B-slack trees: Space Efficient B-trees

Trevor Brown

Department of Computer Science, University of Toronto, Canada
tabrown@cs.toronto.edu

Abstract. B-slack trees, a subclass of B-trees that have substantially better worst-case space complexity, are introduced. They store n keys in height $O(\log_b n)$, where b is the maximum node degree. Updates can be performed in $O(\log_{\frac{b}{2}} n)$ amortized time. A relaxed balance version, which is well suited for concurrent implementation, is also presented.

1 Introduction

B-trees are balanced trees designed for block-based storage media. Internal nodes contain between $b/2$ and b child pointers, and one less key. Leaves contain between $b/2$ and b keys. All leaves have the same depth, so access times are predictable. If memory can be allocated on a per-byte basis, nodes can simply be allocated the precise amount of space they need to store their data, and no space is wasted. However, typically, all nodes have the same, fixed capacity, and some of the capacity of nodes is wasted. As much as 50% of the capacity of each node is wasted in the worst case. This is particularly problematic when data structures are being implemented in hardware, since memory allocation and reclamation schemes are often very simplistic, allowing only a single block size to be allocated (to avoid fragmentation). Furthermore, since hardware devices must include sufficient resources to handle the worst-case, good expected behaviour is not enough to allow hardware developers to reduce the amount of memory included in their devices. To address this problem, we introduce *B-slack trees*, which are a variant of B-trees with substantially better worst-case space complexity. We also introduce relaxed B-slack trees, which are a variant of B-slack trees that are more amenable to concurrent implementation.

The development of B-slack trees was inspired by a collaboration with a manufacturer of internet routers, who wanted to build a concurrent router based on a tree. In such embedded devices, storage is limited, so it is important to use it as efficiently as possible. A suitable tree would have a simple algorithm for updates, small space complexity, fast searches, and searches that would not be blocked by concurrent updates. Updates were expected to be infrequent. One naive approach is to rebuild the entire tree after each update. Keeping an old copy of the tree while rebuilding a new copy would allow searches to proceed unhindered, but this would double the space required to store the tree.

Search trees can be either node-oriented, in which each key is stored in an internal node or leaf, or leaf-oriented, in which each key is stored at a leaf and the

keys of internal nodes serve only to direct a search to the appropriate leaf. In a node-oriented B-tree, the leaves and internal nodes have different sizes (because internal nodes contain keys and pointers to children, and leaves contain only keys). So, if only one block size can be allocated, a significant amount of space is wasted. Moreover, deletion in a node-oriented tree sometimes requires stealing a key from a successor (or predecessor), which can be in a different part of the tree. This is a problem for concurrent implementation, since the operation involves a large number of nodes, namely, the nodes on the path between the node and its successor.

B-slack trees are leaf-oriented trees with many desirable properties. The average degree of nodes is high, exceeding $b - 1.4$ for trees of height at least three. Their space complexity is better than all of their competitors. Consider a dictionary implemented by a leaf-oriented search tree, in which, along with each key, a leaf stores a pointer to associated data. Suppose that each key and each pointer to a child or to data occupies a single word. Then, $\frac{2b}{b-2.4}n$ is an upper bound on the number of words needed to store a B-slack tree with $n > b^3$ keys. For large b, this tends to $2n$, which is optimal. B-slack trees have logarithmic height, and the number of rebalancing steps performed after a sequence of m updates to a B-slack tree of size n is amortized $O(\log(n+m))$ per update. Furthermore, the number of rebalancing steps needed to rebalance the tree can be reduced to amortized *constant* per update at the cost of slightly increased space complexity, as will be explained in the full version of the paper.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 introduces B-slack trees and relaxed B-slack trees. Height, average degree, space complexity and rebalancing costs of relaxed B-slack trees (and, hence, of B-slack trees) are analyzed in Section 4. Finally, we conclude in Section 5.

2 Related Work

B-trees were initially proposed by Bayer and McCreight in 1970 [4]. Insertion into a full node in a B-tree causes it to split into two nodes, each half full. Deletion from a half-full node causes it to merge with a neighbour. Arnow, Tenenbaum and Wu proposed P-trees [2], which enjoy moderate improvements to average space complexity over B-trees, but waste 66% of each node in the worst case.

