Space-Efficient B Trees via Load-Balancing

Tomohiro I[2][0000−0001−9106−6192] and Dominik Köppl[1][0000−0002−8721−4444]

[1] M&D Data Science Center, Tokyo Medical and Dental University, Japan
koeppl.dsc@tmd.ac.jp
[2] Department of Artificial Intelligence, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp

Abstract. We study succinct variants of B trees in the word RAM model that require $s + o(s)$ bits of space, where s is the number of bits essentially needed for storing keys and possibly other satellite values. Assuming that elements are sorted by keys (not necessarily in the order of their integer representations), our B trees support standard operations such as searching, insertion and deletion of elements. In some applications it is useful to associate a satellite value to each element, and to support aggregate operations such as computing the sum of values, the minimum/maximum value in a given range, or search operations based on those values. We propose a B tree representation storing n elements in $s + \mathcal{O}(s/\lg n)$ bits of space and supporting all mentioned operations in $\mathcal{O}(\lg n)$ time.

Keywords: B tree, succinct data structure, predecessor data structure

1 Introduction

A B tree [1] is the most ubiquitous data structure found for relational databases and is, like the balanced binary search tree in the pointer machine model, the most basic search data structure in the external memory model. A lot of research has already been dedicated for solving various problems with B trees, and various variants of the B tree have already been proposed (cf. [12] for a survey). Here, we study a space-efficient variant of the B tree in the word RAM model under the context of a dynamic predecessor data structure, which provides the following methods:

predecessor(K) returns the predecessor of a given key K (or K itself if it is already stored);
insert(K) inserts the key K; and
delete(K) deletes the key K.

We call these three operations *B tree operations* in the following. Nowadays, when speaking about B trees we actually mean B+ trees [4, Sect. 3] (also called *leaf-oriented B-tree* [2]), where the leaves store the actual data (i.e., the keys). We stick to this convention throughout the paper. Another variant we want to focus on in this paper is the B* tree [16, Sect. 6.2.4], where a node split on inserting a key into a full node has chances to be deferred by balancing the loads of this node with one of its siblings.

1.1 Related Work

The standard B tree as well its B+ and B* tree variants support the above methods in $\mathcal{O}(\lg n)$ time, while taking $\mathcal{O}(n)$ words of space for storing n keys. Even if each key uses only $k = o(\lg n)$ bits, the space requirement remains the same since its pointer-based tree topology already needs $\mathcal{O}(n)$ pointers. To improve the space while retaining the operational time complexity in the word RAM model is main topic of this article. However, this is not a novel idea:

The earliest approach we are aware of is due to Blandford and Blelloch [3] who proposed a representation of the leaves as blocks of size $\Theta(\lg n)$. Assuming that keys are integer of k bits, they store the keys not in their plain form, but by their differences encoded with Elias-γ code [7]. Their search tree takes $\mathcal{O}(n \lg((2^k + n)/n))$ bits while conducting B tree operations in $\mathcal{O}(\lg n)$ time.

More recently, Prezza [19] presented a B tree whose leaves store between $b/2$ and b keys for $b = \lg n$. Like [2, Sect. 3] or [6, Thm. 6], the main aim was to provide prefix-sums by augmenting each internal node of the B tree with additional information about the leaves in its subtree such as the sum of the stored values. Given m is the sum of all stored keys plus n, the provided solution uses $2n\,(\lg(m/n) + \lg\lg n + \mathcal{O}(\lg m/\lg n))$ bits of space and supports B tree operations as well as prefix-sum in $\mathcal{O}(\lg n)$ time. This space becomes $2nk + 2n \lg\lg n + o(n)$ bits if we store each key in plain k bits.

Data structures computing prefix-sums are also important for dynamic string representations [13, 17, 18]. For instance, He and Munro [13] use a B tree as underlying prefix-sum data structure for efficient deletions and insertions of characters into a dynamic string. If we omit the auxiliary data structures on top of the B tree to answer prefix-sum queries, their B tree uses $nk + \mathcal{O}(nk/\sqrt{\lg n})$ bits of space while supporting B tree operations in $\mathcal{O}(\lg n/\lg\lg n)$ time, an improvement over the $\mathcal{O}(\lg n)$ time of the data structure of González and Navarro [11, Thm. 1] sharing the same space bound. In the static case, Delpratt et al. [5] studied compression techniques for a static prefix-sum data structure.

Asides from prefix-sums, another problem is to maintain a set of strings, where each node v is augmented with the length of the longest common prefix (LCP) among all strings stored as satellite values in the leaves of the subtree rooted at v [9].

Next, there is a line of research on implicit data structures supporting B tree operations: Under the assumption that all keys are distinct, the data structure of Franceschini and Grossi [10] supports $\mathcal{O}(\lg n)$ time for $\mathsf{predecessor}$ and $\mathcal{O}(\lg n)$ amortized time for updates (delete and insert) while using only constant number of words of extra space to a dynamic array A of size kn bits storing the keys. However, they assume a more powerful model of computation, where expanding or contracting A at its end can be done in constant time. This model is more powerful in the sense that the standard RAM model only supports the reallocation of a new array and copying the contents of the old array to the new array, thus taking time linear in the size of the two arrays. In the standard RAM model, arrays with such operations (extension or contraction at their ends) are called *extendible arrays*, and the best solution in this model (we are aware of) uses $nk + \mathcal{O}(w + \sqrt{knw})$ bits of space for supporting constant-time access and constant-time amortized updates [20, Lemma 1]. Allowing duplicate keys, Katajainen and Rao [15] presented a data structure with the same time bounds as [10] but using $\mathcal{O}(n \lg \lg n / \lg n)$ bits of extra space.

