by the same sequence of insertions and deletions; as long as the number of insertions is polynomial in the number of items, the height is within a constant factor. If the number of insertions becomes huge compared to the number of items, one can rebuild the tree to reduce its height, either all at once or incrementally. We discuss rebuilding strategies in Section 5.

By truncating the potential function, we obtain good bounds on the number of nodes affected by insertions. For any fixed height h, let the potential function be as defined above for nodes of height at most h, zero for nodes of height greater than h. Then every split of a node of height h reduces the potential by a^h, giving the following theorem:

THEOREM 3.2. *In a relaxed B-tree, the number of splits of nodes of height h is at most* $m/(ca^h)$.

COROLLARY 3.3. *The total time for all restructuring during insertions is* $\mathrm{O}(m/c)$, *or* $\mathrm{O}(1/c)$ *amortized per insertion.*

PROOF. Summing the bound in Theorem 3.2 from $h = 0$ to infinity gives an upper bound of $(m/c)(a/(a - 1))$ on the total number of splits. □

Theorems 3.1 and 3.2 combine to give the following bound on the space utilization of a relaxed B-tree:

THEOREM 3.4. *A relaxed B-tree contains at most* $(m/c)(a/(a - 1)) + \log_a(m/c) + 2$ *nodes.*

PROOF. The number of nodes is at most one per height plus the total number of splits. The total number of splits is upper bounded by $(m/c)(a/(a - 1))$ by the proof of Corollary 3.3. Combining this bound with the bound in Theorem 3.1 gives the result. □

By Theorem 3.4, the number of nodes is $\mathrm{O}(n/c)$ as long as $d \leq (1-\epsilon)m$ for some $\epsilon < 1$. When d/m approaches 1, the space utilization can degenerate, but then we can rebuild the tree. We discuss this in Section 5.

By using a related but different potential function that increases as a node becomes empty rather than as it becomes full, we can obtain a bound on node deletions that depends only on d, independent of the number of insertions m. We define the potential of a root to be zero, that of a non-root leaf of size j to be $\max\{0, c - j\}/c$, and that of a non-root internal node of size j and height h to be $\max\{0, a - j\}a^{h-1}$. We allow $j = 0$, to account for the situation in the middle of a deletion: a leaf of size zero has just lost its only item; an internal node of size 0 is empty and has just lost its only child.

An insertion cannot increase the potential: splitting a node creates two nodes of potential zero and adds one to the size of the parent; if the node split is the root, the new root has potential zero. Ignoring node deletions, the deletion of an item from a leaf increases the potential by at most $1/c$. Deletion of a node of size 0 and height h decreases the potential by that of the deleted node, namely a^h, and increases the potential of the parent of the deleted node by at most a^h, producing a non-positive total increase in potential. Deletion of a root of height $h > 0$ must be preceded by deletion of its only child, reducing the potential by a^{h-1}. If we truncate the potential function at height h by letting the potential of nodes of height greater than h be zero, then the deletion of a non-root node of height h reduces the potential by a^h. This gives the following theorem:

THEOREM 3.5. *The number of deletions of a root of height $h > 0$ is at most $d/(ca^{h-1})$. The number of deletions of a non-root node of height h is at most $d/(ca^h)$.*

COROLLARY 3.6. *While the tree remains non-empty, the total time for all restructuring during deletions is $\mathrm{O}(d/c)$, or $\mathrm{O}(1/c)$ amortized per deletion.*

PROOF. As long as the tree is non-empty, its root has positive height. Summing the bound in Theorem 3.5 from $h = 0$ to infinity gives an upper bound of $(d/c)(a/(a-1))$ on the total number of node deletions. □

In addition to the search time (which is proportional to the tree height) and the update time, a third way to measure the efficiency of a search tree is storage space. A standard B-tree has no under-filled nodes, so every node is at least half full if splitting is bottom-up, at least almost half full if splitting is top-down. On the other hand, nodes in relaxed B-trees can become arbitrarily under-filled, although each leaf must contain at least one item. Theorem 3.5 does imply a bound on the amount of unused space for keys in internal nodes, however. We define the *underflow* of an internal node of size j to be $\max\{0, a - j\}$.

