On Optimal Balance in B-Trees: What Does It Cost to Stay in Perfect Shape?

Rolf Fagerberg
University of Southern Denmark, Odense, Denmark
rolf@imada.sdu.dk

David Hammer
Goethe University Frankfurt, Germany
University of Southern Denmark, Odense, Denmark
hammer@imada.sdu.dk

Ulrich Meyer
Goethe University Frankfurt, Germany
umeyer@ae.cs.uni-frankfurt.de

Abstract

Any B-tree has height at least $\lceil \log_B(n) \rceil$. Static B-trees achieving this height are easy to build. In the dynamic case, however, standard B-tree rebalancing algorithms only maintain a height within a constant factor of this optimum. We investigate exactly how close to $\lceil \log_B(n) \rceil$ the height of dynamic B-trees can be maintained as a function of the rebalancing cost. In this paper, we prove a lower bound on the cost of maintaining optimal height $\lceil \log_B(n) \rceil$, which shows that this cost must increase from $\Omega(1/B)$ to $\Omega(n/B)$ rebalancing per update as n grows from one power of B to the next. We also provide an almost matching upper bound, demonstrating this lower bound to be essentially tight. We then give a variant upper bound which can maintain near-optimal height at low cost. As two special cases, we can maintain optimal height for all but a vanishing fraction of values of n using $\Theta(\log_B(n))$ amortized rebalancing cost per update and we can maintain a height of optimal plus one using $O(1/B)$ amortized rebalancing cost per update. More generally, for any rebalancing budget, we can maintain (as n grows from one power of B to the next) optimal height essentially up to the point where the lower bound requires the budget to be exceeded, after which optimal height plus one is maintained. Finally, we prove that this balancing scheme gives B-trees with very good storage utilization.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases B-trees, Data structures, Lower bounds, Complexity

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.35

Funding *Rolf Fagerberg*: Supported by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041.
David Hammer: Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.
Ulrich Meyer: Supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/3-2 and ME 2088/4-2.

1 Introduction

1.1 Motivation

B-trees are search trees particularly suited for data organization in external memory. They are widely employed in database systems and file systems [12] and since their introduction in 1972 by Bayer and McCreight [7], they have been the subject of much theoretical and practical study.

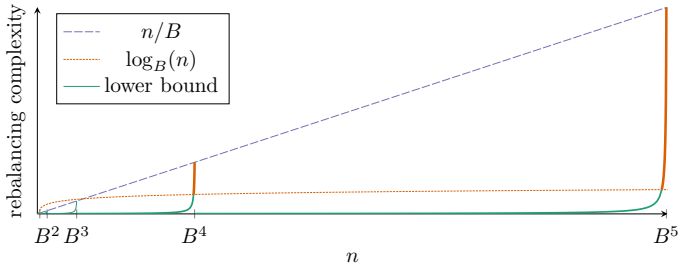


Figure 1 Plot illustrating how the lower bound for rebalancing cost increases as n approaches different powers of B. Intervals where the cost exceeds $\log_B(n)$ are highlighted in orange.

A standard B-tree of order B is a search tree where all leaves are on the same level and every internal node has between $\lceil B/2 \rceil$ and B children except for the root which may have any number of children between 2 and B. One common generalization is (a, b)-trees [16] where internal nodes have between a and b children for $2 \leq a \leq \lceil b/2 \rceil$. The standard operations for rebalancing B-trees are *split* of an overfull node into two nodes (increasing the degree of the parent by one), *merge* which is the reverse of split, and *share* which is a redistribution of keys among neighboring nodes.

In search trees, the cost of an operation is measured as the number of nodes accessed. For a search operation, this cost is determined by the height of the tree. For any search tree with maximal fanout B, $\lceil \log_B(n) \rceil$ is a lower bound on its height. In the static case, this height is easily achieved by a bottom-up linear (that is, $O(n/B)$ assuming presorted keys) cost construction method. In the dynamic case, standard B-trees maintain an upper bound of $\lfloor 1 + \log_{\lceil B/2 \rceil}(n/2) \rfloor$ on their height using $O(\log_B(n))$ rebalancing cost per insertion and deletion. Using (a, b)-trees, the amortized rebalancing cost per update can be reduced to $O(1)$ for $a = \lfloor b/2 \rfloor$ and to $O(1/b)$ for $a = b/4$, at the price of a moderate increase in the height bound [16, 22].

The upper bound of $\lfloor 1+\log_{\lceil B/2 \rceil}(n/2) \rfloor$ on the height is a constant factor of approximately $\log B/(\log B - 1)$ away from the lower bound. The optimal bound $\lceil \log_B(n) \rceil$ can of course be achieved by spending linear cost on a rebuilding after each update, but this cost is prohibitive in most situations and begs the question: can optimal height be maintained more cheaply?

Intuitively, maintaining optimal height should become harder as n approaches the next power of B, since the amount of available space in the tree decreases, giving less flexibility during rebalancing. Or put differently, the number of different trees of optimal height shrinks as n increases – when reaching a power of B, there is only one such tree. Overall, we can expect that there must be a trade-off between how full the tree is and how costly rebalancing to optimal height will be. The goal we pursue in this paper is to find this trade-off. More generally, we would like to know which height bounds can be maintained at which costs – a correlation which may be called the intrinsic rebalancing cost of B-trees.

1.2 Our contributions

For any n, let N denote the next power of B (i.e., $N = B^{\lceil \log_B n \rceil}$) and define ϵ by $n = N(1-\epsilon)$. Our first main result is a lower bound showing that for any B-tree of optimal height, there exists an insertion forcing $\Omega(1/(\epsilon B))$ nodes to be rebalanced before the tree can again have optimal height. This expression describes how the rebalancing cost must change from $\Omega(1/B)$ to $\Omega(n/B)$ as n approaches the next power of B. See Figure 1 for a visualization of this bound.