With respect to similar techniques but different aim, we can point out the succinct dynamic tree representation of Farzan and Munro [8, Thm 2] who propose similar techniques like rebuilding substructures after a certain amount of updates (cf. Sect. 4.1), or storing satellite data in blocks (cf. Sect. 4). They also have a need for space-efficient prefix-sum data structures.

In what follows, we present a solution for B trees based on different known techniques for succinct data structures such as [20] and the aforementioned B tree representations.

1.2 Our Contribution

Our contribution (cf. Sect. 3) is a combination of a generalization of the rearrangement strategy of the B* tree with the idea to enlarge the capacity of the leaves similarly to some approaches listed in the related work. With these techniques we obtain:

Theorem 1. *There is a B tree representation storing n keys, each of k bits, in $nk + \mathcal{O}(nk/\lg n)$ bits of space, supporting all B tree operations in $\mathcal{O}(\lg n)$ time.*

We stress that this representation does not compress the keys, which can be advantageous if keys are not simple data types but for instance pointers to complex data structure such that equality checking cannot be done by merely comparing the bit representation of the keys, but still can be performed in constant time. In this setting of incompressible keys, the space of a *succinct* data structure supporting predecessor, insert, and delete is $nk + o(nk)$ bits for storing n keys.

We present our space-efficient B tree in Sect. 3. Additionally, we show that we can augment our B tree with auxiliary data such that we can address the prefix-sum problem and LCP queries without worsening the query time (cf. Sect. 4).

2 Preliminaries

Our computational model is the word RAM model with a word size of w bits. We assume that each key uses $k = \mathcal{O}(w)$ bits, and that we can compare two keys in $\mathcal{O}(1)$ time. More precisely, we support the comparison to be more complex than just comparing the k-bit representation bitwise as long as it can be evaluated within constant time. Let $n = \mathcal{O}(2^w) \cap \Omega((w \lg^2 n)/k)$ be the number of keys we store at a specific, fixed time.

A B+ tree of degree t for a constant $t \geq 3$ is a rooted tree whose nodes have an out-degree between $\lceil t/2 \rceil$ and t. See Fig. 1 for an example. All leaves are on the same height, which is $\Theta(\lg n)$ when storing n keys. The number of keys each leaf stores is between $\lfloor t/2 \rfloor$ and t (except if the root is a leaf). Each

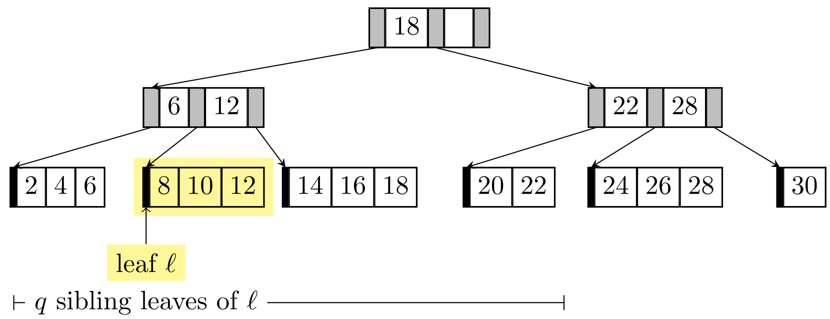

Fig. 1. A B+ tree with degree $t = 3$ and height 3. A leaf can store at most $b = t = 3$ children. A child pointer is a gray field in the internal nodes. An internal node v stores $t - 1$ integers in an array I_v where the value $I_v[i]$ regulates that only those keys of at most $I_v[i]$ go to the children in the range from the first up to the i-th child. In what follows (Fig. 2), we consider inserting the key 9 into the full leaf ℓ (storing the keys 8, 10, and 12), and propose a strategy different from splitting ℓ by considering its $q = 3$ siblings.

leaf is represented as an array of length t; each entry of this array has k bits. We call such an array a *leaf array*. Each leaf additionally stores a pointer to its preceding and succeeding leaf. Each internal node v stores an array of length t for the pointers to its children, and an integer array I_v of length $t-1$ to distinguish the children for guiding a top-down navigation. In more detail, $I_v[i]$ is a key-comparable integer such that all keys of at most $I_v[i]$ are stored in the subtrees rooted (a) at the i-th child u of v or (b) at u's left siblings. Since the integers of I_v are stored in ascending order (with respect to the order imposed on the keys), to know in which subtree below v a key is stored, we can perform a binary search on I_v.

A root-to-leaf navigation can be conducted in $\mathcal{O}(\lg n)$ time, since there are $\mathcal{O}(\lg n)$ nodes on the path from the root to any leaf, and selecting a child of a node can be done with a linear scan of its stored keys in $\mathcal{O}(t) = \mathcal{O}(1)$ time.