A number of generalizations of B-trees have been suggested that achieve much less waste if no deletions are performed. Bayer and McCreight also proposed B*-trees in [4], which improve the worst-case space complexity. At most a third of the capacity of each node in a B*-tree is wasted. This is achieved by splitting a node only when it and one of its neighbours are both full, replacing these two nodes by three nodes. Küspert [9] generalized B*-trees to trees where each node contains between $\lfloor \frac{bm}{m+1} \rfloor$ and b pointers or keys, where $m \leq b - 1$ is a design parameter. Such a tree behaves just like a B*-tree everywhere except at the leaves. An insertion into a full leaf causes keys to be shifted among the nearest $m - 1$ siblings to make room for the inserted key. If the $m - 1$ nearest siblings

are also full, then these m nodes are replaced by $m+1$ nodes which evenly share keys. Large values of m yield good worst-case space complexity.

Baeza-Yates and Per-åke Larson introduced B+trees with partial expansions [3]. Several node sizes are used, each a multiple of the block size. An insertion to a full node causes it to expand to the next larger node size. With three node sizes, at most 33% of each node can be wasted, and worst-case utilization improves with the number of block sizes used. However, this technique simply pushes the complexity of the problem onto the memory allocator. Memory allocation is relatively simple for one block size, but it quickly becomes impractical for simple hardware to support larger numbers of block sizes.

Culik, Ottmann and Wood introduced strongly dense multiway trees (SDM-trees) [7]. An SDM-tree is a node-oriented tree in which all leaves have the same depth, and the root contains at least two pointers. Apart from the root, every internal node u with fewer than b pointers has at least one sibling. Each sibling of u has b pointers if it is an internal node and b keys if it is a leaf. Insertion can be done in $O(b^3 + (\log n)^{b-2})$ time. Deletion is not supported, but the authors mention that the insertion algorithm could be modified to obtain a deletion algorithm, and the time complexity of the resulting algorithm "would be at most $O(n)$ and at least $O((\log n)^{b-1})$." Besides the long running times for each operation (and the lack of better amortized results), the insertion algorithm is very complex and involves many nodes, which makes it poorly suited for hardware implementation. Furthermore, in a concurrent setting, an extremely large section of the tree would have to be modified atomically, which would severely limit concurrency.

Srinivasan introduced a leaf-oriented B-tree variant called an *Overflow tree* [12]. For each parent of a leaf, its children are divided into one or more groups, and an overflow node is associated with each group. The tree satisfies the B-tree properties and the additional requirement that each leaf contains at least $b-1-s$ keys, where $s \geq 2$ is a design parameter and b is the maximum degree of nodes. Inserting a key into a full leaf causes the key to be inserted into the overflow node instead; if the overflow node is full, the entire group is reorganized. Deleting from a leaf is the same as in a B-tree unless it will cause the leaf to contain too few keys, in which case, a key is taken from the overflow node; if a key cannot be taken from the overflow node, the entire group is reorganized. Each search must look at an overflow node. The need to atomically modify and search two places at once makes this data structure poorly suited for concurrent implementation.

Hsuang introduced a class of node-oriented trees called *H-trees* [8], which are a subclass of B-trees parameterized by γ and δ. These parameters specify a lower bound on the number of grandchildren of each internal node (that has grandchildren), and a lower bound on the number of keys contained in each leaf, respectively. Larger values of δ and γ yield trees that use memory more efficiently. When δ and γ are as large as possible, each leaf contains at least $b-3$ keys, and each internal node has zero or at least $\lfloor \frac{b^2+1}{2} \rfloor$ grandchildren. The paper presents $O(\log n)$ insertion and deletion algorithms for node-oriented H-trees. The algorithms are very complex and involve many cases. H-trees have

a minimum average degree of approximately $b/\sqrt{2}$ for internal nodes, which is much smaller than the $b - 1.4$ of B-slack trees (for trees of height at least three).

The full version of the paper describes families of B-trees, H-trees and Overflow trees which require significantly more space than B-slack trees.

Rosenberg and Snyder introduced *compact B-trees* [11], which can be constructed from a set of keys using the minimum number of nodes possible. No compactness preserving insertion or deletion procedures are known. The authors suggested using regular B-tree updates and periodically compacting a data structure to improve efficiency. However, experiments in [1] showed that starting with a compact B-tree and adding only 1.6% more keys using standard B-tree operations reduced storage utilization from 99% to 67%.

