Adaptive Dynamic Bitvectors*

Gonzalo Navarro[1,2,3]

[1] Dept. of Computer Science, University of Chile, Santiago, Chile
[2] CeBiB — Center for Biotechnology and Bioengineering, Chile
[3] IMFD — Millennium Institute for Foundational Research on Data, Chile

Abstract. While operations *rank* and *select* on static bitvectors can be supported in constant time, lower bounds show that supporting updates raises the cost per operation to $\Theta(\log n/\log\log n)$. This is a shame in scenarios where updates are possible but uncommon. We develop a representation of bitvectors that, if there are q queries per update, supports all the operations in $O(\log(n/q))$ amortized time. Our experimental results support the theoretical findings, displaying speedups of orders of magnitude compared to standard dynamic implementations.

1 Introduction

Bitvectors are the basic bricks of most compact data structures [11]. Apart from the basic query $\mathsf{access}(B,i)$, which retrieves the bit $B[i]$ of the bitvector $B[1\,..\,n]$, they support two fundamental queries: $\mathsf{rank}_b(B,i)$, which tells the number of times the bit $b\in\{0,1\}$ occurs in $B[1\,..\,i]$, and $\mathsf{select}_b(B,j)$, which gives the position of the jth occurrence of $b\in\{0,1\}$ in B. It is well known since the nineties that those operations can be supported in $O(1)$ time with a bitvector representation that uses $n+o(n)$ bits of space [4, 10]. Things are considerably different, however, if we aim to allow updates to the bitvector: just supporting rank and bit flips requires $\Omega(\log n/\log\log n)$ time [7]. Indeed, one can incorporate in $O(\log n/\log\log n)$ time, and still $n+o(n)$ bits of space, the operations $\mathsf{write}(B,i,v)$, which sets $B[i]=v$, $\mathsf{insert}(B,i,v)$, which inserts the bit value v at position i in B, and $\mathsf{delete}(B,i)$, which removes the bit $B[i]$ from B [12].

This almost logarithmic gap between static and dynamic bitvectors permeates through most compact data structures that build on them, making dynamic compact data structures considerably slower than their static counterparts, and not as competitive with classic data structures. Although this price is in principle unavoidable, one may wonder whether it must be so high in cases where updates are sparse compared to queries, as is the case in many applications. As an extreme example, since the static data structures can be built in linear time, one could have $O(1)$ amortized time if queries were $\Omega(n)$ times more frequent than updates, by just rebuilding the static structure upon each update. The idea degrades quickly, however: If queries are q times more frequent than updates, this technique yields $O(n/q)$ amortized times.

* Funded by ANID, Chile, via Basal Funds FB0001, Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755.

In this paper we introduce a representation of dynamic bitvectors $B[1\,..\,n]$ that uses at most $4n+o(n)$ bits and offers $O(\log(n/q))$ amortized time for all the operations, if queries are q times more frequent than updates. We modify classic dynamic bitvector representations [3, 9]. Our structure is a binary tree whose leaves may either be "dynamic", storing $O(\log^2 n)$ bits and supporting updates, or long "static" bitvectors handling only queries. A whole subtree is converted into static—which we call "flattening"—when it has received sufficient queries to amortize the cost of building the static structures (i.e., linear in the number of bits it represents). When an update falls in a static leaf, the leaf is recursively halved into static leaves of decreasing lengths until producing a (short) dynamic leaf where the update is executed—a process we call "splitting". For maintaining balance in the tree we resort to weight-balancing [13, 1], which interacts well with our new operations of flattening and splitting.

2 Our Work in Context

Our problem is an instance of the so-called "dynamic bitvector with indels" problem, which as said requires $\Omega(\log n/\log\log n)$ time per operation even if we support only rank and write [7]. Several solutions have matched this lower bound, or been close to. Hon et al. [8] store a dynamic bitvector $B[1\,..\,n]$ in $n+o(n)$ bits of space, handling queries in time $O(\log_b n)$ and updates in time $O(b)$, for any $b = \Omega((\log n/\log\log n)^2)$. Their main structure is a weight-balanced B-tree [5, 15]. Chan et al. [3] use balanced binary trees with leaves containing $\Theta(\log n)$ bits, obtaining $O(n)$ bits of space and $O(\log n)$ time for all the operations. Mäkinen and Navarro [9] still use balanced binary trees, but use leaves of $\Theta(\log^2 n)$ bits, retaining their $O(\log n)$ times but reducing the space to $n + o(n)$ bits. Finally, Navarro and Sadakane [12] replace binary trees by structures closer to B-trees, retaining the $n + o(n)$ bits of space and supporting all the operations in the optimal time $O(\log n/\log\log n)$. In those terms the problem is considered closed.

In this paper we aim at obtaining $O(\log(n/q))$ times under a regime where there are, on average, q queries per update. Our results are amortized, as we rely on converting whole subtrees into static structures (which answer queries in constant time) when they have received sufficient queries to pay for that conversion. The conversion needs to temporarily double the space for the bits stored in the converted tree, thus we cannot aim at using $n+o(n)$ bits of space. For simplicity, we will aim at using $4n + o(n)$ bits—though we could reduce it to $(3 + \epsilon)n$ for any $\epsilon > 0$. To reach such a concrete constant, we use the bigger leaves of $\Theta(\log^2 n)$ bits, but we allow them to be partly filled, which considerably simplifies matters in comparison with the solutions using $n + o(n)$ bits [9, 12].