Regarding space, each leaf stores at least $t/2$ keys. So there are at most $2n/t$ leaves. Since a leaf array uses kt bits, the leaves can use up to $2nk$ bits. This is at most twice the space needed for storing all keys in a plain array. In what follows, we provide a space-efficient variant.

3 Space-Efficient B Trees

To obtain a space-efficient B tree variant, we apply two ideas. We start with the idea to share keys among several leaves (Sect. 3.1) to maintain the space of the leaves more economically. Subsequently, we can adapt this technique for leaves maintaining a *non-constant* number of keys efficiently (Sect. 3.2), leading to the final space complexity of our proposed data structure (Sect. 3.3) and Thm. 1.

3.1 Resource Management by Distributing Keys

Our first idea is to keep the leaf arrays more densely filled. For that, we generalize the idea of B* trees [16, Sect. 6.2.4]: The B* tree is a variant of the B tree (more precisely, we focus on the B+ tree variant) with the aim to defer the split of a full leaf on insertion by rearranging the keys with a dedicated sibling leaf. On inserting a key into a full leaf, we try to move a key of this leaf to its dedicated sibling. If this sibling is also full, we split both leaves up into three leaves, each having $2/3 \cdot b$ keys on average [16, Sect. 6.2.4], where $b = t$ is the maximum number of keys a leaf can store. Consequently, we have the number of leaves is at most $3n/2b$. We can generalize this bound by allowing a leaf to share its keys with $q \in \Theta(\lg n)$ siblings. For that, we introduce the following invariant:

Among the q siblings of every non-full leaf, there is at most one other non-full leaf.

We can leave it open to precisely specify which q siblings are assigned to which leaf. For instance, the following is possible: we can assign $q/2$ leaves to the right and to the left side to each leaf. However, if the leaf in question has $o(q)$ left siblings like the leftmost leaf, we take more of its right siblings in considerations (and by symmetry if the leaf has $o(q)$ right siblings), such that each leaf gets q siblings assigned. For this to work, we need at least q leaves, which is granted by $n = \Omega(bq)$ as stated in Sect. 2. We note that it is possible to also accommodate smaller numbers with our techniques; we defer this analysis to Sect. 3.4.

Let us first see why this invariant helps us to improve the upper bound on the number of leaves; subsequently we show how to sustain the invariant while retaining our operational time complexity of $\mathcal{O}(\lg n)$: By definition, for every q subsequent leaves, there are at most two leaves that are non-full. Consequently, these q subsequent leaves store at least $qb - 2b$ keys. Hence, the number of leaves is at most $\lambda := nq/(qb - 2b)$, and all leaves of the tree use up to

$$\begin{aligned} \lambda bk &= nqbk/(qb - 2b) = nkq/(q - 2) = nk + 2nk/(q - 2) \\ &= nk + \mathcal{O}(nk/\lg n) \text{ bits for } q \in \Theta(\lg n). \end{aligned} \tag{1}$$

To obey the aforementioned invariant, we need to take action whenever we delete a key from a full leaf or try inserting a key into a full leaf:

Deletion When deleting a key from a full leaf ℓ having a non-full leaf ℓ' as one of its q siblings, we shift a key from ℓ' to ℓ such that ℓ is still full after the deletion. If ℓ' becomes empty, then we delete it.

Insertion Suppose that we want to insert a key into a leaf ℓ that is full. Given that one of the q sibling leaves of ℓ, say ℓ', is not full, then we shift a key from ℓ to ℓ' such that ℓ can store the new key. If there is no such ℓ', then we split ℓ. In that case, we create two new leaves, each inheriting half of the keys of the old leaf. In particular, these two leaves are the only non-full leaves among their q siblings.

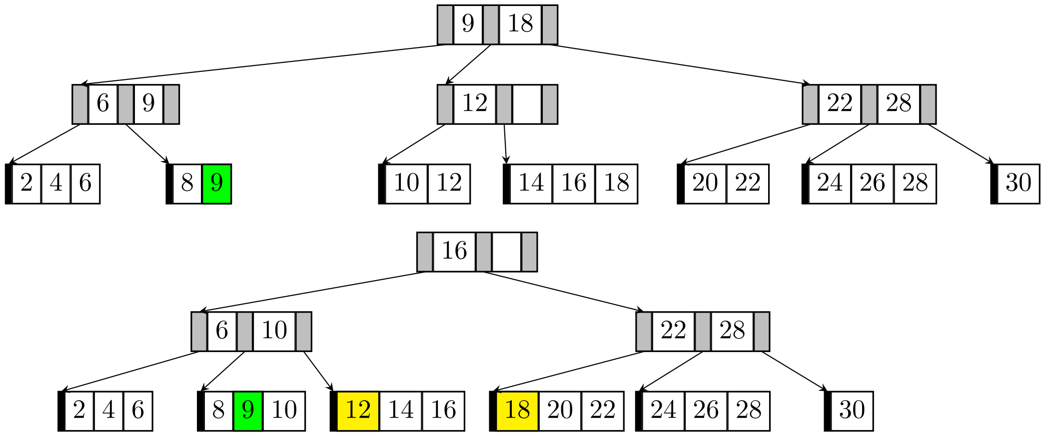

Fig. 2. Fig. 1 after inserting the key 9 into the leaf ℓ. Top: The standard B+ and B* variants split ℓ on inserting 9, causing its parent to split, too. Bottom: In our proposed variant (cf. Sect. 3) for $q \geq 3$, we shift the key 12 of ℓ to its succeeding leaf, from which we shift the key 18 to the next succeeding leaf, which was not yet full.

