Deletion Without Rebalancing in Multiway Search Trees

Siddhartha Sen[1,3] and Robert E. Tarjan[1,2,3]

[1] Princeton University, {sssix,ret}@cs.princeton.edu
[2] HP Laboratories, Palo Alto CA 94304

Abstract. Many database systems that use a 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. We conclude that rebalancing on deletion can be considered harmful.

1 Introduction

Deletion in balanced search trees [2–6, 8–10, 14, 15, 18] 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 [1] did not discuss deletion, and many textbooks neglect it. Third, if operations on the search tree occur in parallel, as in many database systems that use B or B^+ trees, the synchronization necessary to do rebalancing on deletion reduces the available parallelism [7]. Whereas rebalancing *must* be done on insertion into a B or B^+ tree to guarantee correctness (nodes cannot become overfull), it is optional on deletion, since a B or B^+ tree remains a valid search tree even if it has underfilled nodes.

The first problem with deletion can be overcome by storing the items only in the external nodes of the tree, storing keys in the internal nodes to support search. B^+ trees [6] are an example of such a data structure. This takes extra space, but the 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 a height logarithmic in

[3] Research at Princeton University partially supported by NSF grants CCF-0830676 and CCF-0832797 and US-Israel Binational Science Foundation grant 2006204. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

the number of items. Nevertheless, this method has been used successfully in Berkeley DB [16, 17], which uses B^+ trees with underfilled nodes, and in other database systems.

Avoiding rebalancing on deletion has been justified empirically [7, 13, 16, 17] and by average-case analysis [11, 12], 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 [10]. Furthermore the idea of deletion without rebalancing has also been used in red-black trees, resulting in unforseen and unfortunate consequences in at least one application [19]. 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 time per insertion or deletion is $\mathrm{O}(1)$, and that nodes are affected by updates with a frequency exponentially small in their heights. These latter results do not hold for standard B^+ trees. Thus in some ways deletion with rebalancing is not only not helpful but actually harmful. 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 [19] we present similar results for balanced binary trees. These results require careful design of the deletion method; certain natural choices result in the tree height becoming linear in the number of insertions in the worst case.

The remainder of our paper consists of five sections. In Section 2 we define the B^- tree, a relaxed form of B^+ tree in which deletions are done without rebalancing. B^- trees are essentially those used in Berkeley DB. We describe how to do searches, insertions, and deletions in such trees. Insertions require node-splitting, which can be done either bottom-up or top-down; we describe both methods. Deletions require only deletion of empty nodes. In Section 3 we analyze B^- trees. We show that the height, and hence the search time, is $\mathrm{O}(\log_b m)$, where m is the total number of insertions and b is the maximum node degree. This bound is independent of the number of deletions. We also show that an insertion or deletion takes $\mathrm{O}(1)$ amortized time in addition to a search, and that nodes are modified by insertions and deletions with a frequency that is exponentially small in their height. (These results require $b > 3$ if node-splitting is top-down.)

In Section 4 we discuss how and when to rebuild the tree. Such rebuilding eliminates two drawbacks of B^- trees: it keeps the space used proportional to the number of items in the tree, and it keeps the height logarithmic in the number of items. If rebuilding is done appropriately often, the amortized rebuilding time is $\mathrm{O}(\epsilon)$ per insertion for an arbitrarily small positive constant ϵ. In Section 5 we sketch how to do rebalancing with deletion while retaining our inverse exponential bounds on node updates. Section 6 contains some concluding remarks, including a comparison between the case of multiway trees and that of binary trees.

2 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. We assume that the initial tree is empty. We measure the time of an operation by counting the number of nodes examined or modified. In B$^+$ trees, the structure of internal nodes differs from that of external nodes: internal nodes contain keys and pointers to children; external nodes contain items (and their keys) but no pointers. To allow for this difference, we define our trees using two parameters, which give upper bounds on the sizes of the internal and external nodes. A *B$^-$ tree of type $b > 2$, $c > 0$* consists of an ordered tree whose external nodes all have the same depth, each of whose internal nodes has at most b children, and each of whose external nodes contains at least one and at most c items. Generally we think of b and c as large but within a small constant factor of each other (though our results do not require this). Each item has a distinct key selected from a totally ordered universe. (If keys are not distinct we break ties by item identifier.) Increasing key order corresponds to left-to-right node order. That is, if external node y is to the right of external node x, all items in y have larger key than all items in x. In order to facilitate searching, each internal node x with j children contains $j-1$ keys in increasing order, alternating with pointers to its children; if a pointer to node y immediately follows (precedes) key k, then all items in the subtree rooted at y have key greater than (not greater than) k. We allow $j = 1$; an internal node with one child contains no key.