There has been work to store the bitvectors within entropy space, which means Hn bits with $H = \frac{m}{n}\log_2\frac{n}{m} + \frac{n-m}{n}\log_2\frac{n}{n-m}$, m being the number of 1s in the bitvector. Assuming $m < n$, Blandford and Blelloch [2] obtain $O(nH+\log n)$ bits of space while supporting all operations in $O(\log n)$ time, using a balanced binary tree where the distances between consecutive 1s are gap-encoded in the leaves. Mäkinen and Navarro [9] improve the space to $nH + o(n)$ bits, while

retaining $O(\log n)$ time for the operations. Navarro and Sadakane [12] retain this space and reduce the time to the optimal $O(\log n/\log\log n)$. We discuss in the Conclusions how our results can be extended to use entropy-bounded space.

3 Adaptive Dynamic Bitvectors

We use the transdichotomous RAM model of computation, with computer words of $w = \Theta(\log N)$ bits, N being the maximum size of a bitvector that fits in memory. Pointers use w bits. We call $n \leq N$ is the current size of bitvector B.

3.1 Structure

As anticipated, our data structure is a binary tree. Its leaves are of two types:

- A "dynamic leaf", which allocates space for $b = \Theta(w \log n)$ bits and no precomputed answers. A dynamic leaf answers access queries in $O(1)$ time and rank/select queries in $O(\log n)$ time, via word-wise scanning [9, 11].[4]
- A "static leaf", which stores arbitrarily large bitvectors with their corresponding precomputation to solve access/rank/select in $O(1)$ time [4, 10].

The internal tree nodes v record, apart from their two children v.left and v.right, the following fields:

v.size : total number of bits represented in the subtree rooted at v.
v.ones : total number of 1-bits represented in the subtree rooted at v.
v.leaves : number of leaves below v (static leaves count as many, see later).
v.queries : number of queries that traversed v since the last update that traversed v, or since the creation of v.

Our binary tree is maintained with balanced weight [13, 1]: given a parameter $1/2 < \alpha < 1$, for every node v, v.left.size $\leq \alpha \cdot v$.size and v.right.size $\leq \alpha \cdot v$.size. This implies that the tree has height at most $\log_{\frac{1}{1-\alpha}} n = O(\log n)$. Balancing will be maintained by reconstructing biased trees as perfectly balanced [1]. Similarly, we will ensure that v.leaves $= O(v.\text{size}/(w \log n))$, by converting nodes into static when the leaves below them are too empty. Since each tree node uses $O(w)$ bits of space, this ensures that all the tree nodes use together $O(n/\log n)$ bits.

3.2 Queries

The queries use in principle the standard mechanism for dynamic bitvectors [9]: access(B, i) traverses the tree from the root, going to the left child if $i \leq v$.size and to the right otherwise (subtracting v.size from i in that case). When arriving at a leaf, the query completes in $O(1)$ additional time. Since our trees are balanced, access takes time $O(\log n)$ in the worst case.

[4] Mäkinen and Navarro [9] show how to maintain b when $\lceil \log n \rceil$ changes without affecting any complexity. Our implementation uses a fixed value of b for simplicity.

The procedure for $\mathsf{rank}_1(B,i)$ is analogous, adding up v.left.ones whenever we descend to v.right, and adding at the end $\mathsf{rank}_1(i)$ on the leaf. For $\mathsf{rank}_0(B,i)$ we compute $i - \mathsf{rank}_1(B,i)$. For $\mathsf{select}_1(B,j)$, we descend guided by v.ones instead of by v.size, subtract v.left.ones from j when going to v.right, and accumulate v.left.size instead of v.left.ones; at the end we add $\mathsf{select}_1(j)$ on the leaf. Finally, $\mathsf{select}_0(B,j)$ is analogous to $\mathsf{select}_1(B,j)$, using v.left.size $-$ v.left.ones instead of v.left.ones. Both rank and select take time $O(\log n)$ plus the time spent on the leaf, which is $O(\log n)$ on dynamic leaves—by counting the 1s in w-bit words in constant time [11, Sec. 4.2]—and $O(1)$ on static ones.

Flattening. The novelty in our adaptive scheme is that, every time we traverse an internal node v for any of the three queries, we increment v.queries, and if v.queries $\geq \theta \cdot v$.size, for an appropriate tuning constant θ, we convert the whole subtree of v into a static leaf, which we call "flattening" v. Flattening is done in time $O(v.\text{size})$, by traversing and deleting the subtree of v, while writing the bits of all the leaves onto a new bitvector, which is finally preprocessed for constant-time queries and converted into the static leaf corresponding to v. We show later, however, that its amortized cost is absorbed by the preceding $\theta \cdot v$.size queries.

Note that flattening temporarily increases the space used by v.size, which may be as much as n if v is the root. Note also that flattening does not change v.size, and thus it does not affect the balancing.

3.3 Updates

Updates are handled, in principle, as in previous work [9]. To perform $\mathsf{write}(B,i)$, we traverse the tree as for access, modify the corresponding bit in the (dynamic) leaf we arrive at (we consider soon the case where we arrive at a static leaf), and update v.ones as we return from the recursion. This takes time $O(\log n)$ because our trees are balanced. Note that write has no effect on the tree balancing.