It is left to analyze the time for the shifting of a key: Since each leaf stores up to $b = t = \mathcal{O}(1)$ keys, shifting a key to one of the q siblings takes $\mathcal{O}(bq) = \mathcal{O}(\lg n)$ time. That is because, for shifting a key from the i-th leaf to the j-th leaf with $i < j$, we need to move the largest key stored in the g-th leaf to the $(g+1)$-th leaf for $g \in [i..j)$ (the moved key becomes the smallest key stored in the $(g+1)$-th leaf, cf. Fig. 2). Since a shift changes the entries of $\mathcal{O}(q)$ leaves, we have to update the information of those leaves' ancestors. By updating an ancestor node v we mean to update its integer array I_v as described in Sect. 2, which can be done in $\mathcal{O}(t)$ time. There are at most $\sum_{h=1}^{\lg n} \lceil q(t/2)^{-h} \rceil = \mathcal{O}(\lg n + q)$ many such ancestors, and all of them can be updated in time linear to the tree height, which is $\mathcal{O}(\lg n)$ for B trees with constant degree $t = \mathcal{O}(1)$. Thus, we obtain a B* tree variant with the same time complexities, but higher occupation rates of the leaves.

3.2 Shifting Keys Among Large Leaves

Next, we want to reduce the number of internal nodes. For that, we increase the number of elements a leaf can store up to $b := (w \lg n)/k$. Since a leaf now maintains a large number of keys, shifting a key to one of its q neighboring sibling leaves takes $\mathcal{O}(bqk/w) = \mathcal{O}(\lg^2 n)$ time. That is because, for an insertion into a leaf array, we need to shift the stored keys to the right to make space for the key we want to insert. We do not shift the keys individually (that would take $\mathcal{O}(b)$ total time). Instead, we can shift $\Theta(w/k)$ keys in constant time by using word-packing, yielding $\mathcal{O}(bk/w)$ time for an insertion or deletion of a key in a leaf array. In what follows, we combine the word-packing technique with *circular buffers* representing the leaf arrays to improve the time bounds to $\mathcal{O}(\lg n)$.

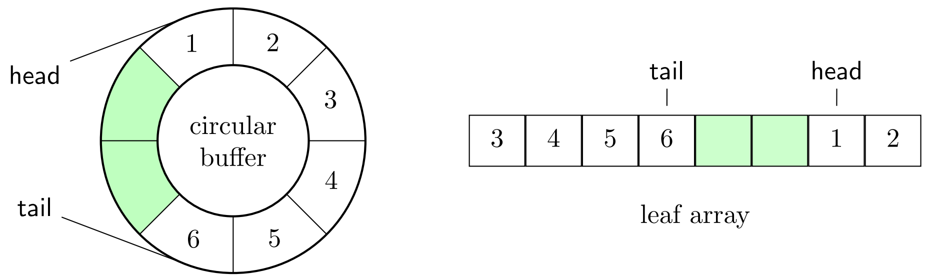

Fig. 3. A circular buffer representation of a leaf array capable of storing 8 keys. The pointers head and tail support prepending a key, removing the first key, appending a key, and removing the last key, all in constant time. The right figure shows that the circular buffer is actually implemented as a plain array with two pointers.

A *circular buffer* supports, additionally to removing or adding the last element in constant time like a standard (non-resizable) array, the same operations for the first element in constant time as well. See Fig. 3 for a visualization. For an insertion or deletion elsewhere, we still have to shift the keys to the right or to the left. This can be done in $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time with word-packing as described in the previous paragraph for the plain leaf array (only extra care has to be taken when we are at the borders of the array representing the circular buffer). Finally, on inserting a key into a full leaf ℓ, we pay $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time for the insertion into this full leaf, but subsequently can shift keys among its sibling leaves in constant time per leaf. Similarly, on deleting a key of a full leaf ℓ, we first rearrange the circular buffer of ℓ in $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time, and subsequently shift a key among the $\mathcal{O}(q)$ circular buffers of ℓ's siblings to keep ℓ full, which takes also $\mathcal{O}(q) = \mathcal{O}(\lg n)$ time.

3.3 Final Space Complexity

Finally, we can bound the number of internal nodes by the number of leaves λ defined in Sect. 3.1: Since the minimum out-degree of an internal node is $t/2$, there are at most

$$\lambda \sum_{i=1}^{\infty} (2/t)^i = 2\lambda/(t-2) = \mathcal{O}(n(q+1)/(qtb)) = \mathcal{O}(n/tb) \text{ internal nodes.}$$

Since an internal node stores t pointers to its children, it uses $\mathcal{O}(tw)$ bits. In total we can store the internal nodes in

$$\mathcal{O}(twn/tb) = \mathcal{O}(wn/b) = \mathcal{O}(nk/\lg n) \text{ bits.} \tag{2}$$

Each circular buffer (introduced in Sect. 3.2) requires $\Theta(\lg b)$ bits (for the pointers in Fig. 3). The additional total space is $\lambda\,\mathcal{O}(\lg b) = \mathcal{O}(nq \lg b/(bq-2b)) = \mathcal{O}(n \lg b/b) = o(n)$ bits. Together with Eq. (1), we finally obtain Thm. 1.