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

To insert a new item into a B$^-$ tree, if the tree is empty merely create a new external node containing the item. Otherwise, do a search for the key of the item. Upon reaching an external node, insert the new item into this node. If the node overflows (because it now has $c + 1$ items), split it into two external nodes, one containing the smallest $\lceil (c+1)/2 \rceil = \lfloor c/2 + 1 \rfloor$ items and the other containing the remaining (largest) $\lfloor (c+1)/2 \rfloor = \lceil c/2 \rceil$ items; in the parent, replace the pointer to the original external node by two pointers to the two nodes formed by the split, separated by a copy of the largest key in the new external node containing the smaller keys. If this causes the parent to overflow, because it now has $b+1$ children and b keys, split it, but in a slightly different way: find a median of its keys; put the keys smaller than the median and the child pointers preceding the median in a new node, and put the keys larger than the median and the child pointers following the median in another new node; in the parent, replace the pointer to the old node by two pointers to the new nodes, separated by the median. That is, the median is promoted to the parent, not copied. Walk back up the path toward the root, splitting nodes in this way, until some node does not overflow or the root splits. If the root splits, create a new root containing pointers to the two new nodes formed by the split, separated by the promoted or copied median. (The old root could be an internal or external node.)

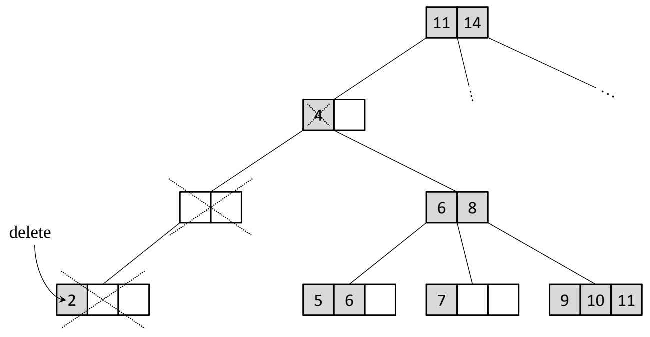

Fig. 1. Deletion in a B$^-$ tree with $b = c = 3$. Deleting item 2 in the above tree causes the additional node and item deletions shown in dotted crosses.

An alternative way to do an insertion is to split full nodes top-down as the search proceeds, rather than splitting overfull nodes bottom-up after the search. This method only gives good bounds if $b > 3$. To do an insertion, do a search on the key of the new item, but if the current node of the search is full (it has b children), find a median of its keys, split the remaining keys and the child pointers into two nodes, containing the keys less than the median and the child pointers preceding the median, and the keys greater than the median and the child pointers following the median, respectively; in the parent, replace the pointer to the old child by pointers to the two new nodes, separated by the median. (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 an external node, insert the new item into this node and split the node if it is overfull. (Again, the parent cannot be full.)

To delete an item in a B$^-$ tree, find the external node containing it and delete the item from the node. If the node 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 contained no key (and only one child pointer), delete the parent as well, and walk back up the path toward the root deleting each node with no children until reaching a node that still has at least one child or until reaching the root. If the root has no children, delete it; the 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. In the standard deletion method, a node that becomes underfull is refilled, by fusing it with a sibling (the inverse of splitting) and then possibly resplitting. Also, a root with only one child is deleted. What "underfull" means depends on whether node-splitting during insertion is bottom-up or top-down; in the former case, an internal node is underfull if it has less than $\lceil b/2 \rceil$ children; in the latter case, if it has less than $\lfloor b/2 \rfloor$ children. In either case, an external node is underfull if it has less than $\lceil c/2 \rceil$ items. A B$^+$ tree has height at most $\log_{\lceil b/2 \rceil}(n/c) + 1$ if splitting is bottom-up, at most $\log_{\lfloor b/2 \rfloor}(n/c) + 1$ if top-down. Avoiding refilling simplifies deletion considerably, but at the expense of having underfull nodes, which worsens space utilization and can increase the time for accesses and updates. Nevertheless, our analysis in the next section provides theoretical support for this method.