COROLLARY 3.7. *The total underflow of all nodes of height $h > 0$ is at most $d/(ca^{h-1})$.*

If there is too much unused space, one can rebuild the tree, as we discuss in Section 5.

4. ALTERNATIVE DELETION METHODS

The deletion method described in Section 2 and analyzed in Section 3 deletes empty internal nodes bottom-up as leaves become empty. Alternatively, one can delete empty internal nodes more eagerly. This requires fusing and borrowing, which makes deletion as complicated as in standard B-trees. Nevertheless, we describe and analyze this approach as a way to further demonstrate the power of exponential potential functions, and to provide an additional basis for comparing our work with that of Huddleston and

Mehlhorn [Huddleston and Mehlhorn 1981; 1982] and Maier and Salveter [Maier and Salveter 1981].

An empty root is easy to delete: we merely make its only child the new root. Consider an empty node x with a non-empty parent. Since the parent is non-empty, the empty node has a sibling y. If y is not full, we can eliminate x by fusing it with y and demoting a key from the parent to y. This decreases the size of the parent by one and increases the size of y by one; it makes the parent empty if it previously had size two. In the extreme case in which x and y are empty and their parent has size two, nodes x and y fuse into a single node of size one and the parent becomes empty. Whether or not y is empty, the fusing decreases the number of empty nodes by one. If y has size at least three, we can make x non-empty by moving a key from the parent to x and a key from y to the parent. This increases the size of x to two, decreases the size of y by one, and does not change the size of the parent. The choice of whether to fuse or borrow is governed by the *fusing threshold* f: fuse if y has size less than f, borrow otherwise. To avoid creating an empty node by borrowing, we require $f > 2$. We choose f as small as possible, namely $f = 3$.

There are two ways to use fusing and borrowing to eliminate empty nodes. One is top-down: during a deletion (or during any search), eliminate each empty node encountered along the search path. Each fusing or borrowing reduces the number of empty nodes by one. This method maintains two invariants: all leaves are non-empty, and the current node of the search during deletion is non-empty. Thus when the search during a deletion reaches a leaf x, the parent z of x is non-empty. If x contains only the item to be deleted, the deletion of the item followed by a fusing or borrowing at x preserves the invariant that all leaves are non-empty. Such a fusing may make the parent node z empty, but we allow empty internal nodes. Indeed, fusings at leaves is the way this method creates empty nodes in the first place.

A more eager method is to eliminate empty nodes as soon as they are created, by doing fusing bottom-up when a deletion makes a leaf empty, continuing until the root is deleted, or a fusing results in a non-empty parent, or a fusing results in an empty parent that can be made non-empty by borrowing instead of fusing. With this method each item deletion triggers a sequence of fusings and at most one borrowing. It maintains the invariant that there are no empty nodes, from which it follows that the tree height is at most $\lg n + 1$, where $\lg$ denotes the base-two logarithm. (This bound is much worse than the bound in Theorem 3.1 unless b is small or m is huge compared to n.) We obtain interesting results as long as $a' \geq 3$. Since $f = 3$, $f \leq a'$, which implies that fusing cannot increase the size of an internal node beyond a', and Theorems 3.1 and 3.2 and Corollary 3.3 hold for this eager method by the proof in Section 3 (since a deletion still does not increase the potential). Theorem 3.5 remains true for height zero; in particular, the number of deletions of non-root leaves is at most d/c. We shall derive bounds on the cost of deletions for two cases: $a > 2$, and the boundary case $a = 2$. For $a > 2$, we use the second potential function in Section 3: the potential of a root is zero, that of a leaf of size j is $\max\{0, c - j\}/c$, and that of an internal node of size j and height h is $\max\{0, a - j\}a^{h-1}$. Leaves but not internal nodes can have $j = 0$. Insertions do not increase the potential. Deleting an item from a leaf increases the potential by at most $1/c$. Let x be an empty node with sibling y. If y has size less than a, we fuse x and y. The potential of the fused node is at least a^h less than the sum of the potentials of x and y, and the fusing increases the potential of the parent of x and y by at most a^h, so the net potential increase is non-positive. Borrowing from y does not increase the potential. If we truncate the potential at height h by making the potential of all nodes of height greater than h equal to zero, then fusing two nodes of height h decreases the potential by a^h. This gives the following theorem:

THEOREM 4.1. *If $a > 2$, the number of fusings at height h is at most $d/(ca^h)$.*

COROLLARY 4.2. *If $a > 2$, the number of borrowings at height h is at most $d/(ca^{h-1})$.*

PROOF. A borrowing eliminates an empty node. An empty node of height $h > 1$ can only be created by a fusing at height $h-1$, so the theorem holds in this case. An empty node of height 1 can only be created by deletion of a non-root leaf, of which there are at most d/c. □

For $a = 2$ and $a' = 3$, we modify the potential function on internal nodes: we make the potential of an internal node of size j and height h equal to $\max\{0, 3-j\}2^{h-1}$. Now an insertion can increase the potential: a split of an internal node of height h increases the potential by 2^{h-1} by creating a new node of height h and size 2. To bound such increases we shall use Theorem 3.2. A borrowing does not increase the potential. A fusing at height h decreases the sum of the potentials of the fused nodes from $3 \cdot 2^{h-1}$ to zero and increases the potential of the parent by at most 2^h, for a net decrease of at least 2^{h-1}. Now suppose we truncate the potential function by making it zero for nodes of height greater than h. Then a fusing at height h decreases the potential by $3 \cdot 2^{h-1}$. The only increases in potential are caused by item deletions, totaling d/c overall, and splits of internal nodes of height at most h, totaling by Theorem 3.2 at most $\sum_{i=1}^{h}(m/(c2^i))2^{i-1} = hm/(2c)$. This gives the following theorem:

THEOREM 4.3. *If $a = 2$ and $a' = 3$, the number of fusings at height h is at most $(hm/2 + d)/(3c2^{h-1})$.*

COROLLARY 4.4. *If $a = 2$ and $a' = 3$, the number of borrowings at height $h > 1$ is at most $((h-1)m/2 + d)/(3c2^{h-2})$. The number of borrowings at height 1 is at most d/c.*

PROOF. Analogous to the proof of Corollary 4.2. □

Theorem 4.3 and Corollary 4.4 imply that if $a = 2$ and $a' = 3$, the number of fusings and borrowings of height h is $O(m/\alpha^h)$ for any constant $1 < \alpha < 2$. We do not know if such a bound holds for $\alpha = 2$.

There is another way to eliminate empty nodes that we think is much better than doing fusing and borrowing. This is to replace each path of empty nodes by a single pointer. This gives a data structure equivalent to the one described in Section 2. Such replacement must be done carefully, however, since it is no longer true that all leaves have the same depth. We add to each node a non-negative integer *rank* that is a proxy for its height. Each leaf has rank zero. When a node becomes empty, we delete it; if it is a root, its only child becomes the new root, if it is a child, we replace the pointer to it from its parent by a pointer to its child from its parent. When a node x of rank h splits, we give both of the two nodes into which it splits a rank of h. If x is the root, we give the new root a rank of $h + 1$. If x is a child of a node of rank $h + 1$, we do the split as in Section 2, promoting a key to the parent, but if x is a child of a node of rank greater than $h + 1$, we create a new node y of rank $h + 1$ to hold the promoted key and pointers to the two nodes into which x is split, and we replace the pointer to x from its parent by a pointer to y. (See Figure 2.) This is equivalent to re-inserting a previously deleted empty node of height $h + 1$. With this method, all the results in Section 3 hold. In addition, a deletion requires at most two node deletions in the worst case, one of a leaf and one of an internal node, and $O(1)$ restructuring time.