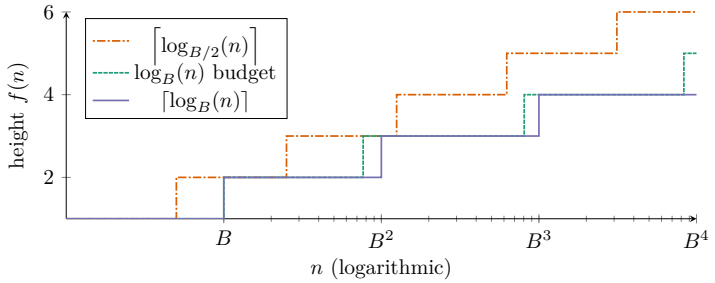


Figure 2 Plot showing different B-tree heights as a function of n. The functions represent the height of standard B-trees, the height achieved by our new scheme using a rebalancing budget of $O(\log_B(n))$, and the optimal height bound. For this particular plot, $B = 10$.

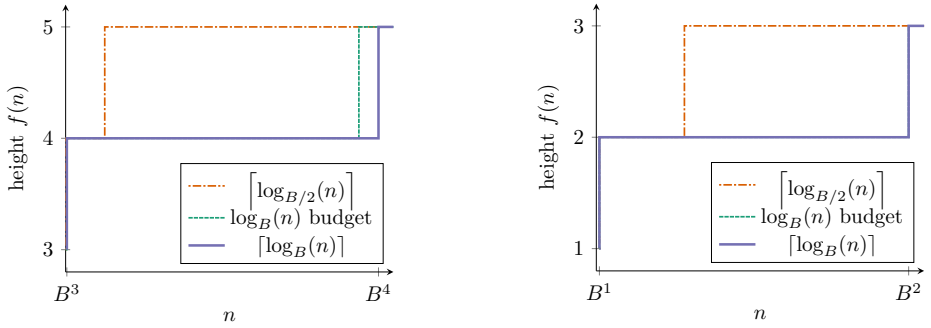
Our second main result is an almost matching upper bound, which maintains optimal height using amortized $O(\log^2 B/(\epsilon B))$ rebalancing per update, thereby proving the lower bound essentially tight.

This lower bound (and hence the upper bound) approaches linear cost as n approaches the next power of two. As our third main result, we give a variant rebalancing scheme that allows almost optimal height at much lower amortized cost. More precisely, for any rebalancing budget $f(n)$, we can maintain optimal height $\lceil \log_B(n) \rceil$ as n approaches the next power of B essentially up to the point where the lower bound result requires the budget to be exceeded, after which height $\lceil \log_B(n) \rceil + 1$ is maintained. To our knowledge, this is the first rebalancing scheme for B-trees with a height bound whose difference compared to optimal is an additive constant rather than a multiplicative constant.

One natural choice of budget is $f(n) = \log_B(n)$ as this matches the search cost. Figure 2 illustrates the height bound of this new scheme, the height bound of standard B-trees, and the optimal height bound. As is hinted by the figure, the fraction of values of n for which this scheme does not maintain optimal height $\lceil \log_B(n) \rceil$ is actually vanishing for growing n. Since in real life uses of B-trees in external memory the values of B are large and realistic values of n are fairly bounded in terms of possible powers of B, one may also want to look at the concrete improvements in height bounds for some practically occurring values of B and n. In database systems and file systems there are two main regimes, often termed OLTP and OLAP in the database setting. In the former, $B = 256$ is a typical value, in the latter, $B = 10^6$ is a typical value. For $B = 256$, the expression $B^3 < M \ll n \lesssim B^4$ (where M denotes the number of keys that can be held in internal memory) describes many real situations. Figure 3a repeats the plot of Figure 2 for the interval $B^3 < n < B^4$, but with $B = 256$ and the horizontal axis linear (not logarithmic). Standard B-trees and our scheme achieve the optimal height bound 4 for some part of the interval, but that part differs significantly in size: 12.16% versus 93.71%. For $B = 10^6$, the expression $B^1 < M \ll n \lesssim B^2$ describes many real situations. As illustrated by Figure 3b, the part of the interval $B^1 < n < B^2$ where optimal height is guaranteed is here 27.49% and 99.98% for the two schemes.

Another natural choice of budget is $f(n) = O(1/B)$ as this is the best possible amortized rebalancing cost (we can always make one node overflow at least every B insertions). With this budget, our scheme maintains height at most $\lceil \log_B(n) \rceil + 1$ for all values of n.

Finally, we prove that our near-optimal balancing scheme gives B-trees with very good storage utilization. Storage utilization is the fraction of the total space in the nodes allocated which is occupied by tree pointers, keys, and elements, i.e., the average use of the space in a node. High storage utilization is a desirable quality since it allows more of the tree to be cached in main memory and it has been the topic of much previous literature on B-trees, in particular in the database setting.



(a) $B = 256$. Standard B-trees are optimal in 12.16% of the interval, our scheme is in 93.71% of the interval.

(b) $B = 10^6$. Standard B-trees are optimal in 27.49% of the interval, our scheme is in 99.98% of the interval. Note: two of the plots are essentially coincident.

Figure 3 Plots showing different B-tree heights as a function of n similar to Figure 2 but for two specific, realistic settings. The horizontal axes are non-logarithmic to more faithfully show proportions.

1.3 Previous work

Not much work has been done with an explicit focus on the height of B-trees. The concept of storage utilization is rather closely related, since optimizing the height requires increasing the average fanout of nodes, which again increases storage utilization. A number of previous results present different trade-offs between update complexity and storage utilization, and we now survey these. Standard B-trees have a worst-case storage utilization of approximately $1/2$ and a logarithmic rebalancing cost. It is well known that lowering the worst-case storage utilization of B-trees slightly by using (a, b)-trees with $a < \lceil b/2 \rceil$ allows for amortized constant rebalancing costs [22, 16]. The average storage utilization may be somewhat better than the worst-case. In [26], an analysis of 2–3 trees with random insertions suggests that such trees tend towards an expected storage utilization of $\ln 2 \approx 0.69$.