3 Analysis of B$^-$ Trees

To analyze B$^-$ trees we use the potential method of amortized analysis [20]. 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 [9, 19]. Each node has a non-negative potential; the potential of a tree is the sum of the potentials of its nodes. We begin by analyzing bottom-up splitting; then we modify the analysis to handle top-down splitting. We define the potential of a node of height $h > 0$ with j children to be $\max\{0, j - \lfloor b/2 + 1\rfloor\}\lceil b/2\rceil^h$ and the potential of an external node containing j items to be $\max\{0, j - \lfloor c/2+1\rfloor\}\lceil b/2\rceil/\lceil c/2\rceil$; to cover temporarily overfull nodes, we allow $j = b + 1$ for an internal node, $j = c + 1$ for an external node.

A deletion cannot increase the potential. Ignoring the effect of splits, an insertion increases the potential by at most $\lceil b/2\rceil/\lceil c/2\rceil$, by adding one item to an external node. If an overfull external node (containing $c + 1$ items) splits, the potential of both new nodes is zero; the potential of the parent (if any) increases by at most $\lceil b/2\rceil$. The potential of the old external node was $(c + 1 - \lfloor c/2 + 1\rfloor)\lceil c/2\rceil/\lceil b/2\rceil = \lceil b/2\rceil$, so the split does not increase the total potential. If an overfull internal node (having $b + 1$ children) splits, the potential of both new nodes resulting from the split is zero, and the potential of the parent (if any) increases by at most $\lceil b/2\rceil^{h+1}$. The potential of the old node that split was $(b + 1 - \lfloor b/2 + 1\rfloor)\lceil b/2\rceil^h = \lceil b/2\rceil^{h+1}$, so the split does not increase the potential. If the node that splits is the root, the potential decreases by $\lceil b/2\rceil^{h+1}$. Since the potential can never be negative and increases by a total of at most $m\lceil b/2\rceil/\lceil c/2\rceil$, we obtain the following theorem:

Theorem 1. *A B$^-$ tree built by m insertions with bottom-up splitting intermixed with arbitrary deletions has height at most* $\log_{\lceil b/2\rceil}(m/\lceil c/2\rceil) + 1$.

Thus as long as m, the total number of insertions, is polynomial in n, the current number of items in the tree, the B$^-$ tree built by an update sequence has height within a multiplicative constant of that of the B$^+$ tree built by the same update sequence; the constant depends on the degree of the polynomial. If m is linear in n, the height of the B$^-$ tree is within an additive constant of that of the B$^+$ tree.

By truncating the potential function, we can obtain an inverse-exponential bound on the number of splits, and the number of nodes, of a given height. For any fixed 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 at height h reduces the potential by $\lceil b/2\rceil^{h+1}$, which gives the following theorem:

Theorem 2. *In a B$^-$ tree built by m insertions with bottom-up splitting intermixed with arbitrary deletions, the number of splits of nodes at height h is at most* $m/(\lceil b/2\rceil^h\lceil c/2\rceil)$.

Corollary 1. *In a B$^-$ tree built by m insertions with bottom-up splitting intermixed with arbitrary deletions, the number of deletions of roots of height $h > 0$ is at most $m/(\lceil b/2\rceil^{h-1}\lceil c/2\rceil)$. The number of deletions of non-roots of height h is at most $m/(\lceil b/2\rceil^{h}\lceil c/2\rceil)$.*

Proof. The number of nodes deleted at height h is at most the number added. For a node to be added at height h, a split must occur at height $h-1$. Each deletion of a non-root at height h must correspond to a previous split at height h. Thus both parts of the corollary follow from Theorem 2. $\square$

Corollary 2. *With bottom-up splitting, the amortized time for an insertion, excluding the search time, is* $\mathrm{O}(1)$. *The amortized time for a deletion is zero. (Each deletion can be charged to the corresponding insertion.)*

Proof. The sum over all heights of the bounds given by Theorem 2 on node splits and by Corollary 1 on node deletions is a geometric series summing to $\mathrm{O}(m)$. $\square$

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 as a function of d, the number of item deletions, rather than m. We define the potential of a root to be zero, that of a child of height $h > 0$ with j children to be $\max\{0, \lceil b/2\rceil - j\}\lceil b/2\rceil^h$, and that of an external child containing j items to be $\max\{0, \lceil c/2\rceil - j\}\lceil b/2\rceil/\lceil c/2\rceil$; to cover underfull nodes, we allow $j = 0$.