It is tempting to try the same idea but without storing ranks (or something equivalent to ranks, such as rank differences), but this can fail badly, as we discuss in our companion paper on binary trees [Sen and Tarjan 2010]. The key to a working method is to replace a previously deleted empty node when its only child splits. If instead the
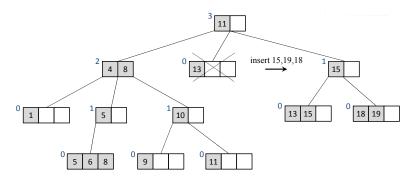
Fig. 2.   Alternative deletion method using ranks (shown to the left of each node) as a proxy for node height. Unlike the tree in Figure 1, each path of empty nodes is replaced by a single pointer. Inserting items 15, 19, and 18 causes a node of rank 0 (a leaf) to split. Since the node's parent (the root in this case) has rank greater than 1, a new internal node of rank 1 becomes the parent of the split nodes.

key promoted by such a split is moved all the way to the top of a compressed path of empty nodes, the tree can become very badly balanced. Indeed, the height can become super-logarithmic in the number of insertions.

5. REBUILDING THE TREE

Relaxed B-trees have two possible disadvantages: the number of nodes can become $\omega(n/c)$, and the search time can become $\omega(\log_a(n/c))$. This can only occur when m becomes large compared to n; that is, most of the inserted items have been deleted. These problems do not arise in many applications, and they do not seem to arise in actual database systems. If these problems do arise, we can fix them by periodically rebuilding the tree. How to rebuild the tree, and how often, are interesting questions that deserve careful study and experimentation. One simple scheme is to completely rebuild the tree each time $n/m < \epsilon$, where $0 < \epsilon \le 1/2$. More precisely, we initialize $m = n = 0$. On each insertion we increment m and n; on each deletion we decrement n. When $n/m < \epsilon$, we rebuild the tree and set $m = n$: the new tree has suffered no deletions. This guarantees that the number of nodes is $O(n/c)$ and the tree height is at most $\log_a(n/c) + 1 + \log_a(1/\epsilon)$.

The rebuilding process deletes items from the old tree in increasing order and inserts them into the new tree. In order to distinguish between insertions and deletions done by the application and those done by the rebuilding process, we shall call the latter *additions* and *removals*. To rebuild the tree, we initialize a new tree to be empty. We traverse the old tree in symmetric order, removing each item and adding it in the new tree. To facilitate the rebuilding process, we maintain the left spine of the old tree (the path from the root to the leftmost leaf) and the right spine of the new tree (the path from the root to the rightmost leaf) as lists of (pointers to) nodes, or by temporarily reversing child pointers along these paths, or temporarily storing parent pointers in the nodes along the paths. The last node on the first list contains the next item to be removed; the last node on the second list is the one to which it must be added. Thus the rebuilding process need not search in either tree. The nodes affected by removals are on the first list; the nodes affected by additions are those on the second list. We update the first list as empty nodes are removed, and those of the second list as nodes are split. The entire rebuilding process takes $O(n/c)$ time because only $O(n/c)$ nodes are accessed in both the old and new trees, and no searches are done. If d is the number of deletions since the last rebuilding, $\epsilon d/c = (\epsilon/c)(m-n) > (\epsilon/c)(n/\epsilon - n) = (n/c)(1-\epsilon) \ge n/(2c)$. Thus the rebuilding time is $O(\epsilon/c)$ amortized per deletion.

We can also do the rebuilding more incrementally. During rebuilding, we maintain two trees, an old tree and a new tree. All items in the new tree have keys smaller than all items in the old tree; the rebuilding process moves items from the old tree to the new tree. The following method guarantees that the rebuilding time is $O(\epsilon/c)$ amortized per deletion, $O(h+1)$ worst-case per insertion and deletion, and both the old tree and the new tree have height at most $\log_a(n/c) + 1 + \log_a(1/\epsilon)$, where $\epsilon \leq 1/4$ is a fixed positive constant. We begin building the new tree incrementally when a deletion makes $n/m < 2\epsilon$. During the rebuilding process, we move two items from the old tree to the new one after each insertion or deletion (in either the old or the new tree), including the deletion that triggers the rebuilding. Each insertion or deletion takes place in the appropriate tree. We keep track of the number of insertions into the new tree and the number of items added to it from the old tree, so that m and n have the correct value after the old tree becomes empty. As in non-incremental rebuilding, we maintain the left spine of the old tree and the right spine of the new tree, so that moving the item of smallest key from the old tree to the new tree can be done in $O(1/c)$ amortized time, without a search.