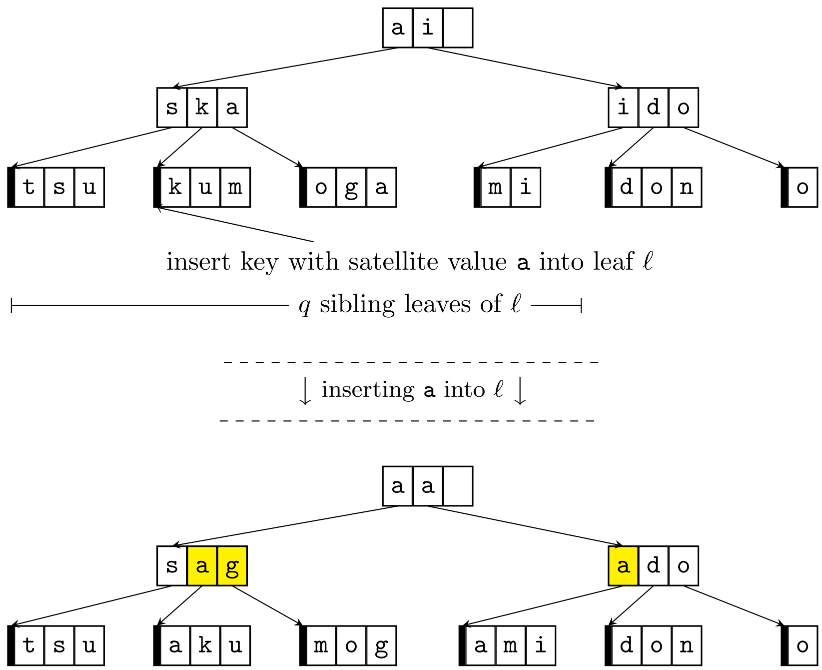

Fig. 4. Change of aggregate values on shifting keys. A shift causes the need to recompute the aggregate values of the satellite values stored in a leaf whose contents changed due to the shift. The example uses the same B tree structure as Fig. 1, but depicts the satellite values (plain characters) instead of the keys. Here, we used the minimum on the canonical Latin alphabet order as aggregate function.

3.4 Low Number of Keys

For our B tree, we require that $n = \Omega((w \lg^2 n)/k)$. When $n = \mathcal{O}((w \lg^2 n)/k)$ but $k = o(n/\lg^2 n)$, we can still provide a succinct solution within the same operational time complexity, which consists of a single internal node (i.e., the root node) governing the leaves. The leaves are defined as before, except that we set the maximum number of keys a leaf can store to $b = (w \lg n)/k^2$. Consequently, the root maintains $\mathcal{O}(k \lg n)$ leaves, and for each leaf ℓ the root stores a key to delegate a search to ℓ. By maintaining this key-leaf delegation in a binary search tree, we can search and update these keys in the root in $\mathcal{O}(\lg(k \lg n)) = \mathcal{O}(\lg n)$ time. We only keep at most one non-full leaf costing us $(w \lg n)/k$ bits. The distribution of the keys among the leaves is performed as before, except that we consider, when shifting keys, all leaves instead of just the q siblings. In total, we have an overhead of $(w \lg n)/k + \mathcal{O}(k \lg^2 n) = o(n)$ bits.

4 Augmenting with Aggregate Values

As highlighted in the related work section (Sect. 1.1), B trees are often augmented with auxiliary data to support prefix sum queries or LCP queries when storing strings. We present a more abstract solution covering these cases with *aggregate values*, i.e., values composed of the satellite values stored along with the keys in the leaves. In detail, we augment each node v with an *aggregate value* that is the return value of a decomposable aggregate function applied on the satellite values stored in the leaves of the subtree rooted at v. A *decomposable aggregate function* [14, Sect. 2A] such as the sum, the maximum, or the minimum, is a function f on a subset of satellite values with a constant-time merge operation $\cdot_f$ such that, given two disjoint subsets X and Y of satellite values, $f(X \cup Y) = f(X) \cdot_f f(Y)$, and the left-hand and the right-hand side of the equation can be computed in the same time complexity. We further assume that each aggregate value produced by f is storable in $\mathcal{O}(w)$ bits to fit into the $\mathcal{O}(tw)$-bits space bound of an internal node.

While sustaining the methods described in the introduction like predecessor for *keys*, we enhance insert to additionally take a value as argument, and provide access to the aggregate values:

insert(K, V) inserts the key K with satellite value V;
access(v) returns the aggregate value of the node v; and
access(K) returns the satellite value of the key K.

To make use of access(v), the B tree also provides access to the root, and a top-down navigation based on the way predecessor(K) works, for a key K as search parameter. To keep things simple, we assume that all keys are distinct[3] (i.e., we allow no duplicates).

For the computational analysis, let us assume that every satellite value uses $\mathcal{O}(k)$ bits, and that we can evaluate the given aggregate function f bit-parallel such that it can be evaluated in $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time for a leaf storing $b = \Theta(w \lg n/k)$ values.