Insertions and deletions are analogous, yet at the end they insert or delete a bit in a dynamic leaf and must update v.size and v.ones along the path. Apart from costing $O(\log n)$ for shifting the needed bits wordwise, one must handle overflows and underflows in leaves. An overflow occurs when we insert a bit in a leaf v containing b bits already, and it is handled by splitting the leaf into two holding $b/2$ bits and making v the parent of both new leaves. An underflow occurs when we delete the only bit in a leaf v, in which case v is eliminated together with its parent node. We also act when the leaf v that receives the deletion is the sibling of another (dynamic) leaf, so that after the deletion both leaves add up to at most γb bits, where $1/2 \leq \gamma < 1$ is the desired fill ratio of newly created leaves. In this case both leaves are concatenated into one and their parent is eliminated. As we return from the recursion, we keep trying to merge the two children of the current node if both are dynamic leaves. As a consequence, every internal node v has v.size $\geq \gamma b$ (and can be flattened).

Splitting. The interesting part of updates occurs when we arrive at a static leaf v. In this case we halve the bitvector stored at v, and make v an internal node with

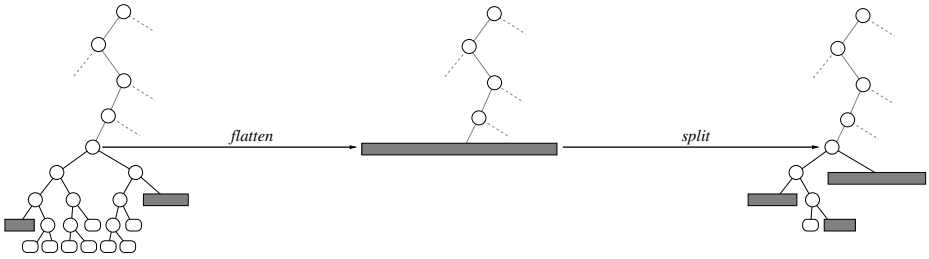

Fig. 1. Flattening and posterior splitting of a node, the former after receiving over $\theta \cdot v.\text{size}$ consecutive queries, and the latter when receiving an update at a static leaf; the leaf is recursively halved until the update falls in a dynamic leaf. Circles are internal nodes, round rectangles are dynamic leaves, and gray rectangles are static leaves.

one child holding each half (halving is slightly corrected to make the left half contain a multiple of γb bits, so that dynamic leaves can be created of size γb). The bitvector half that does not contain i is converted into a leaf again (static if it is of size over γb and dynamic if not). The half containing i is recursively halved, until it contains at most γb bits and so it becomes a dynamic leaf. Note that splitting v does not change $v.\text{size}$, and thus has no effect on the tree balance.

Splitting takes total time $O(v.\text{size})$, as we create leaves in time proportional to their bit length. Though the worst-case cost of updates can be $\Theta(n)$, we prove logarithmic amortized bounds later. Figure 1 illustrates flattening and splitting.

Balancing. We maintain the weight balance of the tree by checking, at every internal node v in the path from the root to the leaf where the insertion or deletion will take place, that the weight-balancing conditions will be satisfied once we insert or delete the bit on $v.\text{left}$ or $v.\text{right}$, depending on the case. For example, if the insertion must continue by the left, and it holds that $v.\text{left.size} + 1 > \alpha \cdot (v.\text{size} + 1)$, then node v will become unbalanced after the insertion.

When we detect that v will become unbalanced in our top-down traversal to insert or delete at position i, we (i) flatten v, and (ii) split v at position i. This will correct the imbalance of v, ensuring that all the nodes in the path towards position i are perfectly balanced until the final dynamic leaf. The cost is $O(v.\text{size})$, of the same order of the tree balancing performed on weight balanced trees, in the variant that rebuilds whole subtrees within low amortized time [1].

Because our splitting by half can be slightly shifted to ensure that lengths are multiples of γb, we avoid balancing when this shifting would leave the tree unbalanced anyway. This can be the case when the subtree is very small (e.g., if $v.\text{size} = 3\gamma b$, splitting will produce a child of size $2\gamma b$ and another of size γb, which is considered unbalanced if $\gamma > 1/3$), or if γ is too close to $1/2$. This correction does not affect the asymptotically logarithmic height of the tree.

As anticipated, despite our attempts to maintain leaves as full as possible, a constant fill ratio cannot be guaranteed. We then monitor, just like we do for bias in the left versus right subtree sizes, the ratio between $v.\text{leaves}$ and $v.\text{size}$. This

cannot be predicted in the top-down traversal as the bias, so we check instead, when returning from the recursion, that $v.\text{size} \geq (b/3) \cdot v.\text{leaves}$, and otherwise flatten v (there is no point in eagerly splitting v after flattening, as the deletion already took place). Note that leaves are created with fill ratio $1/2$ when a leaf overflows, and with fill ratio $\gamma \geq 1/2$ when a static leaf is split; we rebuild when the fill ratio falls below $1/3$, so this flattening cost will be amortized by the deletions. We assume $v.\text{leaves} = v.\text{size}/(\gamma b)$ for static leaves v, as γb is the fill ratio of the leaves it will create if split (thus splitting v does not alter $v.\text{leaves}$).

4 Amortized Analysis

We first show that all the operations have an amortized cost of $O(\log n)$; later we analyze the case where the frequency of updates is $1/q$. Because our trees are balanced, the actual cost of all the operations is always $O(\log n)$, except for flattening and splitting, and their use for balancing.