To improve the storage utilization in dynamic B-trees, Bayer and McCreight [7] proposed an *overflow* technique: full nodes share their load with their siblings (when possible) instead of splitting (Knuth [18] refers to this variant as B*-trees). This improves the storage utilization in the worst case from $1/2$ to $2/3$, and the height bound from $\lfloor 1 + \log_{\lceil B/2 \rceil}(n/2) \rfloor$ to $\lfloor 1 + \log_{\lceil 2B/3 \rceil}(n/2) \rfloor$, at the price of increasing the update cost by a constant factor. As suggested in [7], this approach can be generalized to redistribution in bigger groups of nodes before splitting, thus further improving the storage utilization and the height bound at the cost of even higher rebalancing costs. However, no matter the group size, the height bound will be some constant factor away from optimal. The average storage utilization in a randomized setting using this approach is analyzed in [19], indicating that storage utilization converges to $m \ln((g+1)/g)$ for random trees under insertions where g is the overflow group size.

Similarly, an overflow scheme is tested in [25] which attaches overflow nodes to groups of nodes to delay splitting. They compute the average storage utilization to be $2g/(2g+3)$ when each overflow node is shared by a group of up to g leaf nodes. Both theoretical analysis and practical simulations show improved storage utilization at a cost of increased complexity during updates.

Rosenberg and Snyder [24] introduce so-called *compact B-trees* which are node-oriented trees shown to be close to optimal with respect to access costs while requiring minimal space. However, it is essentially a static structure as compaction incurs linear cost and

no faster compactness-preserving update algorithm is known. The empirical analysis in a dynamic setting presented in [6] indeed shows that insertion into compact B-trees is very costly and that, without compaction, storage utilization rapidly degrades when the number of updates grows.

Recently, Brown [10, 11] has developed a practically motivated variation called *B-slack trees* which has amortized logarithmic update complexity while achieving high storage utilization. *Slack* for a node refers to the difference between its maximum degree and actual degree (number of children "missing"). Special to B-slack trees is that the children of any internal node have a combined slack of at most $B-1$ where $B>4$ is the maximum degree of nodes. Maintaining this property is shown to ensure that a tree with n keys occupies at most $\frac{2B}{B-3.4}n$ words (which approaches $2n$ – the optimal value given the model used – as B increases). The rebalancing cost is amortized logarithmic per update.

It is worth mentioning key compression techniques like those found in prefix B-trees [8] as an approach to improve fanout and thus potentially lower tree height. This and similar key compression techniques represent an orthogonal line of research which we do not pursue.

In contrast to B-trees, there for binary trees is a much stronger tradition for focusing on the height. In particular, there is a body of work [23, 3, 1, 2, 4, 5, 21, 20, 14, 15] focusing on rebalancing schemes with height bounds close to the optimal value $\lceil \log_2(n) \rceil$. The end result of this line of investigation is the matching upper and lower bounds in [14, 15]. Our results in this paper can be seen as a generalization of those methods and results to the case of B-trees – setting $B=2$ in our bounds gives those of [14, 15]. One core new ingredient is the methods of Section 4.4 which are necessary for the upper bound to almost match the loss of a factor of B in the lower bound in Section 3 compared to the bound in [15].

2 Model

We describe our results for leaf-oriented B-trees, as these are standard in the literature. In leaf-oriented B-trees, internal nodes contain a total of $2B-1$ fields: B pointers to subtrees and $B-1$ search keys to guide the search (consistent with B-trees of order B as defined by Knuth [18]). Both keys and pointers can be nil. Since keys separate subtrees, the number of non-nil pointers is equal to the number of non-nil keys plus one. Leaf nodes contain the actual elements, including their search keys. For simplicity, we assume that leaf nodes contain up to B elements.[1] All leaf nodes appear at the same level.

A B-tree of height h can contain up to $N=B^h$ elements. We let T denote a B-tree with n elements and optimal height $h=\lceil \log_B n \rceil$ and we define ϵ by $n=N(1-\epsilon)=B^h(1-\epsilon)$. Since the height is optimal, we have $B^{h-1}<n\leq B^h$ and hence $0\leq\epsilon<(B-1)/B$. A node is said to be modified by an update operation if any of its fields are changed. Clearly the number of modified nodes is a lower bound on the rebalancing cost of an update operation. Unlike in standard B-trees, we do not impose a lower bound on the number of keys. This only makes our lower bound stronger and it thus applies to standard B-trees. Our upper bounds can easily be adapted to conform to a lower bound on node contents.

[1] In practice, the size of elements relative to the size of keys may vary between data sets (and leaves may also store pointers to elements instead of elements themselves), but it is straight-forward to adapt our statements accordingly.

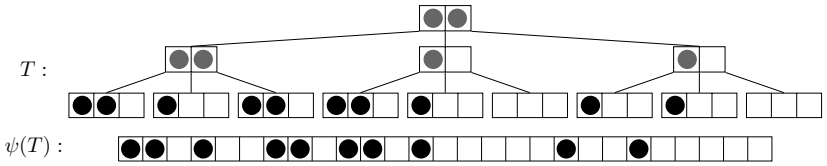


Figure 4 Illustration of the mapping of a tree T (with $B = 3$) into $\psi(T)$. Grey circles represent keys in internal nodes, black circles represent elements (in leaves).

3 Lower bound

In this section, we prove the following main theorem.

▶ **Theorem 1.** *For any B-tree T of optimal height $h = \lceil \log_B n \rceil$, there exists an insertion into T such that rebalancing T to optimal height after the insertion will require modifying $\Omega(1/(\epsilon B))$ nodes, where ϵ is given by $n = B^h(1 - \epsilon)$.*

The proof is based on creating a mapping from B-trees into arrays (by suitably generalizing a mapping for binary trees in [14]). The mapping allows us to exploit the existence in any array of a "uniformly dense" position, which will point to an update position in T for which we can prove the lower bound stated in Theorem 1.

3.1 Mapping into array

We create the mapping from elements of T to entries of an array $\psi(T)$ of length B^h as follows. For a full tree ($n = B^h$), this mapping is given by an inorder traversal which maps elements met during the traversal to increasing array entries. For a non-full tree, we embed T into a full tree of the same height before applying the mapping. We use the following specific embedding: keys fill up nodes from the left such that missing keys (and for internal nodes the corresponding missing subtrees) for non-full nodes will be on the right. This embedding and the resulting mapping is illustrated in Figure 4.