An impressive paper by Bronnimann et al. [5] presented three ways to transform an arbitrary sequential dictionary into a more space efficient one. One of these ways will be discussed here; of the other two, one is extremely complex and poorly suited for concurrent hardware implementation, and the other pushes the complexity onto the memory allocator.

Bronnimann's transformation takes any sequential tree data structure and modifies it by replacing each key in the sequential data structure with a *chunk*, which is a group of $b - 2$, $b - 1$ or b keys, where b is the memory block size. All chunks in the data structure are also kept in a doubly linked list to facilitate iteration and movement of keys between chunks. For instance, a BST would be transformed into a tree in which each node has zero, one or two children, and $b - 2$, $b - 1$ or b keys. All keys in chunks in the left subtree of a node u would be smaller than all keys in u's chunk, and all keys in chunks in the right subtree of u would be larger than all keys in u's chunk. A search for key k behaves the same as in the sequential data structure until it reaches the only chunk that can contain k, and searches for k within the chunk. An insertion first searches for the appropriate chunk, then it inserts the key into this chunk. Inserting into a full chunk requires shifting the keys of the b nearest other chunks to make room. If the b closest neighboring chunks are full, then a key is taken from each, and a new node containing b keys is inserted using the sequential data structure's insertion algorithm. Deletion is similar. Each operation in the resulting data structure runs in $O(f(n) + b^2)$ steps, where $f(n)$ is the number of steps taken by the sequential data structure to perform the same operation.

After this transformation, a B-tree with maximum degree b requires $2n + O(n/b)$ words to store n keys and pointers to data. In the worst-case, each chunk wastes $2/b$ of its space, which is somewhat worse than in B-slack trees. Furthermore, supporting fast searches can introduce significant complexity to the hardware design. A node in the transformed B-tree contains up to $b - 1$ chunks, each of which occupies one block of memory. Therefore, hardware must be able to quickly load up to $b - 1$ blocks at once, or else deciding which child pointer to follow will be slow.

3 B-slack trees

A B-slack tree is a variant of a B-tree. Each node stores its keys in sorted order, so binary search can be used to determine which child of an internal node should be visited next by a search, or whether a leaf contains a key. Let $p_0, p_1, ..., p_m$ be the sequence of pointers contained in an internal node, and $k_1, k_2, ..., k_m$ be its sequence of keys. For each $1 \leq i \leq m$, the subtree pointed to by p_{i-1} contains keys strictly smaller than k_i, and the subtree pointed to by p_i contains keys greater than *or equal to* k_i. We say that the *degree of an internal node* is the number of non-NIL pointers it contains, and the *degree of a leaf* is the number of keys it contains. This unusual definition of degree simplifies our discussion. The degree of node v is denoted $deg(v)$. If the maximum possible degree of a node is b, and its degree is $b - x$, then we say it contains x *slack*.

A B-slack tree is a leaf-oriented search tree with maximum degree $b > 4$ in which:

- **P1:** every leaf has the same depth,
- **P2:** internal nodes contain between 2 and b pointers (and one less key),
- **P3:** leaves contain between 0 and b keys, and
- **P4:** for each internal node u, the total slack contained in the children of u is at most $b - 1$.

P4 is the key property that distinguishes B-slack trees from other variants of B-trees. It limits the aggregate space wasted by a number of nodes, as opposed to limiting the space wasted by each node. Alternatively, P4 can be thought of as a lower bound on the sum of the degrees of the children of each internal node. Formally, for each internal node with children $v_1, v_2, ..., v_l$, $deg(v_1) + deg(v_2) + ... + deg(v_l) \geq lb - (b-1) = lb - b + 1$. This interpretation is useful to show that all nodes have large subtrees. For instance, it implies that a node u with two internal children must have at least $b + 1$ grandchildren. If these grandchildren are also internal nodes, we can conclude that u must have at least $b^2 - b + 2$ great grandchildren.

A tree that satisfies P1, and in which every node has degree $b - 1$, is an example of a B-slack tree. Another example of a B-slack tree is a tree of height two, where b is even, the root has degree two, its two children have degree $b/2$ and $b/2+1$, respectively, and the grandchildren of the root are leaves with degree b, except for two, one in the left subtree of the root, and one in the right subtree, that each have degree one. This tree contains the smallest number of keys of any B-slack tree of height two.