An insertion cannot increase the potential: splitting a node creates two nodes of potential zero and adds a child to the parent; if there is no parent, a new root is created, of potential zero. Ignoring node deletions, an item deletion increases the potential by at most $\lceil b/2\rceil/\lceil c/2\rceil$. Deletion of a node of height h decreases the potential by that of the deleted node, namely $\lceil b/2\rceil^{h+1}$, and increases the potential of the parent by at most the same amount, resulting in no net increase. A deletion of a root of height $h > 0$ must be preceded by a deletion of its only child, reducing the potential by $\lceil b/2\rceil^h$. If we truncate the potential function by letting the potential of nodes of height greater than h be zero for some fixed h, then a deletion of a node of height h other than the root reduces the potential by $\lceil b/2\rceil^{h+1}$. This gives the following theorem:

Theorem 3. *In a B$^-$ tree built by insertions with bottom-up splitting intermixed with d arbitrary deletions, the number of deletions of roots of height $h > 0$ is at most $d/(\lceil b/2\rceil^{h-1}\lceil c/2\rceil)$. The number of deletions of non-roots of height h is at most $d/(\lceil b/2\rceil^{h}\lceil c/2\rceil)$.*

For top-down splitting we obtain the same results, but with $\lfloor b/2\rfloor$ in place of $\lceil b/2\rceil$, so the bounds are slightly worse. We assume $b > 3$. Let the potential of a node of height $h > 0$ with j children be $\max\{0, j - \lceil b/2\rceil\}\lfloor b/2\rfloor^h$ and that of an external node containing j items be $\max\{0, j - \lfloor b/2 + 1\rfloor\}\lfloor b/2\rfloor/\lceil c/2\rceil$. An argument like that preceding Theorem 1 gives the following:

Theorem 4. *A B$^-$ tree built by m insertions with top-down splitting intermixed with arbitrary deletions has height at most* $\log_{\lfloor b/2\rfloor}(m/\lceil c/2\rceil) + 1$.

By defining the potential of a node of height exceeding h to be zero for any fixed h, we obtain the following analogues of Theorem 2, Corollary 1, and Corollary 2:

Theorem 5. *In a B$^-$ tree built by m insertions with top-down splitting intermixed with arbitrary deletions, the number of splits of nodes at height h is at most $m/(\lfloor b/2 \rfloor^h \lceil c/2 \rceil)$.*

Corollary 3. *In a B$^-$ tree built by m insertions with top-down splitting intermixed with arbitrary deletions, the number of deletions of roots of height $h > 0$ is at most $m/(\lfloor b/2 \rfloor^{h-1} \lceil c/2 \rceil)$. The number of deletions of non-roots of height h is at most $m/(\lfloor b/2 \rfloor^h \lceil c/2 \rceil)$.*

Corollary 4. *With top-down splitting, the amortized time for an insertion, excluding the search time, is* $\mathrm{O}(1)$.

To obtain a bound on node deletions as a function of d, we let the potential of a root be zero, the potential of a child of height $h > 0$ with j children be $\max\{0, \lfloor b/2 \rfloor - j\} \lfloor b/2 \rfloor^h$, and that of an external child containing j items be $\max\{0, \lceil c/2 \rceil - j\} \lfloor b/2 \rfloor / \lceil c/2 \rceil$. An argument like that preceding Theorem 3 gives the following:

Theorem 6. *In a B$^-$ tree built by insertions with top-down splitting intermixed with d arbitrary deletions, the number of deletions of roots of height $h > 0$ is at most $d/(\lfloor b/2 \rfloor^{h-1} \lceil c/2 \rceil)$. The number of deletions of non-roots of height h is at most $d/(\lfloor b/2 \rfloor^h \lceil c/2 \rceil)$.*

4 Rebuilding the Tree

B$^-$ trees have two possible disadvantages: their space usage may be $\omega(n)$, where n is the current number of items, and the search time may exceed $\mathrm{O}(\log_b n)$. This can only occur when d/m approaches one. For applications in which insertions significantly outnumber deletions, this is not a concern, but for other applications it may be. We can solve both problems by periodically rebuilding the tree. How to rebuild the tree, and how often, are interesting questions that deserve careful study and experimentation. We describe a simple rebuilding method that takes $\mathrm{O}(\epsilon)$ amortized time per insertion for an arbitrarily small positive constant ϵ, analogous to the rebuilding method for binary trees presented in [19].