Let m be the number of insertions into the old tree and n the number of items in the old tree just before a deletion that triggers rebuilding. The time for rebuilding is $O(n/c)$, which by the argument above is $O(\epsilon/c)$ per deletion since the previous rebuilding. While rebuilding is on-going, each deletion, including those from the new tree, causes two additions to the new tree, so the number of items in the new tree is never less than the number of deletions from the new tree. Thus the ratio of items to insertions in the new tree never drops below $1/2$, and rebuilding of the new tree cannot be triggered until rebuilding of the old tree finishes. The height of the new tree while rebuilding is on-going is at most $\log_a(n/c) + 1 + \log_a(2) \leq \log_a(n/c) + 1 + \log_a(1/\epsilon)$. The number of insertions into the old tree while rebuilding is on-going is at most $n/2$, so the height of the old tree while rebuilding is on-going is at most $\log_a(m/c+n/(2c))+1 \leq \log_a(n/c) + 1 + \log_a(1/(2\epsilon) + 1/2) \leq \log_a(n/c) + 1 + \log_a(1/\epsilon)$.

One can also maintain just a single tree and run a background incremental rebuilding process. We leave the development of this idea and the further study of rebuilding as possible future work.

6. REMARKS

We have defined relaxed B-trees, which are like standard B-trees except that they allow nodes to be arbitrarily under-filled, even empty. In relaxed B-trees, deletions require no restructuring beyond elimination of empty nodes. We have shown that the tree height remains logarithmic in the number of insertions, independent of the number of deletions. We have introduced exponential potential functions to prove that in relaxed B-trees the number of node updates during insertions or deletions is inversely exponential in the node height. These results provide theoretical justification for the observed efficiency of such trees in certain database systems.

We have also shown that if the tree is rebuilt periodically, the space is linear and the tree height is logarithmic in the number of items in the tree. Rebuilding can be done all at once or incrementally, and the rebuilding time can be made $O(\epsilon/c)$ per deletion for an arbitrarily small positive ϵ.

We have also considered more eager strategies to eliminate empty nodes. One is to use fusing and borrowing, as in standard B-trees. This complicates the deletion algorithm but preserves our bounds. Our results for deletion with fusing and borrowing are similar to those of Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981; 1982] and Maier and Salveter [Maier and Salveter 1981]. Both pairs of researchers independently proposed relaxing the size lower bound on internal nodes to a value a'' such that $2 \leq a'' \leq a$. Maier and Salveter called the resulting data structure a

"hysterical" B-tree; Huddleston and Mehlhorn called it a "weak" B-tree. Maier and Salveter proved that the update time in such trees is $O(1)$ amortized per insertion. Huddleston and Mehlhorn proved this and stronger results as well, including inverse-exponential bounds on the number of node updates as a function of height. To obtain their results, they used a multilevel credit method. Our exponential potential function method unifies and simplifies their method. Neither pair of authors allowed $a'' = 1$, which is required to simplify deletion, nor did they consider top-down rebalancing, only bottom-up. They also did not analyze insertions and deletions separately, as we do, but together.

It is straightforward to re-derive the inverse-exponential bounds of Huddleston and Mehlhorn using exponential potential functions, and to extend their results to obtain separate bounds for insertions and deletions. We leave this to others; the best bounds come from allowing empty nodes, as we do. There is one interesting overlap between our results and those of Huddleston and Mehlhorn, however. This is for the case $a = 2$, $b = 4$ with bottom-up rebalancing during both insertions and deletions. Huddleston and Mehlhorn [Huddleston and Mehlhorn 1981, Theorem 5(b)] showed that the number of node updates at height h is at most $(m+d)(3/5)^{h-1}$. Our Theorem 4.3 and Corollary 4.4 give a bound of $O(m/\alpha^h)$ for any $\alpha < 2$, where the constant factor depends on α. Our improvement comes at least partially from first analyzing insertions, then analyzing deletions.