3.1 Relaxed B-slack trees

A relaxed balance search tree decouples updates that rebalance (or reorganize the keys of) the tree from updates that modify the set of keys stored in the tree [10]. The advantages of this decoupling are twofold. First, updates to a relaxed balance version of a search tree are smaller, so a greater degree of concurrency is possible in a multithreaded setting. Second, for some applications, it may be useful to temporarily disable rebalancing to allow a large number of updates to be performed quickly, and to gradually rebalance the tree afterwards.

A relaxed B-slack tree is a relaxed balance version of a B-slack tree that has weakened the properties. A weight of zero or one is associated with each node. These weights serve a purpose similar to the colors red and black in a red-black tree. We define the *relaxed depth* of a node to be one less than the sum of the weights on the path from the root to this node. A relaxed B-slack tree is a leaf-oriented search tree with maximum degree $b > 4$ in which:

P0′: every node with weight zero contains exactly two pointers,
P1′: every leaf has the same relaxed depth,
P2′: internal nodes contain between 1 and b pointers (and one less key), and
P3 : leaves contain between 0 and b keys

To clarify the difference between B-slack trees and relaxed B-slack trees, we identify several types of *violations* of the B-slack trees properties that can be present in a relaxed B-slack tree. We say that a *weight violation* occurs at a node with weight zero, a *slack violation* occurs at a node that violates P4, and a *degree violation* occurs at an internal node with only one child (violating P2). Observe that P1 is satisfied in a relaxed B-slack tree with no weight violations. Likewise, P2 is satisfied in a relaxed B-slack tree with no degree violations, and P4 is satisfied in a relaxed B-slack tree with no slack violations. Therefore, a relaxed B-slack tree that contains no violations is a B-slack tree. Rebalancing steps can be performed to eliminate violations, and gradually transform any relaxed B-slack tree into a B-slack tree.

3.2 Updates to relaxed B-slack trees

We now describe the algorithms for inserting and deleting keys in a relaxed B-slack trees (in a way that maintains P0′, P1′, P2′ and P3). The updates for relaxed B-slack trees are shown in Figure 1. There, weights appear to the right of nodes, and shaded regions represent slack. If u is a node that is not the root, then we let $\pi(u)$ denote the parent of u. Our insertion and deletion algorithms always ensure that all leaves have weight one.

Deletion. First, a search is performed to find the leaf u where the deletion should occur. If the leaf does not contain the key to be deleted, then the deletion terminates immediately, and the tree does not change. If the leaf contains the key to be deleted, then the key is removed from the sequence of keys stored in that leaf. Deleting this key may create a slack violation.

Insertion. To perform an insertion, a search is first performed to find the leaf u where the insertion should occur. If u contains some slack, then the key is added to the sequence of keys in u, and the insertion terminates. Otherwise, u cannot accommodate the new key, so Overflow is performed. Overflow replaces u by a subtree of height one consisting of an internal node with weight zero, and two leaves with weight one. The b keys stored in u, plus the new key, are evenly distributed between the children of the new internal node. If u was the root before the insertion, then the new internal node becomes the new root. Otherwise, u's parent $\pi(u)$ before the insertion is changed to point to the new internal node instead of u. After Overflow, there is a weight violation at the new