Under this setting, we claim that we can obtain $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time for every B tree operation while maintaining the aggregate values, even if we distribute keys among q leaves on (a) an insertion of a key into a full leaf or (b) the deletion of a key. This is nontrivial: For instance, when maintaining minima as aggregate values, if we shift the key with minimal value of a leaf ℓ to its sibling, we have to recompute the aggregate value of ℓ (cf. Fig. 4), which we need to do from scratch (since we do not store additional information about finding the next minimum value). So a shift of a key to a leaf costs $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time, resulting in $\mathcal{O}(qbk/w) = \mathcal{O}(\lg^2 n)$ overall time for an insertion.

Our idea is to decouple the satellite values from the leaf arrays where they are actually stored. To explain this idea, let us *conceptually* think of the leaf arrays as a global array — meaning that these arrays are still represented by their respective circular buffers *individually*. Given our B tree has λ leaves, we

[3] Note that if all keys are distinct, then $k \geq \lg n$ by the pigeonhole principle.

partition this global array into λ blocks, where the i-th block with $i \in [1..\lambda]$ starts initially at entry position $1+(i-1)b$, corresponds to the i-th leaf, and has initially the size equal to the capacity of the circular buffer of its corresponding leaf. The crucial change is that we let the aggregate value of a leaf depend on its corresponding block instead of its leaf array. While leaf arrays (represented by circular buffers) have a fixed capacity, we can move block boundaries freely to extend or shrink the size of a block.

Now suppose that we want to insert an element e into a full leaf ℓ, and that one of its q siblings is not full. Hence, we can redistribute one element of ℓ's leaf array by shifting one element across $\mathcal{O}(q)$ leaf arrays as explained in Sect. 3.2. After the redistribution, without the blocks, we would have to update the aggregate values of $\mathcal{O}(q)$ siblings. Instead of that, we just enlarge the block of ℓ to cover e, and update the aggregate value of ℓ with e. This allows us to process an update operation by $\mathcal{O}(q)$ block boundary updates, and a constant number of updates of the aggregate values stored in the leaves. In summary, we can decouple the aggregate values from the leaf arrays with the aid of the blocks in the global array, and therefore can use the techniques introduced in Sect. 3.2, where we shift keys among $q+1$ sibling leaves, without the need to recompute the aggregate values of the sibling leaves when shifting keys.

Example 1. Let us assume for simplicity that $b = 3$ and that the keys are the values. Suppose that our B tree consists of exactly three leaves ℓ_i for $i = 1, 2, 3$. Each leaf ℓ_i has a leaf array A_i with the following contents: $A_1 = (1, 2, 4)$, $A_2 = (5, 6, 7)$, and $A_3 = (8, 9, \perp)$, where $\perp$ denotes an empty slot. Further assume that our aggregate function f is min such that $f(A_1) = 1, f(A_2) = 5, f(A_3) = 8$. Now suppose that we want to insert 3 into A_1. Without the block reassignment, we would shift 4 to A_2 and 7 to A_3 such that we need to update the aggregate values of ℓ_2 and ℓ_3 to $f(A_2) = 4, f(A_3) = 7$. Now, with the block reassignment, we do the following: We think of the A_i's as a single array $A[1..9] = (1, 2, 4, 5, 6, 7, 8, 9, \perp)$ and partition it initially into blocks of equal length $B_1 = A[1..3], B_2 = A[4..6], B_3 = A[7..9]$. The blocks are basically just pointers into A such that updates of A automatically update the contents of the B_i's. Now the aggregate values of the leaves are no longer based on the A_i's, but on the B_i's, i.e., $f(B_1) = 1, f(B_2) = 5, f(B_3) = 8$. If we perform the same insertion as above inserting 3 into A_1, we perform the shifting as before such that $A[1..9] = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, but additionally increase the size of B_1 and shift B_2 and B_3 to the right such that $B_1 = A[1..4], B_2 = A[5..7], B_3 = A[8..9]$. Consequently, the aggregate values of ℓ_1's siblings do not have to be updated. The key observation is that while the leaf array A_1 of ℓ_1 governs $b = 3$ elements, the block B_1 of ℓ_1 is allowed to contain more/fewer than b elements.

To track the boundaries of the blocks, we augment each leaf ℓ with an offset value and the current size of its block. The offset value stores the relative offset of the block with respect to the initial starting position of the block (equal to the starting position of ℓ's leaf array) within the global array. We decrement the offset by one if we shift a key from ℓ to ℓ's preceding sibling, while we increment its offset by one if we shift a key of ℓ's preceding leaf to ℓ.

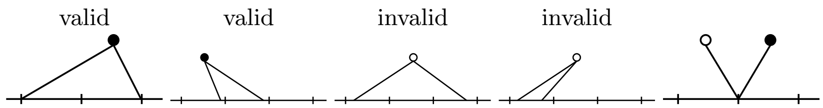

Fig. 5. Valid and invalid blocks according to the definition given in Sect. 4.2. The (conceptual) global array is symbolized by a horizontal line. The leaf arrays are intervals of the global array separated by vertical dashes. A dot symbolizes a leaf ℓ and the intersection of the triangle spawning from ℓ with the global array symbolizes the block of ℓ. A node has an invalid block if its dot is hollow. The rightmost picture shows the border case that a block is invalid if its offset is b, while a block can be valid even if it is empty.