The mapping has the property that any subtree of the full tree is mapped to a contiguous interval in $\psi(T)$. In particular, for the full tree, a subtree with root at height h' spans an interval of size $B^{h'}$ in $\psi(T)$. This implies that if such a subtree of T is moved during a rebalancing, (say, if siblings of the parent of the subtree are added or removed in T), the positions in $\psi(T)$ of its nodes will be shifted by a multiple of $B^{h'}$.

We denote intervals of array indices (that is, intervals of integers) as $[a; b]$ and the length $b - a + 1$ of such intervals as $l([a; b])$. The following lemma regarding density of subsets of intervals is from Dietz et al. [13].

▶ **Lemma 2.** *For any two indices a and b and any $S \subseteq [a; b]$, there is an index $i \in [a; b]$ such that for any indices s and t where $a \leq s \leq i \leq t \leq b$, the following holds:*

$$\frac{|S \cap [s; t]|}{l([s; t])} \leq 2\frac{|S|}{l([a; b])}.$$

Consider the subset $S = \{j \in [a; b] \mid \psi(T)[j] \text{ is empty}\}$ where a and b are the endpoints of the array $\psi(T)$. Applying Lemma 2 with this S, a, and b shows the existence of an index i where *any* interval around i in $\psi(T)$ will have a density of "holes" (empty array entries) of at most two times the global density of holes in $\psi(T)$, i.e. of at most 2ϵ. For this i, setting $s = t = i$ and $s = i - 1$, $t = i + 1$ in Lemma 2 shows that $\psi(T)[i]$ and at least one of its direct neighbors are non-empty if $\epsilon < 1/3$. We insert a new element in the tree with a key x lying between $\psi(T)[i]$ and this neighbor. The rest of the proof assumes w.l.o.g. that $\psi(T)[i] < x < \psi(T)[i + 1]$.

3.2 Counting node changes

Consider any rebalancing operations ensuing from the insertion of x into the original tree T and let T' be the tree after rebalancing. Let all nodes in T modified by the rebalancing be colored blue and let unmodified nodes be colored white. We apply a recursive splitting procedure on all blue nodes layer for layer in a top-down manner. This procedure will maintain a set of *parts*. Each part is a tuple which contains a connected subgraph of T and a contiguous subset of the elements in $\psi(T)$ which we call a segment. As the subgraph in a part is connected, it has a unique highest node which will be called the root of the part.

Initially, the set of parts contains a single tuple consisting of T and a segment containing all elements of $\psi(T)$. Each splitting step on a blue node splits a part into new parts containing subsets of the original part.

We now describe a splitting step on a blue node v. Let T_v be the subtree of v and let S_v be the elements corresponding to the elements contained in the leaves of T_v mapped into $\psi(T)$. Let (G_p, S_p) be the existing part (where p is the root of G_p) containing v and its subtree. The existing part (G_p, S_p) is replaced by the following new parts.

For each subtree under v, add a new part consisting of the subtree and all elements in the subtree in $\psi(T)$. Refer to Figure 5 for an illustration of these subtree parts. When splitting

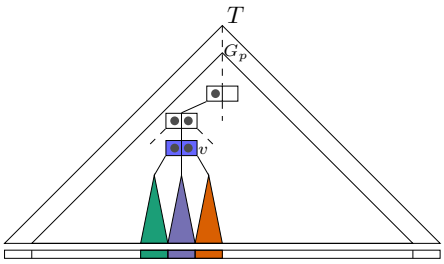
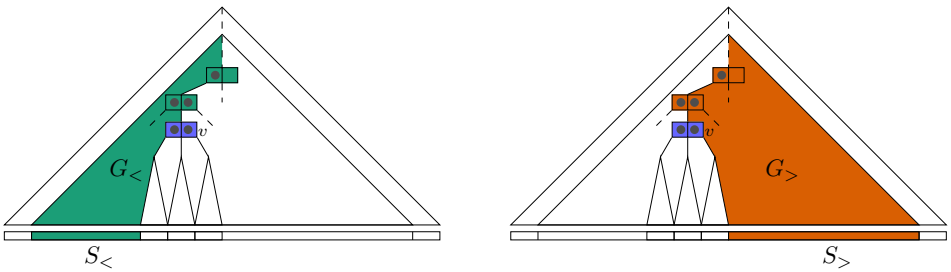


Figure 5 Illustrating of a general case of a splitting step on the node v. The subtree parts created in this step (both subgraphs and segments) have been highlighted.

blue leaf nodes, we create a part for each element stored in the node. Such a part will consist of an empty subgraph and a singleton segment containing the element. Generally, G_p may be a proper superset of T_v (see Figure 5). In this case, we create up to two boundary parts for $S_p \setminus S_v$. The segment $S_<$ for the left boundary part $(G_<, S_<)$ consists of all elements in S_p left of S_v. The subgraph $G_<$ consists of all nodes on the path from v to p and all nodes in G_p to the left of this path. Creating the right boundary part is done similarly. See Figure 6 for examples of these boundary parts. For later use, we note that this splitting procedure results in $O(B)$ segments per blue node.



(a) $(G_<, S_<)$. **(b)** $(G_>, S_>)$.

Figure 6 Illustration of the boundary parts of Figure 5.

The following simple invariants for the splitting procedure are easy to verify: 1) When processing layer k, no part will have a subgraph containing a blue node on any layer $k' > k$. 2) For each part (G, S) where $|S| > 1$, elements of S are in leaves of G. 3) Until the leaves are reached, the process does not create any parts with empty subgraphs.

After the splitting procedure, there will by 1) be no blue nodes within connected subgraphs of the remaining parts. This can be used to show that such subgraphs are not moved up or down.

▶ **Lemma 3.** *After splitting is done, for parts (G, S) with $|S| > 1$, G will appear in T and T' with the same height.*

Proof. By 3), G will be a non-empty subgraph consisting of white nodes. Hence, this subgraph must appear again in T'. Due to 2), G contains some leaves, hence they appear on the same height in both T and T' (all leaves appear on the same level, so G cannot have moved vertically between T and T'). ◀

We now focus on the segments of parts as the construction will impose restrictions on their ability to move around from $\psi(T)$ to $\psi(T')$. The previous lemma implies that segments can only be *shifted* when going from $\psi(T)$ to $\psi(T')$ (meaning that elements appearing in a segment will have the same relative positions to each other in both arrays). How far a segment is shifted can be bounded from below by its size.