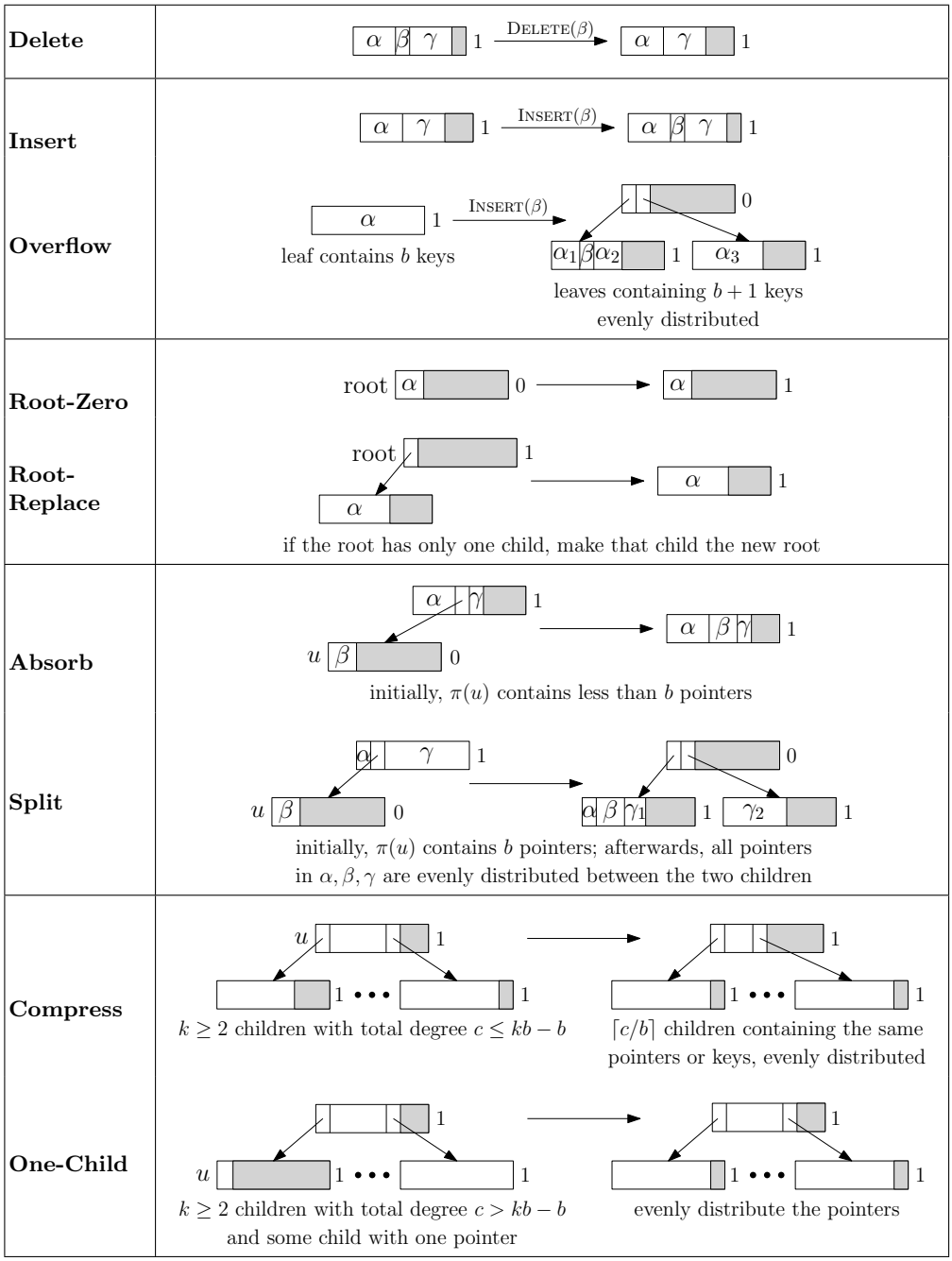

Fig. 1. Updates to B-slack trees (and relaxed B-slack trees). Nodes with weight zero contain exactly two pointers.

internal node. Additionally, since the new internal node contains $b - 2$ slack, whereas u contained no slack, there may be a slack violation at $\pi(u)$.

Delete, Insert and Overflow maintain the properties of a relaxed B-slack tree. They will also maintain the properties of a B-slack tree, provided that rebalancing steps are performed to remove any violations that are created.

3.3 Rebalancing steps

There are six different rebalancing steps for relaxed B-slack trees: Root-Zero, Root-Replace, Absorb, Split, One-Child and Compress. If there is a degree violation at the root, then Root-Replace is performed. If not, but there is a weight violation at the root, Root-Zero is performed. If there is a weight violation at an internal node that is not the root, then Absorb or Split is performed. Suppose there are no weight violations. If there is a degree violation at a node u and no degree or slack violation at $\pi(u)$, then One-Child is performed. If there is a slack violation at a node u and no degree violation at u, then Compress is performed. Figure 1 illustrates these steps. The goal of rebalancing is to eliminate all violations, while maintaining the relaxed B-slack tree properties.

Root-Zero. Root-Zero changes the weight of the root from zero to one, eliminating a weight violation, and incrementing the relaxed depth of every node. If P1′ held before Root-Zero, it holds afterwards.