If we only care about insertions (and not about deletions and blocks becoming too large) we are done since we can update $f(X)$ to $f(X \cup \{x\})$ in constant time for a new satellite value $x \notin X$ per definition. However, deletions pose a problem for the running time because we usually cannot compute $f(X \setminus \{x\})$ from $f(X)$ with $x \in X$ in constant time. Therefore, we have to recompute the aggregate value of a block by considering all its stored satellite values. However, unlike leaf arrays whose sizes are upper bounded by b, blocks can grow beyond $\omega(b)$. Supporting deletions, we cannot ensure with our solution up so far to recompute the aggregate value of a block in $\mathcal{O}(\lg n)$ time. In what follows, we show that we can retain logarithmic update time, first with a simple solution taking $\mathcal{O}(\lg n)$ time amortized, and subsequently with a solution taking $\mathcal{O}(\lg n)$ time in the worst case.

4.1 Updates in Batch

Our amortized solution takes action after a node split occurs, where it adjusts the blocks of all $q+2$ nodes that took part in that split (i.e., the full node, its q full siblings and the newly created node). The task is to evenly distribute the block sizes, reset the offsets, and recompute the aggregate values. We can do all that in $\mathcal{O}(q(bk/w + \lg n)) = \mathcal{O}(\lg^2 n)$ time, since

- there are $\mathcal{O}(q)$ leaves involved,
- each leaf stores at most b values, whose aggregate value can be computed in $\mathcal{O}(bk/w) = \mathcal{O}(\lg n)$ time, and
- each leaf has $\mathcal{O}(\lg n)$ ancestors whose aggregate values may need to be recomputed.

Although the obtained $\mathcal{O}(\lg^2 n)$ time complexity seems costly, we have increased the total capacity of the $b+2$ nodes involved in the update by $\Theta(b)$ keys in total. Consequently, before splitting one of those nodes again, we perform at least $b = \Omega(\lg n)$ insertions (remember that we split a node only if it and its q siblings are full). Now, whenever a block becomes larger than $2b$, we can afford the above rearrangement costing $\mathcal{O}(\lg n)$ amortized time.

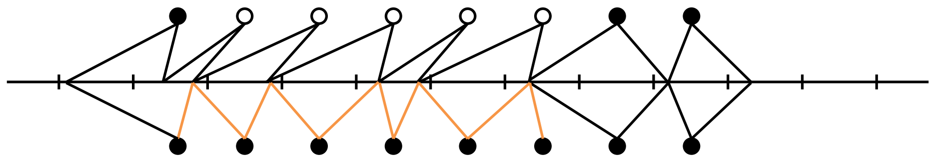

Fig. 6. Revalidation of multiple invalid blocks. The figure uses the same pictography as Fig. 5, but additionally shows on the bottom (vertically mirrored) the outcome of our algorithm fixing the invalid blocks (Sect. 4.2), where we empty the rightmost invalid block and swap the blocks until we find a block that can be merged with the previous block.

4.2 Updates by Merging

To improve the time bound to $\mathcal{O}(\lg n)$ worst case time, our trick is to merge blocks and reassign the ownership of blocks to sibling leaves. For the former, a merge of two blocks means that we have to combine two aggregate values, but this can be done in constant time by the definition of the decomposable aggregate function. To keep the size of the blocks within $\mathcal{O}(b)$, we watch out for blocks whose shape underwent too much changes, which we call invalid (see Fig. 5 for a visualization). We say a block is *valid* if it covers at most $2b$ keys, it has an offset in $(-b..b)$ (i.e., the block starts within the leaf array of the preceding leaf or of its corresponding leaf), and the sum of offset and size is in $[0..2b)$ (i.e., the block ends within the leaf array of its corresponding leaf or its succeeding leaf). Initially, all blocks are valid because they have size b and offset 0. If one of those conditions for a block becomes violated, we say that the block is *invalid*, and we take action to restore its validity. Blocks can become invalid when changing their sizes by one, or when shifting their boundaries by one. A shift can cause $\mathcal{O}(q)$ blocks to become invalid (i.e., the number of siblings considered when distributing keys). Suppose that a block B_i has become invalid due to a tree update, which already costed $\mathcal{O}(\lg n)$ time (the time for a root-to-leaf traversal). Our goal is to rectify the invalid block B_i within the same time bound. B_i has become invalid because of the events that (a) it covers $2b+1$ keys, (b) has offset $-b$ or the sum of offset and size are negative, or (c) has offset $+b$ or the sum of offset and size are at least $2b$.

For event (a), we redistribute sizes and offsets of B_i with B_{i-1} and B_{i+1}. This is possible since at least one block B_{i-1} or B_{i+1} has less than $2b$ keys. Otherwise, since they are valid blocks[4], there is no space left for B_i to have $2b+1$ keys. It is therefore possible for B_i to consign at least one key to its neighbors without making them overfull. We finish by recomputing the aggregate values of the three nodes and their ancestors, costing $\mathcal{O}(\lg n)$ total time.