▶ **Lemma 4.** *A segment containing s elements which has different position in $\psi(T)$ compared to $\psi(T')$ has moved by a distance of $\Omega(s)$.*

Proof. For segments of at most one element, the statement is obvious. Consider a part (G, S) with $s = |S| > 1$ and let v be the topmost node of G. By Lemma 3, G will be found in T', too, with v having the same height in both trees. As discussed in Section 3.1, nodes on a given height can only appear in specific positions. When v moves, it translates the elements of S by (at least) a multiple of the size (in number of elements in leaves) of a full subtree under v. By (2), this size is at least s. ◀

Proof of Theorem 1. Suppose that the newly inserted element x lies between two segments, S_{l-1} and S_l. As no holes were available, at least one of S_{l-1} and S_l must have moved in $\psi(T')$ compared to $\psi(T)$. Suppose w.l.o.g. that S_l has moved to the right (potentially along with some segments to the right of S_l). Let j be the index of the right-most element in the last consecutive segment S_r right of i which has moved to the right and let $\ell = l([i; j])$. Due to the choice of i, at most $2\epsilon\ell$ of the entries in $[i; j]$ are empty, so there are at least $(1 - 2\epsilon)\ell$ elements in this interval of $\psi(T)$. All of these elements are in the segments $S_l, S_{l+1}, \ldots, S_r$ by choice of S_r. Segments can only be translated if there is room in the array (available holes). Since E_{r+1} did not move to the right, this implies that none of the segments $S_l, S_{l+1}, \ldots, S_r$ can have moved more than $2\epsilon\ell$, so by Lemma 4, they can then only contain $O(2\epsilon\ell)$ elements each. This means that the number of segments $S_l, S_{l+1}, \ldots, S_r$ must be $\Omega\left(\frac{(1-2\epsilon)\ell}{2\epsilon\ell}\right) = \Omega\left(\frac{1}{\epsilon}\right)$. As each blue node accounts for $O(B)$ segments, this implies the existence of $\Omega(1/(\epsilon B))$ blue nodes. ◀

4 Upper bound

In this section, we present a rebalancing scheme for performing insertions into a B-tree containing $B^h(1-\epsilon)$ elements while maintaining optimal height h. The basic scheme is building on ideas from a rebalancing scheme for binary search tree in [15]. It may also be viewed as a (highly non-trivial) extension of the overflow technique analyzed in [19]. Adding ideas from density keeping algorithms [17], we arrive at the final scheme.

For convenience of notation in the proof, we introduce the parameter $k = 1/\epsilon$. Also, ϵ from now on denotes a value fixed over a sequence of updates (it will be used in such a way that it is never more than a constant factor from its previous meaning, defined by $n = B^h(1-\epsilon)$).

▶ **Theorem 5.** *For any ϵ with $0 < \epsilon < (B-1)/B$ there exists a rebalancing scheme for maintaining optimal height h in a B-tree while its size ranges between $(1-\epsilon)B^h$ and $(1-\epsilon/2)B^h$ which has an amortized rebalancing cost per update of*

$$O\left(\frac{k\log^2(\min\{k,B\})}{B}\right),$$

where $k = 1/\epsilon$.

In Section 4.1, we describe the initial setup of the scheme, and in Section 4.2 we describe its rebalancing operations. In Section 4.3, we show how these rebalancing operations can maintain the structure at an amortized rebalancing cost of $O(k)$ node updates per update. In Section 4.4, we combine the scheme with density keeping methods in order to lower the amortized rebalancing cost to $O(k\log^2(\min\{k,B\})/B)$. We focus on insertions, since rebalancing after deletions can be handled by simply running the operations in reverse. We assume $n = (1-\epsilon)B^h$ initially.

4.1 Layout

The main mechanism employed in the structure is the distribution and redistribution of *holes* in the tree. A hole in a node is simply a missing entry – for leaves, this means that an element field is NIL, for internal nodes this means that a search key field and a corresponding pointer field are NIL. We define the *weight* of a hole in a node of height h' to be $B^{h'}$, i.e., the capacity of a subtree of height h'. This is the number of elements missing from the full tree due to this hole.

Since we initially have $n = (1-\epsilon)B^h$, we must initialize the tree such that the sum of weights of all holes in the tree is ϵB^h. We call this sum our weight budget. On each level of the tree, we divide the nodes into horizontal *groups* of contiguous nodes. It will be an invariant of the rebalancing scheme that each group contains between 0 and B holes. Initially, this value is B. The bigger the groups, the more sparse the holes will be. On the leaf level we set the group size (the number of nodes in a group) to $\Theta(k)$. On the levels above, we double the group size for each new level. If we conceptually consider this to happen in the full tree of degree B, the B holes per group will on the leaf level correspond to a constant fraction of the weight budget. By tuning the exact group size on the leaf level, we make this fraction be at most $1/8$. While the weight of a hole increases by a factor B per level, the width of the layers in the full tree decreases by the same factor. Thus, doubling the group size means that the total weight of holes in a level decreases by a factor of two for each level. Hence, the level sums in the full tree form a geometrically decreasing series with total sum at most twice the weight of the leaf level, i.e., at most $1/4$ of the weight budget. Some level

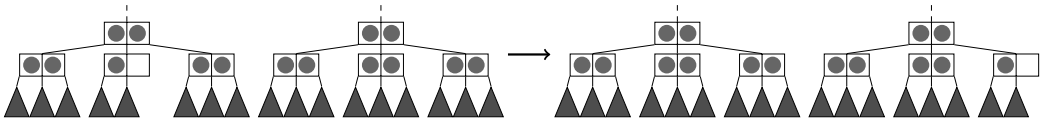


Figure 7 Illustration of horizontally sliding a hole within a (sub)tree.

$h_{\max}$ will be the last where the group size does not exceed the total number of nodes on that level. On this level, we place 3/4 of the total weight budget (arbitrarily positioned on the level). We will refer to this as the *reservoir*.