Our amortized analysis will define three potential functions, ϕ_v, ψ_v, and β_v, for every node v of the tree, and the global potential will be $\Phi = \sum_v (\phi_v + \psi_v + \beta_v)$. We then have that $\Phi = 0$ when the tree is empty. The potentials ϕ_v, ψ_v, and β_v will accumulate work to later pay for the operations of flattening, splitting, and balancing, respectively. They are defined operationally, as follows:

1. Dynamic leaves and internal nodes v are always created with $\phi_v = \beta_v = 0$, but those potentials can increase later. Instead, they always have $\psi_v = 0$.
2. Static leaves v created via flattening start with $\psi_v = 0$ and with $\phi_v = \sum_{u \sqsubseteq v} \phi_u$, where $u \sqsubseteq v$ means that u is v or descends from v. Some of this ϕ_v is later transferred to the potentials ψ_u of static leaves u created upon splitting, as seen later. Note that transfers of potentials do not alter the sum Φ. Static leaves may inherit β_v from former descendants, as seen later.
3. Query operations increment ϕ_v on every visited internal node and dynamic leaf v. If they arrive at a static leaf v, they increase ϕ_v by $\log_{1/\alpha}(v.\text{size}/b)$.
4. Update operations increase β_v by a constant τ, to be defined later, on every visited internal node and dynamic leaf v. They can also trigger splittings, which are analyzed separately.

Consider query operations. Their actual cost is $O(\log n)$. In addition, they increase Φ by $O(\log n)$, in part by increasing ϕ_v by $O(1)$ on the internal nodes and dynamic leaves v they traverse, and in part by increasing ϕ_v on the static leaves they reach by $O(\log n)$. Update operations cost $O(\log n)$ and also increase Φ by $O(\log n)$. Their amortized cost is then also $O(\log n)$. We analyze flattening, splitting, and balancing as separate operations.

Flattening. Now consider the flattening operation. Recall that v is flattened whenever $v.\text{queries} \geq \theta \cdot v.\text{size}$, and that the actual cost of flattening is $v.\text{size}$. Since the updates that traverse v reset $v.\text{queries}$ to zero, it follows that the last $v.\text{queries}$ operations that traversed v have been queries. Each of those incremented the potential ϕ_v, and also increased the potentials ϕ_u of descendants u of v by at least $\log_{1/\alpha}(v.\text{size}/b)$ in total:

- If the query ended in a dynamic leaf, $\log_{1/\alpha}(v.\text{size}/b)$ is the minimum possible distance from v to a dynamic leaf (i.e., when the path to the leaf is as short as possible in the α-balanced tree and that the leaves are all full).
- In case the query arrived at a static leaf u, it increased ϕ_u by $\log_{1/\alpha}(u.\text{size}/b)$. Since the distance between v and u is at least $\log_{1/\alpha}(v.\text{size}/u.\text{size})$, the query also incremented ϕ at that many internal nodes between v and u, and summed with the increase it produced in ϕ_u adds up to $\log_{1/\alpha}(v.\text{size}/b)$.

The potential ϕ_u of the descendants u of v is then $\geq \theta \cdot v.\text{size} \cdot \log_{1/\alpha}(v.\text{size}/b)$. When flattening v, its descendants u disappear and we add their ϕ_u to ϕ_v, which then becomes $\phi_v \geq \theta \cdot v.\text{size} \cdot (1 + \log_{1/\alpha}(v.\text{size}/b))$. Assume $\theta \geq 2$, so $\phi_v \geq v.\text{size} \cdot (2 + \theta \log_{1/\alpha}(v.\text{size}/b))$. From ϕ_v we spend $v.\text{size}$ to pay for the flattening and transfer the rest to ψ_v (which does not alter Φ). Flattening has then zero amortized cost and, after it, it holds $\phi_v = 0$ and $\psi_v \geq v.\text{size} \cdot (1 + \theta \log_{1/\alpha}(v.\text{size}/b))$. The potential ψ_v will be used to pay for future splittings, and will be lost if, instead, some ancestor of v is flattened and makes v disappear.

Splitting. Now assume a static leaf v is split by an update operation. Assuming for simplicity that $v.\text{size}$ is of the form $2^k \cdot \gamma b$ (otherwise only constants change), splitting creates a sequence of static leaves $u_1, u_2, \ldots$ of lengths $u_1.\text{size} = v.\text{size}/2$, $u_2.\text{size} = v.\text{size}/4, \ldots$, until $u_k.\text{size} = u_{k+1}.\text{size} = \gamma b$, where u_1 to u_{k-1} are static and the last two are dynamic. Let the actual cost of splitting be $v.\text{size}$. This cost will be paid from ψ_v, so that splitting has zero amortized cost. The remaining potential in ψ_v, $v.\text{size} \cdot \theta \log_{1/\alpha}(v.\text{size}/b)$, will be transferred to the potentials ψ_{u_i} of the static leaves just created. Concretely, the static leaves u_i are created with $\phi_{u_i} = 0$ and $\psi_{u_i} = u_i.\text{size} \cdot (1 + \theta \log_{1/\alpha}(u_i.\text{size}/b))$, which is what those static leaves need to face their own possible future splittings. There remains enough potential in ψ_v to feed all the new potentials ψ_{u_i} because

$$\begin{aligned}
\sum_{i=1}^{k-1} \psi_{u_i} &= \sum_{i=1}^{k-1} u_i.\text{size} \cdot (1 + \theta \log_{1/\alpha}(u_i.\text{size}/b)) \\
&= \sum_{i=1}^{k-1} \frac{v.\text{size}}{2^i} \cdot \left(1 + \theta \log_{1/\alpha}\left(\frac{v.\text{size}/b}{2^i}\right)\right) \\
&= v.\text{size} \cdot \sum_{i=1}^{k-1} \frac{1}{2^i} \left(1 + \theta(\log_{1/\alpha}(v.\text{size}/b) - i \log_{1/\alpha} 2)\right) \\
&< v.\text{size} \cdot \theta \log_{1/\alpha}(v.\text{size}/b) + v.\text{size} \cdot \left(1 - \frac{1}{2^{k-1}} - \theta\left(2 - \frac{k+1}{2^{k-1}}\right) \log_{1/\alpha} 2\right).
\end{aligned}$$

The first term is what we have available in ψ_v to distribute across the potentials ψ_{u_i}. It suffices that $\theta \geq \log_2(1/\alpha)$ for the second term to be nonpositive for all $k \geq 1$. As we have assumed $\theta \geq 2$ and $\log_2(1/\alpha) \geq 1$, we define $\theta = 1 + \log_2(1/\alpha)$.