Root-Replace. Root-Replace replaces the root r by its only child u, and sets u's weight to one. This eliminates a degree violation at r, and any weight violation at u. If u had weight zero before Root-Replace, then the relaxed depth of every leaf is the same before and after Root-Replace. Otherwise, the relaxed depth of every leaf is decremented by Root-Replace. In both cases, if P1′ held before Root-Replace, it holds afterwards.

Absorb. Let u be a non-root node with weight zero. Absorb is performed when $\pi(u)$ contains less than b pointers. In this case, the two pointers in u are moved into $\pi(u)$, and u is removed from the tree. Since the pointer from $\pi(u)$ to u is no longer needed once u is removed, $\pi(u)$ now contains at most b pointers. The only node that was removed is u and, since it had weight zero, the relaxed depth of every leaf remains the same. Thus, if P1′ held before Absorb, it also holds afterwards. Absorb eliminates a weight violation at u, but may create a slack violation at $\pi(u)$.

Split. Let u be a non-root node with weight zero. Split is performed when $\pi(u)$ contains exactly b pointers. In this case, there are too many pointers to fit in a single node. We create a new node v with weight one, and evenly distribute all of the pointers and keys of u and $\pi(u)$ (except for the pointer from $\pi(u)$ to u) between u and v. Now $\pi(u)$ has two children, u and v. The weight of u is set to one, and the weight of $\pi(u)$ is set to zero. As above, this does not change the relaxed depth of any leaf, so P1′ still holds after Split. Split moves a weight violation from u to $\pi(u)$ (closer to the root, where it can be eliminated by a Root-Zero or Root-Replace), but may create slack violations at u and v.

Compress. Compress is performed when there is a slack violation at an internal node u, there is no degree violation at u, and there are no weight violations

at u or any of its $k \geq 2$ children. Let $c \leq kb - b$ be the number of pointers or keys stored in the children of u. Compress evenly distributes the pointers or keys contained in the children of u amongst the first $\lceil c/b \rceil$ children of u, and discards the other children. This will also eliminate any degree violations at the children of u if $c > 1$. After the update, u satisfies P4. Compress does not change the relaxed depth of any node, so P1′ still holds after. Compress removes at least one child of u, so it increases the slack of u by at least one, possibly creating a slack violation at $\pi(u)$. (However, it decreases the total amount of slack in the tree by at least $b - 1$.) Thus, after a Compress, it may be necessary to perform another Compress at $\pi(u)$. Furthermore, as Compress distributes keys and pointers, it may move nodes with different parents together, under the same parent. Even if two parents initially satisfied P4 (so the children of each parent contain a total of less than b slack), the children of the combined parent may contain b or more slack, creating a slack violation. Therefore, after a Compress, it may also be necessary to perform Compress at some of the children of u.

One-Child. One-Child is performed when there is a degree violation at an internal node u, there are no weight violations at u or any of its siblings, and there is no violation of any kind at $\pi(u)$. Let k be the degree of $\pi(u)$. Since there is no slack violation at $\pi(u)$, there are a total of $c > kb - b = b(k - 1)$ pointers stored in u and its siblings. Since u has only one child pointer, each of its other $k - 1$ siblings must contain b pointers. One-Child evenly distributes the keys and pointers of the children of $\pi(u)$. One-Child does not change the relaxed depth of any node, so P1′ still holds after. One-Child eliminates a degree violation at u, but, like Compress, it may move children with different parents together under the same parent, possibly creating slack violations at some children of $\pi(u)$. So, it may be necessary to perform Compress at some of the children of $\pi(u)$.

All of these updates maintain P0′, P2′ and P3. While rebalancing steps are being performed to eliminate the violation created by an insertion or deletion, there is at most one node with weight zero.

We prove that a rebalancing step can be applied to any relaxed B-slack tree that is not a B-slack tree.

Lemma 1 *Let T be a relaxed B-slack tree. If T is not a B-slack tree, then a rebalancing step can be performed.*

Proof. If T is not a B-slack tree, it contains a weight violation, a slack violation or a degree violation. If there is weight violation, then Root-Zero, Absorb or Split can be performed. Suppose there are no weight violations. Let u be the node at the smallest depth that has a slack or degree violation. Suppose u has a degree violation. If u is the root, then Root-Replace can be performed. Otherwise, $\pi(u)$ has no violation, so One-Child can be performed. Suppose u does not have a degree violation. Then, u must have a slack violation, and Compress can be performed. □

4 Analysis

Due to space constraints, this section merely gives an outline of results proved about B-slack trees. In the full version of the paper, we provide a detailed analysis of B-slack trees that store n keys, by giving: an upper bound on the height of the tree, a lower bound on the average degree of nodes (and, hence, utilization), and an upper bound on the space complexity.