The actual initial layout is built top-down: Above $h_{\max}$ no weight is placed and all nodes have degree B. At level $h_{\max}$, 3/4 of the total weight budget is placed, which compared to the full tree removes nodes (by removing entire subtrees) on the levels below. The weight on the lower levels (which removes further nodes of the tree during the top-down process) is given by the group sizes defined above and the rule that each group has B holes. Due to the top-down removal of nodes in the tree, the resulting tree will be thinner than the full tree, hence the level sums in the actual tree produced will be smaller than the 1/4 of the total budget. Hence, after this top-down procedure, there is some budget of left. This is distributed evenly as holes on the leaf level (these holes are for simplicity of argumentation in proofs considered inactive, i.e., they will not take part in the rebalancing process described below and will not be considered in the invariant that groups have at most B holes). Once the shape of the tree has been produced, it is filled with elements and search keys in a bottom-up fashion. Assuming the elements given in sorted order (for instance during a global rebuild), the above process can be done at linear cost $O(n/B)$.

The rebalancing scheme we describe below works until there are no more holes in the reservoir. By the invariant on the number of holes per group in layers below the reservoir, the total weight outside of the reservoir will never exceed the fraction 1/4, so emptying the reservoir requires a number of insertions proportional to the initial weight in the reservoir. Hence, the scheme will last at least until size $(1 - \epsilon/2)B^h$, as required for Theorem 5.

4.2 Rebalancing operations

Two basic operations will be used in the rebalancing scheme to move holes around within the tree: *horizontal sliding* and *vertical redistribution*. As the names imply, horizontal sliding will move a hole from one location to another (within a group) on the same level while vertical redistribution moves a hole from one level to another.

The horizontal sliding procedure is illustrated in Figure 7. To efficiently access the nodes to be slid on the sliding level, we maintain horizontal level-pointers between nodes which are neighbors on the level. As groups do not necessarily align with subtree boundaries, neighbors on the level may have a lowest common ancestor which is further away than the sliding distance. Since this ancestor is among the nodes whose keys should be changed due to the slide (to maintain search tree order), we maintain pointers to these ancestors for all pairs of neighboring nodes on a level which are not siblings (these pointers and the level-pointers are not shown in Figure 7). When sliding, subtrees below the slide level must be moved along with the keys to maintain search tree order of the keys (as shown in Figure 7). In that process, the ancestor pointers between the edges of these subtrees may have to be updated, which for each subtree is a number of pointers proportional to its height. Thus, sliding a hole from another node in the current group on a level at height i will require accessing $O(i2^ik)$ nodes, since the group size is 2^ik.

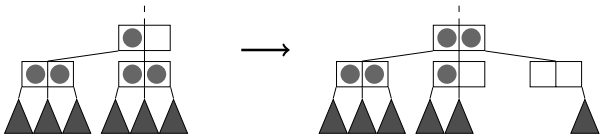


Figure 8 Illustrating the vertical redistribution. A hole is moved down one layer, thus allowing the creation of a new node on said layer. This corresponds directly to splitting in standard B-trees.

A vertical redistribution transforms one hole on a level $l+1$ into B holes on level l. It is illustrated in Figure 8. This operation is the same as the split operation in standard B-trees. To perform a vertical redistribution, one clearly needs to access less nodes than for a slide at the same level.

4.3 Insertion

We now describe how to insert a new element with a given key. As in standard B-trees, we first search for the key in the tree to find a leaf node, v. If v contains a hole, we simply add the element to v as in a standard B-tree and no further work is done. Otherwise, a hole is first moved to v to make room for the new element. This is done by a searching the group of v for a hole to slide to v. If none exists, we ask for the parent of v to obtain a hole, which we then can redistribute to v's level as B holes. If the parent has no hole, the process is repeated recursively. This recursive process, called REQUEST, is described as Algorithm 1. The recursion ends at the latest when the reservoir is reached.

Algorithm 1 REQUEST.

```
procedure REQUEST(v)
    if any node in v's group has a hole then
        slide this hole to v
    else
        REQUEST(parent node of v)
        redistribute a hole from parent
        if v's group is too big then
            split v's group
        end if
    end if
end procedure
```

Moving down holes from higher levels via a vertical redistribution increases the size of the receiving group by one (see Figure 8). To keep the sizes of groups, and hence the cost of sliding within groups, we split a group in two when a redistribution discovers that the size of the receiving group has doubled compared to its initial group size from Section 4.1. Groups are simply implemented by marking the border nodes of groups, so this splitting is straight-forward to carry out as part of the operation.

As each redistribution provides B holes to a group, Algorithm 1 will only recurse upwards (the **else** case) from a group when B new calls of Algorithm 1 to nodes in the group have been issued since the last recursion upwards from this group. Recall that a horizontal slide on a level at height i has a cost of $O(i2^i k)$ and that vertical redistributions are also covered by this bound. From this follows an amortized bound on the rebalancing cost of:

$$k + \frac{1 \cdot 2k}{B} + \frac{2 \cdot 2^2 k}{B^2} + \cdots + \frac{i \cdot 2^i k}{B^i} + \cdots + \frac{h_{\max} \cdot 2^{h_{\max}} k}{B^{h_{\max}}} \tag{1}$$

which is $O(k)$ (assuming $B > 2$). As is standard, this can formally be proven by a potential function which amounts to each insertion bringing an amount of "coins" equal to Equation (1). Coins are conceptually stored in groups and will pay for all rebalancing work. When splitting a group, the new group has an additional need for coins not covered by Equation (1). As groups grow slowly, the extra amount compared to Equation (1) is low order in its terms, so just doubling Equation (1) is more than enough.

4.4 Achieving log squared amortized rebalancing

We now describe how to modify the rebalancing procedure such that the amortized cost per insertion becomes:

$$O\left(\frac{k\log^2(\min\{k,B\})}{B}\right).$$

Note that the amortized cost of rebalancing on all non-leaf levels is already as low as $O\left(\frac{k}{B}\right)$. This can be seen by excluding the first term in Equation (1). Since $k = \Theta(1/\epsilon)$, that value matches the lower bound, so we only need to lower the cost of rebalancing on the leaf level.