To rebuild the tree, we initialize a new tree to empty. We traverse the old tree in symmetric order, deleting each item and inserting it into the new tree. To facilitate the rebuilding process, we maintain the right spine (the path from the root to the rightmost external node) of the new tree as a list. The list provides the location of the next insertion: the new item is inserted into the last node on the list. Thus there is no need to search for the insertion location. The list also contains consecutively all the nodes at which splits occur. When a node is split, the new node containing the larger keys replaces the original node on the list. A split at the root additionally creates a new root, which is added to the end of the list. Each insertion into the new tree takes $\mathrm{O}(1)$ amortized time, and the entire rebuilding process takes $\mathrm{O}(n)$ time.

To decide when to rebuild the tree, we keep track of n and rebuild the tree when the space usage or the tree height (or both) becomes too high. If we rebuild the tree when the space usage becomes greater than an for a suitably large but fixed constant a, then the space usage of the tree is always $\mathrm{O}(n)$, and the rebuilding time is $\mathrm{O}(1/a)$ per insertion. Similarly, if we rebuild the tree when its height exceeds $\log_{\lceil b/2 \rceil}(n/c) + g$ for a suitably large but fixed constant g, then the height of the tree is always $\mathrm{O}(\log_b n)$ and the rebuilding time is $\mathrm{O}(1/\lceil b/2 \rceil^g)$ per insertion. The larger the values of a and g, the smaller the overhead for rebuilding, but the worse the space usage and the larger the tree height can become in terms of n.

Rebuilding can also be done incrementally, by maintaining both the old tree and the new tree and doing $\mathrm{O}(\epsilon)$ rebuilding work per insertion. For example, we can start the rebuilding process when the tree exceeds the maximum space usage or height, and then move two items from the old tree to the new tree for each subsequent insertion or deletion until the old tree becomes empty. During rebuilding, insertions occur in the new tree if the key of the item is less than the largest key of an item in the old tree moved so far, and in the old tree otherwise. We maintain the left spine of the old tree and the right spine of the new tree as lists, so the next item to be deleted from the old tree and its new location in the new tree can be found in $\mathrm{O}(1)$ time. We need to update these lists when insertions and deletions occur, but this takes $\mathrm{O}(1)$ amortized time per insertion or deletion. If n is the number of items in the old tree at the start of the rebuilding process, the number of items in the new tree at the end of the rebuilding process is between $n/2$ and $2n$.

Whether the tree is rebuilt incrementally or all at once, the space usage of the tree is always $\mathrm{O}(n)$ and the tree height is always $\mathrm{O}(\log_b n)$. The inverse-exponential bounds on node updates in Section 3 also continue to hold.

5 Deletion with Weak Rebalancing

As an alternative to rebuilding, one can obtain our inverse-exponential bounds on node updates (with a worse base), while guaranteeing linear space usage and $\mathrm{O}(\log_b n)$ search time, by refilling nodes that become too empty. This makes deletion as complicated as in standard B^+ trees, but achieves $\mathrm{O}(1)$ rebalancing time per insertion or deletion (not including search time).

To obtain the bounds we seek, we need to allow nodes to become emptier than in standard B^+ trees. Rebalancing during a deletion can be done either bottom-up or top-down. In either case, when an external node contains too few items or an internal node has too few children, we fuse it with a sibling, by combining the contents of both nodes and, if the node is internal, adding the key in the parent that is between the pointers to the two nodes being fused. Whether the node is internal or not, we replace the key in the parent and its adjacent pointers by a pointer to the new node. Then, if the new node is too full, we resplit it evenly, as in an insertion. To rebalance bottom-up, we start at the external node that has just lost an item, fuse it with a sibling if necessary, and walk up toward the root fusing each node that is too empty with a sibling, until reaching a node that is full enough, or resplitting a fused node, or reaching the root. If the root is internal and loses all but one child, we delete it and make its only child the new root. Top-down rebalancing works in the same way, except that we pre-emptively do the fusing top-down during the search; if the root is left with only one child as a result of a fusion of its children, we delete it.