Arbitrary B-slack trees are difficult to analyze, so we begin by studying a class of trees called b-overslack trees. A b-overslack tree has a root with degree two, and satisfies P1, P2 and P3, but instead of P4, the children of each internal node contain a total of exactly b slack. Thus, a b-overslack tree is a relaxed B-slack tree, but not a B-slack tree. Consider a b-overslack tree T of height h that contains n keys. We prove that the total degree at depth $\delta \leq h$ in T is $d(\delta) = 2^{-\delta}(\alpha^\delta + \gamma^\delta)$, where $\alpha = b + \sqrt{b^2 - 4b}$ and $\gamma = b - \sqrt{b^2 - 4b}$. Since the total degree at the lowest depth is precisely the number of keys in the tree, every b-overslack tree of height h contains exactly $d(h)$ keys. Furthermore, when $h \geq 3$, we also have $(b - 1.4)^h < (b - \frac{\gamma}{2})^h \leq d(h) \leq b^h$. Therefore, for $n > b^3$ (which implies height at least three), h satisfies $\lceil \log_b n \rceil \leq h \leq \lceil \log_{b-\gamma/2} n \rceil < \lceil \log_{b-1.4} n \rceil$. We also prove that the average degree of nodes in T is $\frac{b \cdot d(h-1) - b + 2}{b \cdot d(h-2) - b + 3}$, which is greater than $b - 1.4$ for $h \geq 3$.

We next prove some connections between overslack trees and B-slack trees. First, we show that each b-overslack tree of height h has a smaller total degree of nodes at each depth than any B-slack tree of height h. We do this by starting with an arbitrary B-slack tree of height h, and repeatedly removing pointers and keys from the children of each internal node that satisfies P4 (taking care not to violate P1, P2 or P3), until we obtain a b-overslack tree. It follows that each b-overslack tree of height h contains fewer keys than any B-slack tree of height h. Consequently, every b-overslack tree with n keys has height at least as large as any B-slack tree with n keys. We next prove that every b-overslack tree of height h has a smaller average node degree than any B-slack tree of height h. As above, the proof starts with an arbitrary B-slack tree of height h, and removes pointers and keys from nodes until the tree becomes a b-overslack tree. However, in this proof, every time we remove a pointer, we must additionally show that the average degree of nodes in the tree decreases.

We then compute the *space complexity* of a B-slack tree containing n keys, which is the number of words needed to store it. Consider a leaf-oriented tree with maximum degree b. For simplicity, we assume that each key and each pointer to a child or data occupies one word in memory. Thus, a leaf occupies $2b$ words, and an internal node occupies $2b - 1$ words. A memory block size of $2b$ is assumed. Let $\bar{D}$ be the average degree of nodes. Then, $U = \bar{D}/b$ is the proportion of space that is utilized (which we call the *average space utilization* of the tree), and $1 - U$ is the proportion of space that is wasted. The space complexity is $2bF$, where F is the number of nodes in the tree. Suppose the tree contains n keys. By definition, the sum of the degrees of all nodes is $F - 1 + n$, since each node, except the root, has a pointer into it and the degree of a leaf is the number of

keys it contains. Additionally, $F\bar{D}$ is equal to the sum of degrees of all nodes, so $F = (n-1)/(\bar{D}-1)$. Therefore, the space complexity is $2b(n-1)/(\bar{D}-1)$. In order to compute an upper bound on the space complexity for a B-slack tree of height h, we simply need a lower bound on $\bar{D}$. Above, we saw that $\bar{D} > b-1.4$ for B-slack trees of height at least three. It follows that a B-slack tree with $n > b^3$ keys has space complexity at most $2b(n-1)/(b-2.4) < \frac{2b}{b-2.4}n$.

The full version of the paper describes pathological families of B-trees, Overflow trees and H-trees, and compares the space complexity of example trees in these families with the worst-case upper bound on the space complexity of a B-slack tree. By studying these families, we obtain lower bounds on the space complexity of these trees that are above the upper bound for B-slack trees.