The overall plan is to distribute holes more evenly within leaf groups, both when constructing the tree initially and when vertically redistributing holes from the next higher level. We will use a density keeping scheme [17] on the nodes within each leaf group. Concretely we will use the version described in section 3.1 of [9] (to which we refer for full details). The scheme maintains a distribution of holes in a binary search tree by enforcing density (number of keys relative to the maximum for a given height) bounds on the levels of the tree. In this scheme, each insertion will require redistribution within an interval of size $O\left(\log^2(n)/\tau_1\right)$ (amortized) where n is the size of the array and τ_1 is upper limit on the density for the array (τ_1 can be set to $1/2$ here).

We apply this scheme to each group of k leaves such that each leaf node within a group is treated like a leaf node in the binary tree of the density keeping scheme. The binary tree is implicitly overlaid on the group and not actually constructed. The analysis in [9] requires that redistributing everything in a subtree takes linear time in the size of the subtree. We point out that this requirement is satisfied in this setting as the number of nodes involved in rebuilding an interval on the leaf level will be dominated by the distance on the leaf level (as described regarding sliding in Section 4.3).

For $B \geq k$, we must ensure that a node will sustain $\Theta\left(B/k\right)$ insertions per request for more holes to achieve the desired amortized cost. To this end, we specify how many holes a node gets from its neighbors via a request (slide) and how many holes a node must have in order to be able to give some holes away (this specifies whether the density keeping scheme should consider the node full). We split the B/k holes in each node into two equal portions: one to handle insertions into the node and one to service a request from another node. This means that a node issues a request only after $\Theta\left(B/k\right)$ keys have been inserted into it and that a node is given $\Theta\left(B/k\right)$ holes when it issues a request for holes. A node can only help another node (potentially itself) once – this is also the case in the setting of [9] as each node has capacity one. The amortized number of insertions between each request within the group is now $\Omega(k/B)$ and hence the amortized cost is $O\left(\frac{k\log^2(k)}{B}\right)$.

For $k > B$, not all nodes can get a hole per vertical redistribution. Instead, we apply the density keeping scheme to subgroups of k/B nodes where each such subgroup gets one hole after a vertical redistribution. The distance to a hole within a subgroup is $O(k/B)$ and as there will be B subgroups per actual group, using the density keeping scheme will mean that insertions will have an amortized cost of $O\left(\frac{k\log^2(B)}{B}\right)$.

It is important to note that, in both cases, $\Theta(B)$ of the holes are used before a vertical redistribution is requested (this is necessary to cover the cost of global rebuilding).

To handle deletions, the operations can be performed in reverse, as mentioned earlier. This implies moving holes upwards in the tree (equivalent to merge in standard B-trees) when the number of holes in a group grows sufficiently big.

5 Global rebalancing scheme

We here describe how repeated global rebuilding can be used with the scheme presented in Section 4 to achieve a global rebalancing scheme (i.e., not bound to a specific range of tree sizes as in Theorem 5).

Immediately before performing a global rebuild, the tree contains $n = N(1-\epsilon)$ elements for some ϵ, where N is the next power of B. We set up the system from Section 4 by performing a global rebuild at linear cost (i.e., $O(n/B)$ node updates). We use this set-up while $N(1-2\epsilon) \leq n \leq N(1-\epsilon/2)$. Once one of these bounds have been reached, we rebuild again, updating ϵ by increasing or decreasing it by a factor of two. There will be at least $\Theta(N\epsilon)$ updates between each rebuilding. Thus, the amortized cost incurred by the rebuilding process per update will be $O(1/(B\epsilon))$, which does not increase the upper bound from Section 4.

Applying this scheme during insertions (with ϵ successively decreasing by factors of two along the way) allows us to maintain optimal height until the tree is arbitrarily full and the update cost hence is arbitrarily close to $\Theta(n/B)$. In practice, one would presumably choose to allow suboptimal height when the complexity of rebalancing exceeds a certain rebalancing budget. Figure 1 illustrates how the complexity (lower bound) behaves as n approaches powers of B and how this complexity lower bound relates to a logarithmic budget. Suppose we want to limit the update complexity to a given budget $f(n)$. We can do that by keeping optimal height from $n = B^{h-1}$ up to the point where the budget is exceeded and from that point up to $n = B^h$ allowing the height to be optimal plus one. Specifically, we define ϵ' by $k \log^2(\min\{k, B\})/B = f(n)$ and $k = 1/\epsilon'$. For most $f(n)$, this has for $B^{h-1} < n \leq B^h$ one solution ϵ' for each h, similarly to Figure 1. We assume this to be the case for the $f(n)$ in question. We adjust the scheme above in the following way:

- If n exceeds $B^h(1-\epsilon'/2)$, rebuild the tree with the height incremented by one and use $\epsilon = 1/2$.
- If n falls below $B^h(1-\epsilon')$, rebuild the tree with the height decremented by one and use $\epsilon = \epsilon'$.

This gives the result below, where ϵ' is defined as above.

▶ **Theorem 6.** *There exists a rebalancing scheme such that, given a rebalancing budget $f(n)$, a B-tree can be maintained with perfect height from B^{h-1} to $(1-\epsilon')B^h$ and with perfect height plus one up to B^h.*

We highlight two interesting special cases. The first case is $f(n) = \Theta(\log_B(n))$, which is a natural choice since it allocates the same cost for rebalancing as for searching time. For this choice, $\epsilon' = o(1)$, leading to:

▶ **Corollary 7.** *There exists a rebalancing scheme that given a rebalancing budget of $\Theta(\log_B(n))$ maintains a B-tree which has optimal height for all but a vanishing fraction of values of n. For the remaining values of n the height is optimal plus one.*

A comparison of the result of Corollary 7 to optimal height and to the height bound obtained by standard B-trees is given in Figure 2.

The second case is $f(n) = \Theta(1/B)$, for which we have $\epsilon' = \Theta(1)$, leading to:

▶ **Corollary 8.** *There exists a rebalancing scheme that given a rebalancing budget of* $\Theta(1/B)$ *maintains a B-tree with optimal height plus one.*

6 Storage utilization

We here consider the connection between the parameter ϵ and the storage utilization.