The events (b) and (c) can happen when shifting blocks by one to the left (b) or to the right (c). Given B_i is the rightmost (for (b)) or the leftmost (for (c)) invalid block, we swap boundaries with the preceding blocks (for (b)) or

[4] More precisely, these blocks were valid at least before the enlargement of B_i, which could have triggered a shifting that invalidated either B_{i-1} or B_{i+1}.

succeeding blocks (for (c)) of B_i until finding a block whose boundaries can be extended to cover the shifted part without becoming invalid. The number of blocks we take into consideration is $\mathcal{O}(q)$, since we stop at a block B_j with $|B_j| + |B_{j+1}| \leq 2b$; if there are more blocks that do not satisfy this condition, then more than q consecutive siblings leaves are full, and a leaf split must have had occurred.

To solve (b), we proceed as follows — (c) can be solved symmetrically. First, we put B_i on a stash S (storing B_i's boundaries and its aggregate value), and empty B_i. Next, we check whether B_{i-1} can be extended to cover S without becoming invalid. If this is possible, we let B_{i-1} cover S, update the aggregate value of B_{i-1}, and terminate. Otherwise (B_{i-1} would become invalid), we swap B_{i-1} with S. Now B_{i-1} stores the boundaries of S. By doing so, B_{i-1} does not become invalid since the offset of B_i was $-b$ (and thus the offset of B_{i-1} becomes 0), or the sum of offset and size was in $[-b..0)$ (which becomes $[0..b)$), while the changed offset poses no problem, since the sum of offset and size of B_{i-1} is now at most $2b - 1$. Finally, we iteratively select the next preceding block B_{i-2} to check whether it is mergeable with the stash S without becoming invalid (cf. Fig. 6). Since each visit of a block takes constant time (either swapping or merging contents), and we visit $\mathcal{O}(q)$ blocks, fixing all invalid blocks takes $\mathcal{O}(q) = \mathcal{O}(\lg n)$ time.

5 Conclusion

We provided a space-efficient variation of the B tree that retains the time complexity of the standard B tree. It achieves succinct space when the keys are considered to be incompressible. Our main tools were the following: First, we generalized the B* tree technique to exchange keys not only with a dedicated sibling leaf but with up to q many sibling leaves. Second, we let each leaf store $\Theta(b)$ elements represented by a circular buffer such that moving a largest (resp. smallest) element of a leaf to its succeeding (resp. preceding) sibling can be performed in constant time. Additionally, we could augment each node with an aggregate value and maintain these values, either with a batch update weakening the worst case time complexities to amortized time, or with a blocking of the leaf arrays that can be maintained within the worst case time complexities.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers JP19K20213 (TI), JP21K17701 and JP21H05847 (DK).

References

[1] Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. In: Proc. SIGFIDET. pp. 107–141 (1970)
[2] Bille, P., Christiansen, A.R., Cording, P.H., Gørtz, I.L., Skjoldjensen, F.R., Vildhøj, H.W., Vind, S.: Dynamic relative compression, dynamic partial sums, and substring concatenation. Algorithmica 80(11), 3207–3224 (2018)

[3] Blandford, D.K., Blelloch, G.E.: Compact representations of ordered sets. In: Proc. SODA. pp. 11–19 (2004)
[4] Comer, D.: The ubiquitous B-tree. ACM Comput. Surv. 11(2), 121–137 (1979)
[5] Delpratt, O., Rahman, N., Raman, R.: Compressed prefix sums. In: Proc. SOFSEM. LNCS, vol. 4362, pp. 235–247 (2007)
[6] Dietz, P.F.: Optimal algorithms for list indexing and subset rank. In: Proc. WADS. LNCS, vol. 382, pp. 39–46 (1989)
[7] Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM 21(2), 246–260 (1974)
[8] Farzan, A., Munro, J.I.: Succinct representation of dynamic trees. Theor. Comput. Sci. 412(24), 2668–2678 (2011)
[9] Ferragina, P., Grossi, R.: The string B-tree: A new data structure for string search in external memory and its applications. J. ACM 46(2), 236–280 (1999)
[10] Franceschini, G., Grossi, R.: Optimal implicit dictionaries over unbounded universes. Theory Comput. Syst. 39(2), 321–345 (2006)
[11] González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theor. Comput. Sci. 410(43), 4414–4422 (2009)
[12] Graefe, G.: Modern B-tree techniques. Foundations and Trends in Databases 3(4), 203–402 (2011)
[13] He, M., Munro, J.I.: Succinct representations of dynamic strings. In: Proc. SPIRE. LNCS, vol. 6393, pp. 334–346 (2010)
[14] Jesus, P., Baquero, C., Almeida, P.S.: A survey of distributed data aggregation algorithms. IEEE Commun. Surv. Tutorials 17(1), 381–404 (2015)
[15] Katajainen, J., Rao, S.S.: A compact data structure for representing a dynamic multiset. Inf. Process. Lett. 110(23), 1061–1066 (2010)
[16] Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley, Redwood City, CA, USA (1998)
[17] Munro, J.I., Nekrich, Y.: Compressed data structures for dynamic sequences. In: Proc. ESA. LNCS, vol. 9294, pp. 891–902 (2015)
[18] Navarro, G., Nekrich, Y.: Optimal dynamic sequence representations. SIAM J. Comput. 43(5), 1781–1806 (2014)
[19] Prezza, N.: A framework of dynamic data structures for string processing. In: Proc. SEA. LIPIcs, vol. 75, pp. 11:1–11:15 (2017)
[20] Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Proc. ICALP. LNCS, vol. 2719, pp. 357–368 (2003)