A second and we think better way to eliminate empty nodes that we have considered is to replace each path of empty nodes by a single pointer. This method requires storing an explicit rank at every node, but it preserves all the results in Section 3.

In a companion paper we study deletion without rebalancing in binary trees [Sen and Tarjan 2010]. In such trees it is not so simple to obtain good bounds: some straightforward methods fail. Our solution can be viewed as adapting to binary trees the idea of compressing paths of unary nodes and maintaining explicit node ranks that serve as a proxy for heights. The resulting data structure is both very simple and very efficient.

Overall, we conclude that it is better to avoid rebalancing during deletions: deletions become much simpler, and efficiency need not suffer.

REFERENCES

G. M. Adel'son-Vel'skii and E. M. Landis. 1962. An algorithm for the organization of information. *Sov. Math. Dokl.* 3 (1962), 1259–1262.

Alok Aggarwal and S. Vitter, Jeffrey. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

Arne Andersson. 1993. Balanced Search Trees Made Simple. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures (WADS)*. 60–71.

Rudolf Bayer. 1971. Binary B-Trees for Virtual Memory. In *Proceedings of the SIGFIDET Workshop on Data Description, Access and Control*. 219–235.

Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.* 1 (1972), 290–306.

Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.

Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.

J. Gray and A. Reuter (Eds.). 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California. 851–876 pages.

Leo J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science (FOCS)*. 8–21.

Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2009. Rank-Balanced Trees. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS)*. 351–362.

Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2014. Rank-Balanced Trees. *ACM Trans. Algorithms* (2014). To appear.

S. Huddleston and K. Mehlhorn. 1981. Robust balancing in B-trees. In *GI-Conference on Theoretical Computer Science (LNCS)*, Vol. 104. 234–244.

Scott Huddleston and Kurt Mehlhorn. 1982. A New Data Structure for Representing Sorted Lists. *Acta Informatica* 17, 2 (1982), 157–184.

Jan Jannink. 1995. Implementing Deletion in B$^+$-Trees. *SIGMOD Record* 24, 1 (1995), 33–38.

T. Johnson and D. Shasha. 1989. Utilization of B-trees with inserts, deletes and modifies. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 235–246.

Theodore Johnson and Dennis Shasha. 1993. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at-Half. *J. Comput. System Sci.* 47, 1 (1993), 45–76.

D. Maier and S. C. Salveter. 1981. Hysterical B-trees. *Inf. Proc. Lett.* 12, 4 (1981), 199–202.

J. Metzger. 1975. *Managing simultaneous operations in large ordered indexes*. Technical Report. Technische Universität München, Institut für Informatik, TUM-Math.

C. Mohan and Frank Levine. 1992. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *SIGMOD Record* 21, 2 (1992), 371–380.

J. Nievergelt and E. M. Reingold. 1973. Binary Search Trees of Bounded Balance. *SIAM J. on Comput.* 2, 1 (1973), 33–43.

Henk J. Olivié. 1982. A New Class of Balanced Search Trees: Half Balanced Binary Search Trees. *ITA* 16, 1 (1982), 51–71.

Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (ATC), FREENIX Track*. 183–191.

Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 2000. Berkeley DB. (2000).

Behrokh Samadi. 1976. B-Trees in a System with Multiple Users. *Inf. Proc. Lett.* 5, 4 (1976), 107–112.

Robert Sedgewick. 2008. Left-leaning Red-Black Trees. http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf. (2008).

Siddhartha Sen and Robert E. Tarjan. 2009. Deletion without Rebalancing in Multiway Search Trees. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC)*. 832–841.

Siddhartha Sen and Robert E. Tarjan. 2010. Deletion Without Rebalancing in Balanced Binary Trees. In *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1490–1499.

R. E. Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic and Disc. Methods* 6 (1985), 306–318.

Bin Zhang and Meichun Hsu. 1989. Unsafe Operations in B-Trees. *Acta Informatica* 26, 5 (1989), 421–438.