Our upper bound rebalancing scheme directly implies excellent storage utilization. When we lay out a tree with a given k this tree will have a storage utilization of at least $(k-1)/k = 1-\epsilon$. This is trivially true for the leaf level as each group consisting of a multiple of k leaves has at most one empty node worth of holes (refer to Section 4.1). As holes are more sparse on higher levels of the tree, these levels have higher storage utilization than the leaves. The total weight of holes in the reservoir is at most a constant times the total weight of holes in the leaves. Since the number of nodes allocated in the reservoir is bounded by the number of allocated leaves and the weight per hole in the reservoir will be greater (by a factor of a power of B), the storage utilization among nodes in the reservoir is at least that of the leaves, too.

The scheme in Section 5 changes ϵ and thus the guaranteed storage utilization while growing a tree. To get a consistent high storage utilization while growing a tree, one can instead choose a desired low ϵ' (lower than needed initially for $n = (1-\epsilon)N$), lay out the holes among the leaves and internal layers according to this ϵ', and leave the extra free storage this would yield in the reservoir. By the same arguments as before, the storage utilization on the leaf and inner levels of the tree would then be $1-\epsilon'$. On the reservoir level, one could compact the nodes such that at most one node would be non-full.

Our lower bound results do not translate directly into a statement about storage utilization. This result is about the amount of room (weight) in a tree of a particular height – the actual number of nodes allocated within the tree is not considered. As a counter-example to implications for storage utilization, take for instance a B-tree with all nodes completely full except the root which has only a tiny fraction of the B possible children. This tree will have excellent storage utilization – it will approach 1 for increasing n – while ϵ will be large as the number of keys could be increased massively without increasing tree height.

References

1 Arne Andersson. Optimal bounds on the dictionary problem. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüuller, J. Stoer, N. Wirth, and Hristo Djidjev, editors, *Optimal Algorithms*, volume 401, pages 106–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. doi:10.1007/3-540-51859-2_10.

2 Arne Andersson. *Efficient Search Trees.* PhD Thesis, Department of Computer Science, Lund University, Sweden, 1990.

3 Arne Andersson, Christian Icking, Rolf Klein, and Thomas Ottmann. Binary search trees of almost optimal height. *Acta Informatica*, 28(2):165–178, February 1990. doi:10.1007/BF01237235.

4 Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In John R. Gilbert and Rolf Karlsson, editors, *SWAT 90*, pages 111–121. Springer Berlin Heidelberg, 1990.

5 Arne Andersson and Tony W. Lai. Comparison-efficient and write-optimal searching and sorting. In Wen-Lian Hsu and R. C. T. Lee, editors, *ISA'91 Algorithms*, pages 273–282. Springer Berlin Heidelberg, 1991.

6 David M. Arnow and Aaron M. Tenenbaum. An empirical comparison of B-trees, compact B-trees and multiway trees. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, page 33. ACM Press, 1984. doi:10.1145/602259.602265.

7 R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, September 1972. doi:10.1007/BF00288683.

8 Rudolf Bayer and Karl Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977. doi:10.1145/320521.320530.

9 Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache Oblivious Search Trees via Binary Trees of Small Height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 39–48, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=545381.545386.

10 Trevor Brown. B-slack Trees: Space Efficient B-Trees. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, R. Ravi, and Inge Li Gørtz, editors, *Algorithm Theory – SWAT 2014*, volume 8503, pages 122–133. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-08404-6_11.

11 Trevor Brown. B-slack trees: Highly Space Efficient B-trees. *arXiv:1712.05020 [cs]*, December 2017. arXiv:1712.05020.

12 Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. doi:10.1145/356770.356776.

13 Paul F. Dietz, Joel I. Seiferas, and Ju Zhang. A tight lower bound for on-line monotonic list labeling. In Erik M. Schmidt and Sven Skyum, editors, *Algorithm Theory — SWAT '94*, pages 131–142. Springer Berlin Heidelberg, 1994.

14 Rolf Fagerberg. Binary search trees: How low can you go? In G. Goos, J. Hartmanis, J. Leeuwen, Rolf Karlsson, and Andrzej Lingas, editors, *Algorithm Theory — SWAT'96*, volume 1097, pages 428–439. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi:10.1007/3-540-61422-2_151.

15 Rolf Fagerberg. The complexity of rebalancing a binary search tree. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 72–83. Springer, 1999.

16 Scott Huddleston and Kurt Mehlhorn. Robust balancing in B-trees. In *Theoretical Computer Science*, pages 234–244. Springer, 1981.

17 Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 417–431. Springer Berlin Heidelberg, 1981.

18 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

19 Klaus Küspert. Storage utilization in B *-trees with a generalized overflow technique. *Acta Informatica*, 19(1), April 1983. doi:10.1007/BF00263927.

20 Tony W. Lai and Derick Wood. Updating almost complete trees or one level makes all the difference. In Christian Choffrut and Thomas Lengauer, editors, *STACS 90*, pages 188–194. Springer Berlin Heidelberg, 1990.

21 Tony Wen Hsun Lai. *Efficient Maintenance of Binary Search Trees*. PhD Thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1990.

22 David Maier and Sharon C. Salveter. Hysterical B-trees. *Information Processing Letters*, 12(4):199–202, August 1981. doi:10.1016/0020-0190(81)90101-0.

23 H.A. Maurer, Th. Ottmann, and H.-W. Six. Implementing dictionaries using binary trees of very small height. *Information Processing Letters*, 5(1):11–14, May 1976. doi:10.1016/0020-0190(76)90094-6.

24 Arnold L. Rosenberg and Lawrence Snyder. Compact B-trees. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, page 43. ACM Press, 1979. doi:10.1145/582095.582102.

25 Balasubramaniam Srinivasan. An Adaptive Overflow Technique to Defer Splitting in B-trees. *The Computer Journal*, 34(5):397–405, 1991. doi:10.1093/comjnl/34.5.397.

26 Andrew Chi-Chih Yao. On random 2–3 trees. *Acta Informatica*, 9(2):159–170, June 1978. doi:10.1007/BF00289075.