Balancing. Balancing on v invokes flattening plus splitting when either $v.\text{left.size} > \alpha \cdot v.\text{size}$ or $v.\text{right.size} > \alpha \cdot v.\text{size}$. Note that, when creating a dynamic node

v by splitting an overflowing leaf, child sizes differ by 1, and when creating v by splitting a static node, they differ by at most γb. We avoid balancing nodes with $v.\text{size} < 5\gamma b$, so as to ensure that $\max(v.\text{left.size}, v.\text{right.size})/v.\text{size} \leq 3/5$ and thus nodes created by splitting do not immediately need balancing (this adds just $O(1)$ to the maximum tree height). We can then use any $3/5 < \alpha < 1$.

It follows that, once a (balanceable) dynamic node v is created, it must undergo i insertions on the larger child (whose initial size can be up to $3/5 \cdot v.\text{size}$) to become unbalanced, because $3/5 \cdot v.\text{size} + i > \alpha(v.\text{size} + i)$. Thus, more than $\frac{\alpha-3/5}{1-\alpha} \cdot v.\text{size}$ insertions must occur in v before balancing takes place. Deletions on the smaller child pose a more stringent condition, as it suffices that d deletions occur, with $3/5 \cdot v.\text{size} > \alpha \cdot (v.\text{size} - d)$, that is, $d > \frac{\alpha-3/5}{\alpha} \cdot v.\text{size}$.

We then set $\tau = \theta/\frac{\alpha-3/5}{\alpha}$ as the constant by which updates increase the potential β_v of the traversed nodes v. Because updates always reach leaves, an update on v increases the potentials β on v and its descendants by at least $\tau(1 + \log_{1/\alpha}(v.\text{size}/b))$. If a dynamic node v must be balanced, at least $\frac{\alpha-3/5}{\alpha} \cdot v.\text{size}$ updates have traversed it, and thus they have increased the potentials β_u, for all $u \sqsubseteq v$, by at least $\frac{\alpha-3/5}{\alpha} \cdot v.\text{size}$ times $\tau(1 + \log_{1/\alpha}(v.\text{size}/b))$, which is $v.\text{size} \cdot \theta(1 + \log_{1/\alpha}(v.\text{size}/b))$. This is exactly the potential needed for flattening (and later splitting) v at zero amortized cost.

The potentials β_v of nodes are maintained upon creation and destruction: merged leaves add up their potentials β, leaves that are split also split their potential β (does not matter how), static leaves created by flattening inherit all the potentials β_u of their destroyed descendants u, and when splitting v we may leave β_v in one of the (one or two) dynamic leaves that are created. The goal is that ancestors of v that may be later balanced preserve the needed potentials β below them. Static leaves v do not need to store β_v for themselves, because when split they will be created as (at least) 2/5–3/5 balanced trees.

Flattening to maintain fill ratios. We also flatten v when $v.\text{size} < (b/3) \cdot v.\text{leaves}$. Note that leaves are created by overflowing, upon an insertion, with fill ratio at least 1/2; only deletions can drive the fill ratio below that fraction. Potential flattening in a subtree of v only improves the average fill ratio, to $\gamma \geq 1/2$ in that subtree. Therefore, a node v must undergo at least $v.\text{size}/6$ deletions before it must be flattened to maintain fill ratios. Just as for balancing, it suffices to assign deletions (i.e., to τ) an additional amortized cost of $\theta/(1/6) = 6\theta$ they pay for the future flattenings (plus possible later splittings) they may trigger.

Since this ensures, in particular, that there are at most $3n/b$ leaves, each using b bits of space, the total space allocated in the leaves is at most $3n$ bits. We may need n additional bits of temporary space when flattening, which sets the maximum usage to $4n$ bits, plus the $o(n)$ bits needed by the tree nodes and static rank/select data structures. Splitting also requires temporary space, but this is less stringent because the $v.\text{size}$ bits are already packed in a static array.

Theorem 1. *An adaptive dynamic bitvector starting empty can be maintained within $4n + o(n)$ bits of space, where n is the current number of bits it represents, so that any operation on it has $O(\log n)$ amortized cost.*

We can reduce the space to $(3+\epsilon)n$ for any constant $\epsilon > 0$, by flattening when $v.\text{size} < (b/(2+\epsilon/2)) \cdot v.\text{leaves}$ and not allocating the b bits for the leaves. Instead, we maintain space for only $(1+\epsilon/2) \cdot v.\text{size}$ bits, and reallocate as necessary.

4.1 Adaptive Analysis

We now show that, if only a fraction $1/q$ of the operations are updates, then the amortized cost per operation is $O(\log(n/q))$. This is clearly true for the updates: though each one costs $O(\log n)$, they are only a fraction $1/q$ of the total, thus their contribution to the global amortized cost is $O(\log(n)/q) \subseteq O(\log(n/q))$.