One can easily parameterize the results based on the sizes at which fusions and resplits occur, but here we consider one illustrative special case. Suppose fusion is bottom-up, we fuse an internal node when it has less than $\lfloor b/8 \rfloor$ children, an external node when it has less than $\lfloor c/8 \rfloor$ items, and we resplit an internal node if it has more than $\lceil b/2 \rceil$ children, an external node when it has more than $\lceil c/2 \rceil$ items. We need b and c sufficiently large, say at least 16. Theorems 1, 2, 4, and 5, and Corollaries 2 and 4 still hold, since nodes created by fusion with or without resplitting have zero potential. The tree height is at most $\log_{\lfloor b/8 \rfloor}(n/\lfloor c/8 \rfloor) + 1$. Furthermore we have the following result:

Theorem 7. *Whether splitting during insertion is bottom-up or top-down, the number of node fusings (with or without resplittings) of nodes at height h is at most $d/(\lfloor b/8 \rfloor^h \lfloor c/8 \rfloor)$.*

Proof. Let the potential of a root be zero, that of an internal node of height h with j children be $\max\{0, 2\lfloor b/8 \rfloor - j\}\lfloor b/8 \rfloor^h$, and that of an external node with j items be $\max\{0, 2\lfloor c/8 \rfloor - j\}\lfloor b/8 \rfloor/\lfloor c/8 \rfloor$. An insertion does not increase the potential. Excluding node fusings and resplittings, a deletion increases the potential by at most $\lfloor b/8 \rfloor/\lfloor c/8 \rfloor$. A fusing of two nodes at height h without a resplit does not increase the potential, since the fused node has potential at least $\lfloor b/8 \rfloor^{h+1}$ less than the sum of the potentials of the two nodes that fuse, and the potential of the parent increases by at most this amount. A fusing of two nodes at height h with a resplit does not change the potential of the parent and decreases the total potential of the affected nodes at height h by at least $\lfloor b/8 \rfloor^{h+1}$. If we change the potential of nodes exceeding a fixed height h to zero, then any fusing at height h decreases the potential by at least $\lfloor b/8 \rfloor^{h+1}$. The theorem follows. □

6 Remarks

We have shown that in B$^+$ trees deletion without rebalancing preserves a logarithmic height bound, and that with this method node updates during insertions and deletions occur exponentially infrequently in the height of the node. If one rebuilds the tree periodically, the space usage remains linear and the height remains logarithmic in the number of items in the tree; the rebuilding time can be made $O(\epsilon)$ per insertion for an arbitrarily small positive constant ϵ. We have obtained similar inverse exponential bounds if rebalancing is done on deletion but nodes are allowed to be less full than in standard B$^+$ trees. Such "weak" rebalancing takes $O(1)$ amortized time per insertion or deletion, whereas in standard B$^+$ trees the amortized time per insertion or deletion can be logarithmic in n. Our bounds for weak rebalancing have worse constants than those for rebalancing without deletion, however.

We have obtained similar results for balanced binary trees [9, 19]. In the case of binary trees it is not so simple to do deletion without rebalancing and still obtain good bounds: the deletion method we have analyzed here can produce long chains of unary nodes, but binary trees do not have unary nodes. We could eliminate unary nodes in our solution for multiway trees by making the parent of such a node point to its only child, but this method fails unless one keeps track, for each pointer replacing a path of unary nodes, of the length of the path it replaces. This is what we did to solve the problem for binary trees, and this seems to be required. See [19].

References

1. G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.
2. A. Andersson. Balanced search trees made simple. In *WADS*, pages 60–71, 1993.
3. R. Bayer. Binary B-trees for virtual memory. In *SIGFIDET*, pages 219–235, 1971.
4. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.
5. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.
6. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
7. J. Gray and A. Reuter, editors. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.
8. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21, 1978.
9. B. Haeupler, S. Sen, and R. E. Tarjan. Rank-balanced trees. In *WADS*, pages 351–362, 2009.
10. J. Jannink. Implementing deletion in B$^+$-trees. *SIGMOD Record*, 24(1):33–38, 1995.
11. T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *PODS*, pages 235–246, 1989.
12. T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, 1993.
13. C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *SIGMOD Record*, 21(2):371–380, 1992.
14. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. on Comput.*, pages 33–43, 1973.
15. H. J. Olivié. A new class of balanced search trees: Half balanced binary search trees. *ITA*, 16(1):51–71, 1982.
16. M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual, FREENIX Track*, pages 183–191, 1999.
17. M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB, 2000.
18. R. Sedgewick. Left-leaning red-black trees. www.cs.princeton.edu/ rs/talks/LLRB/LLRB.pdf.
19. S. Sen and R. E. Tarjan. Deletion without rebalancing in balanced binary trees. In *SODA*, 2010. To Appear.
20. R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic and Disc. Methods*, 6:306–318, 1985.