We also study the number of rebalancing steps necessary to maintain balance in a relaxed B-slack tree. Consider a relaxed B-slack tree obtained by starting from a B-slack tree containing n keys and performing a sequence of i insertions and d deletions. We prove that such a relaxed B-slack tree will be transformed back into a B-slack tree after at most $2i(2 + \lfloor \log_{\lfloor \frac{b}{2} \rfloor}(n+i)/2 \rfloor) + d/(b-1)$ rebalancing steps, irrespective of which rebalancing steps are performed, and in which order. Hence, insertions perform amortized $O(\log(n+i))$ rebalancing steps and deletions perform an amortized constant number of rebalancing steps.

5 Conclusion

We introduced B-slack trees, which have excellent space complexity in the worst case, and amortized logarithmic updates. The data structure is simple, requires only one block size, and is well suited for hardware implementation.

Modifying the definition of B-slack trees so that the total slack shared amongst the children of each internal node of degree k is at most $b+k-1$, instead of $b-1$, yields a data structure with amortized constant rebalancing (with small constants), and only a slight increase in space complexity. Specifically, such a tree containing $n > b^3$ keys occupies at most $\frac{2b}{b-3.4}n$ words. Details appear in the full version of the paper.

The recently introduced technique of Brown, Ellen and Ruppert [6] can be used to obtain a concurrent implementation of relaxed B-slack trees that tolerates process crashes and guarantees some process will always make progress. In the resulting implementation, localized updates to disjoint parts of the tree can proceed concurrently, and searches can proceed without synchronizing with updates, which makes them extremely fast. The implementation can be designed such that, in a quiescent state, when no updates are in progress, the data structure is a B-slack tree.

B-slack trees have been implemented in Java, and code is freely available from http://www.cs.utoronto.ca/~tabrown. Experiments have been performed to validate the theoretical worst-case bounds, and to better understand the level of pessimism in them. The results indicate that few rebalancing steps are performed in practice, and average degree is somewhat better than the already good worst-case bounds. For instance, for $b = 16$ and $b = 32$, over a variety

of simulated random workloads with tree sizes varying between $2^5 = 32$ and $2^{20} = 1{,}048{,}576$ keys, there were at most 1.2 rebalancing steps per insertion or deletion, and average degrees for trees were approximately $b - 0.5$, which is extremely close to optimal.

Acknowledgments This work was dramatically improved by the insightful comments of my supervisor, Faith Ellen.

References

1. D. M. Arnow and A. M. Tenenbaum. An empirical comparison of B-trees, compact B-trees and multiway trees. In *ACM SIGMOD Record*, volume 14, pages 33–46. ACM, 1984.
2. D. M. Arnow, A. M. Tenenbaum, and C. Wu. P-trees: Storage efficient multiway trees. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 111–121. ACM, 1985.
3. R. A. Baeza-Yates and P.-A. Larson. Performance of B+-trees with partial expansions. *Knowledge and Data Eng., IEEE Transactions on*, 1(2):248–257, 1989.
4. R. Bayer and E. McCreight. Organization and maintenance of large indexes. Technical Report D1-82-0989, Boeing Scientific Research Laboratories, 1970.
5. H. Brönnimann, J. Katajainen, and P. Morin. Putting your data structure on a diet. *CPH STL Rep*, 1, 2007.
6. T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.
7. K. Culik II, T. Ottmann, and D. Wood. Dense multiway trees. *ACM Transactions on Database Systems (TODS)*, 6(3):486–512, 1981.
8. S.-H. S. Huang. Height-balanced trees of order (β, γ, δ). *ACM Trans. Database Syst.*, 10(2):261–284, June 1985.
9. K. Kspert. Storage utilization in B*-trees with a generalized overflow technique. *Acta Informatica*, 19(1):35–55, 1983.
10. K. Larsen, E. Soisalon-Soininen, and P. Widmayer. Relaxed balance through standard rotations. In F. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Algorithms and Data Structures*, volume 1272 of *Lecture Notes in Computer Science*, pages 450–461. Springer Berlin Heidelberg, 1997.
11. A. L. Rosenberg and L. Snyder. Compact B-trees. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 43–51, New York, NY, USA, 1979. ACM.
12. B. Srinivasan. An adaptive overflow technique to defer splitting in B-trees. *The Computer Journal*, 34(5):397–405, 1991.