For the queries, the intuition is that nodes v with $v.\text{size} = \Theta(q)$ are in general static leaves, so the query traverses $O(\log n - \log q)$ nodes to finish. To show that this is the case, we start with a particular regime: consider a sequence that starts on a static leaf of length $n = 2^k \cdot \gamma b$, with one $\mathsf{write}(B, 1)$ and then $q-1$ $\mathsf{access}(B, 1)$ queries, for q to be determined soon. After the split, the first $\theta \cdot 2\gamma b$ accesses will cost $\log_2(n/(\gamma b))$, at which point the parent of the leaf (of $2\gamma b$ bits) will be flattened; the accesses will cost $\log_2(n/(\gamma b)) - 1$ from now. The grandparent needs other $\theta \cdot 2\gamma b$ accesses to reach $\theta \cdot 4\gamma b$ and be flattened in turn; now the access costs will be $\log_2(n/(\gamma b)) - 2$ for other $\theta \cdot 4\gamma b$ accesses, at which point the grandgrandparent reaches $\theta \cdot 8\gamma b$ accesses and is flattened, and so on. After ℓ rounds, the total number of queries is $(1 + \sum_{i=0}^{\ell-1} 2^i) \cdot \theta\gamma b = q - 1$, so let us set $q = 1 + 2^\ell \theta\gamma b$, or $\ell = \log_2((q-1)/(\theta\gamma b))$. The total cost of accesses is

$$\begin{aligned} & \theta\gamma b \cdot \log_2(n/(\theta\gamma)) + \sum_{i=0}^{\ell-1} 2^i \cdot \theta\gamma b \cdot (\log_2(n/(\theta b)) - i) \\ = \;& \theta\gamma b \cdot \left(2^\ell \log_2(n/(\gamma b)) - 2^\ell(\ell-2) - 2\right) \;\leq\; (q-1)\left(\log_2(n/(q-1)) + 2\right). \end{aligned}$$

That is, the cost per query is $O(\log(n/q))$. The cost of the $\ell - 1$ flattenings adds up to $O(2^\ell) = O(q/(\theta\gamma b)) = o(q)$ in total, thus adding $o(1)$ amortized time. Note that we have assumed that $q \geq \theta\gamma b$ for this analysis to hold.

Interestingly, the general case cannot be worse than this particular case. In general, for any q, each update can, via splitting, create a path of new nodes up to depth $\log_2(n/(\gamma b))$. Repeatedly accessing the deepest node in the path can produce a cost over $\log_2(n/q)$ only to the next $q-1$ queries, after which the path is flattened at height $\ell = \log_2((q-1)/(\theta\gamma b))$. Per our analysis of the particular case, the amortized cost incurred by each of those queries is still $O(\log(n/q))$. After this flattening, the extra cost induced by the update is canceled, as further queries will traverse $O(\log(n/q))$ nodes. Deviations from this regime only decrease costs: (i) the updates are most effective in increasing the cost if they open other paths (starting on nodes of size $\Theta(n)$, if possible), disjoint from the current one, so we can assume that no other updates fall in the path during the next $q-1$ queries; (ii) the worst case is that we access the deepest node in the path, because accessing higher leaves does not postpone their flattening.

We still need to consider the costs of flattening, splitting, and balancing, which required amortized analysis. Charging 1 on ϕ_v and τ on β_v for the traversed nodes v does not change the actual cost $O(\log(n/q))$, and as we have seen,

suffices to pay for flattening and balancing, but not for splittings. For those, we had charged $\log_{1/\alpha}(v.\text{size}/b) = O(\log n)$ cost to the queries (to transfer them to ψ_v when needed), independently of the depth of the leaf they reach. This now exceeds our budget, so we analyze splittings in another way, without using ψ_v.

Consider the following model. There is a bag of static leaves that evolves over time, and the tree always has n nodes for simplicity. At any moment, with $\theta\ell$ queries we can create a new static leaf of length ℓ. Along m queries, we have created in total r static leaves of lengths $\ell_1, \ell_2, \ldots, \ell_r$ so that $\sum_{i=1}^{r} \ell_i = m/\theta$. Over each static leaf i, we have applied u_i updates, each producing splits in the most costly way, so that we applied $\sum_{i=1}^{r} u_i = m/q$ updates in total.

To measure how much can $u > 0$ updates over a static leaf v may cost, let $v.\text{size} = \ell = 2^k \cdot \gamma b$ for $k \geq 1$ and $u = 2^d$ for $d \geq 0$; the general case has the same order. The first update costs ℓ, and creates static leaves of lengths $\ell/2$, $\ell/4$, etc. The second chooses the leaf of size $\ell/2$, costing $\ell/2$ and creating leaves of size $\ell/4$, $\ell/8$, etc. Now there are two leaves of size $\ell/4$, which are the next ones chosen, and so on. Let $L(d)$ be the number of leaves of length $\ell/2^d$ created in the process. Because leaves of size $\ell/2^d$ are created by leaves of size $\ell, \ell/2, \ldots, \ell/2^{d-1}$, the recurrence is $L(d) = \sum_{j=0}^{d-1} L(j)$, which solves to $L(0) = 1$ and $L(d) = 2^{d-1}$ for $d > 0$. If we apply updates to all the leaves up to size $\ell/2^d$, we will perform $\sum_{j=0}^{d} L(j) = 2^d$ updates. The cost incurred by those $u = 2^d$ updates is $\sum_{j=0}^{d} (\ell/2^j) \cdot L(j) = \ell(1 + d/2) = \ell(1 + \frac{1}{2}\log_2 u)$.

The maximum possible cost we can produce is then $\sum_{i=1}^{r} \ell_i(1 + \frac{1}{2}\log_2 u_i)$, where $\sum_{i=1}^{r} \ell_i = m/\theta$ and $\sum_{i=1}^{r} u_i = m/q$ (if $u_i = 0$, we assume $1 + \frac{1}{2}\log_2 u_i = 0$). The maximum is obtained when we create $r = m/(\theta n)$ leaves of maximum size $\ell_i = n$ and distribute the updates uniformly on them, $u_i = (m/q)/r$, to split the longest possible leaves. This yields total cost $(m/\theta)(1 + \frac{1}{2}\log_2(\theta n/q)) = O(m \log(n/q))$, or $O(\log(n/q))$ extra amortized cost per operation. If $m < \theta n$, we create only $r = 1$ leaf of length $\ell_1 = m/\theta$ and apply the $u_1 = m/q$ updates on it, yielding maximum cost $O(m \log(m/q)) \subseteq O(m \log(n/q))$. The other border case is $m/q < r$, where the best is to apply $u_i = 1$ updates to m/q leaves of maximum length n, which yields total cost $O((m/q) \log n) \subseteq O(m \log(n/q))$.

Theorem 2. *An adaptive dynamic bitvector starting empty can be maintained in $4n + o(n)$ bits, where n is the current number of bits it represents, so that if the fraction of updates over total operations so far is $1/q$, then the bitvector operations have $O(\log(n/q))$ amortized cost.*

5 Implementation and Experiments

We made a proof-of-concept implementation in C, to illustrate the gain in performance that our data structure obtains as the frequency of updates decreases.

5.1 Implementation

Our machine word has $w = 64$ bits. We strive for a lightweight data structure to answer rank/select, as the time to build it impacts in our overall times as

well. We implement static rank with blocks and superblocks [4, 10] of $4w$ and 2^{16} bits, respectively, storing block counters in 16 bits and superblock counters in 64 bits. The space overhead over the bit data is 6.35%, solving rank with 2 accesses to counters plus at most 4 consecutive access to the bitvector array (plus the popcount operations). Operation select performs interpolation search in the superblocks, then on the blocks, and ends with a linear scan in the 4 words.

Dynamic leaves allocate 32 w-bit words for the bit data (i.e., 2048 bits) and implement all the operations sequentially and word-wise, using bit-parallelism. All the operations are then handled by scanning 32 words, each in constant time.

We use $\alpha = 0.65$ as the balance factor, and $\gamma = 3/4$ as the initial fill ratio of leaves after splitting. We try to avoid overflows by transferring bits to a sibling leaf when possible, leaving both leaves with the same fill ratio, yet we avoid it if the transfer would be less than 1/8 of the leaf space (this avoids a cascade of small transfers as the leaf size approaches the maximum).

A difficult parameterization has been the value of θ, which rules the frequency of flattening. A too large value retains the tree in dynamic form for too long, thereby inducing higher query times. A too low value builds the static subtree too eagerly, just to be soon split again by updates. No single constant worked well for all update frequencies, especially for large n. We opted for finding reasonable values by hand, with the aim of showing the best that can be achieved with proper parameterization. All the rest is implemented as described in the paper.

As a sanity check, we compared with a recent highly optimized dynamic bitvector implementation we call DPR after its authors [6], with $O(\log n)$ time complexity for all the operations, independently of q. The well-known DYNAMIC library [14] was considerably slower, so we omit it in the comparisons.

5.2 Experiments

Our machine is a 64-bit 12th Gen Intel Core i7-1260P at 4.7 GHz, with 16 CPUs and 16GB RAM, running Ubuntu 22.04.4 LTS. We compiled with `gcc -O3`.

We generated random bitvectors of sizes $n = 2^{20}$ to $n = 2^{28}$, built from those a flattened leaf, and carried out $m = n$ operations on them. The operations are insertions of random bits at random positions, $m/(2q)$ times, deletions at random positions, $m/(2q)$ times, and queries in the other cases, running separate experiments with queries access, rank, and select (we did not mix different queries, as we do not expect insights from that). Each experiment was repeated 10 times (100 for $n = 2^{20}$ to gain more precision) and we show average user time.

In a static bitmap, access takes 1, rank 13, and select 42 nanoseconds for $n = 2^{20}$. For $n = 2^{28}$, the times are 14, 27, and 108 nanoseconds for access, rank, and select, respectively, which shows how caching affects even the constant-time algorithms. When all the operations we perform are updates, the times go from over 100 nanoseconds with $n = 2^{20}$ to over 700 nanoseconds for $n = 2^{28}$, one or two orders of magnitude slower than the basic operations. We obtained similar times when using previous results as input to the next operations to forbid parallel execution of conecutive queries.

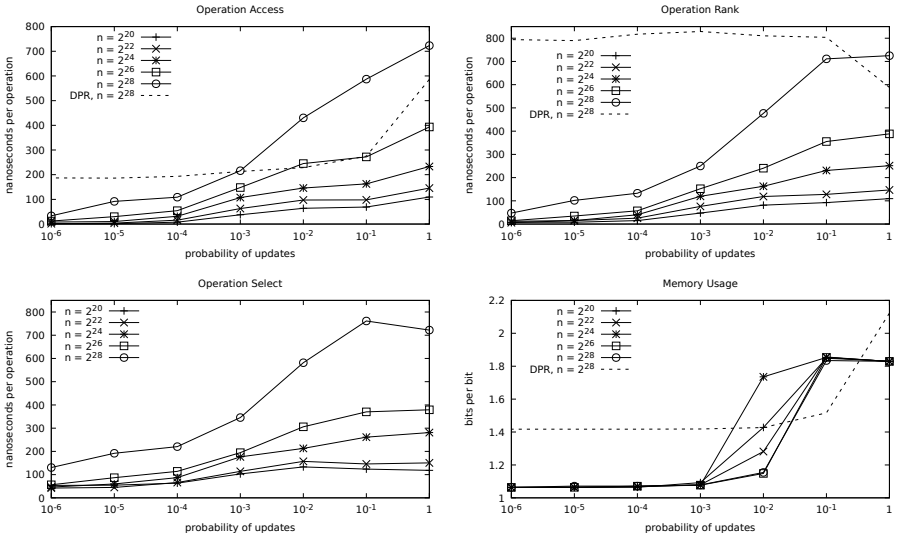

Fig. 2. Average time per operation when mixing queries access, rank, or select, with increasing proportions of updates (insert and delete), on various bitvector sizes. On the bottom right, memory usage of our data structure in bits per bit of the bitvector. DPR did not compile for select in our machine, but we expect times similar to those of rank.

Figure 2 shows the results for increasing values of $1/q$ (the update probability) between 10^{-6} and 1. Times are roughly linear in $\log(1/q)$ (note the logscale in $1/q$), as one would expect from our time complexity $O(\log(n/q)) = O(\log n + \log(1/q))$. DPR is about 15% faster than our implementation when $q = 1$ (all updates) and slower for rank with smaller $1/q$—by a large margin if $1/q \leq 10^{-2}$. For access, DPR is clearly faster if $1/q \geq 10^{-2}$, and clearly slower if $1/q \leq 10^{-4}$.

The figure also shows the use of memory of our data structure, in bits per bit of the bitvector. Note that in practice we always use less than $2n$ bits of space (plus, temporarily, at most n when flattening or splitting). It is clear that, until $1/q = 10^{-3}$, the space overhead is almost the same as for a single static bitvector, meaning that our structure is formed by just a few very long and shallow leaves. Since the leaves contain around 1,000–2,000 bits, this is the last value of $1/q$ for which it is still likely to reach static leaves. Note that a static leaf, even if short and deep, saves considerable time in the last part of the query, completing it in constant time where a dynamic leaf must be scanned. This explains a transition in the slope of times around $1/q = 10^{-3}$, and a saturation point at $1/q = 10^{-1}$, as by then most leaves are likely dynamic and their depth is near $\log n$.

A way to offer better times with higher values of $1/q$ would be to use smaller leaves, at the price of a higher space usage.

6 Conclusions

We have shown how to store a dynamic bitvector $B[1..n]$ within $4n + o(n)$ bits of space so that updates and queries can be solved in $O(\log(n/q))$ amortized time if queries are q times more frequent than updates. Our experiments are in line with our analysis and exhibit speedups of an order of magnitude—and more—compared to classic dynamic data structures, for $q \geq 10^2 .. 10^4$.

Immediate applications of this result are implementations of dynamic sequences by means of wavelet trees or matrices, and tree topologies using parentheses or bits [3, 9, 12], among many others [11, Ch. 12]. Our results can also be extended to maintain dynamic arrays with cells of fixed width (supporting access, write, insert and delete), and sequences of variable-length elements. With the same mechanisms used in previous work [2, 9, 12], this yields a representation of bitvector $B[1..n]$, with entropy H, in $4nH + o(nH)$ bits while supporting the same operations in time $O(\log(n/q))$. In turn, such a representation implements searchable dynamic partial sums with indels [8].

References

1. Andersson, A.: Maintaining α-balanced trees by partial rebuilding. International Journal of Computer Mathematics **38**(1-2), 37–48 (1991)
2. Blandford, D., Blelloch, G.: Compact representations of ordered sets. In: Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 11–19 (2004)
3. Chan, H.L., Hon, W.K., Lam, T.W.: Compressed index for a dynamic collection of texts. In: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 445–456 (2004)
4. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, University of Waterloo, Canada (1996)
5. Dietz, P.: Optimal algorithms for list indexing and subset rank. In: Proc. Workshop on Algorithms and Data Structures (WADS). pp. 39–46 (1989)
6. Dönges, S., Puglisi, S., Raman, R.: On dynamic bitvector implementations. In: Proc. Data Compression Conference (DCC). pp. 252–261 (2022)
7. Fredman, M., Saks, M.: The cell probe complexity of dynamic data structures. In: Proc. 21st Annual ACM Symposium on Theory of Computing (STOC). pp. 345–354 (1989)
8. Hon, W.K., Sadakane, K., Sung, W.K.: Succinct data structures for searchable partial sums. In: Proc. 14th International Symposium on Algorithms and Computation (ISAAC). pp. 505–516 (2003)
9. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms **4**(3), article 32 (2008)
10. Munro, J.I.: Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). pp. 37–42. LNCS 1180 (1996)
11. Navarro, G.: Compact Data Structures – A practical approach. Cambridge University Press (2016)
12. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. ACM Transactions on Algorithms **10**(3), article 16 (2014)

13. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. SIAM Journal on Computing **2**(1), 33–43 (1973)
14. Prezza, N.: A framework of dynamic data structures for string processing. In: Proc. 16th International Symposium on Experimental Algorithms (SEA). pp. 11:1–11:15 (2017)
15. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Proc. 3rd International Symposium on Algorithms and Data Structures (WADS). pp. 426–437 (2001)