Compressed Dynamic Range Majority and Minority Data Structures ⋆

Travis Gagie[1,2], Meng He[3], and Gonzalo Navarro[1,4,5*]

[1] CeBiB — Center for Biotechnology and Bioengineering, Chile
[2] School of Computer Science and Telecommunications, Diego Portales University, Chile, travis.gagie@gmail.com
[3] Faculty of Computer Science, Dalhousie University, Canada, mhe@cs.dal.ca
[4] Millenium Institute for Foundational Research on Data, Chile
[5] Department of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl. (corresponding author)

Abstract. In the range α-majority query problem, we are given a sequence $S[1..n]$ and a fixed threshold $\alpha \in (0,1)$, and are asked to preprocess S such that, given a query range $[i..j]$, we can efficiently report the symbols that occur more than $\alpha(j-i+1)$ times in $S[i..j]$, which are called the range α-majorities.

In this article we describe the first compressed dynamic data structure for range α-majority queries. It represents S in compressed space — $nH_k + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$, where σ is the alphabet size and $H_k \leq H_0 \leq \lg \sigma$ is the k-th order empirical entropy of S — and answers queries in $O\left(\frac{\lg n}{\alpha \lg \lg n}\right)$ time while supporting insertions and deletions in S in $O\left(\frac{\lg n}{\alpha}\right)$ amortized time. We then show how to modify our data structure to receive some $\beta \geq \alpha$ at query time and report the range β-majorities in $O\left(\frac{\lg n}{\beta \lg \lg n}\right)$ time, without increasing the asymptotic space or update-time bounds. The best previous dynamic solution has the same query and update times as ours, but it occupies $O(n)$ words and cannot take advantage of being given a larger threshold β at query time.

We also design the first dynamic data structure for range α-minority — i.e., find a non-α-majority that occurs in a range — and obtain space and time bounds similar to those for α-majorities. We extend the structure to find $\Theta(1/\alpha)$ α-minorities at the same space and time cost.

By giving up updates, we obtain static data structures with query time $O((1/\alpha) \lg \lg_w \sigma)$ for both problems, on a RAM with word size $w = \Omega(\lg n)$ bits, without increasing our space bound. Static alternatives reach time $O(1/\alpha)$, but they compress S only to zero-th order entropy (H_0) or they only handle small values of α, that is, $\lg(1/\alpha) = o(\lg \sigma)$.

⋆ Partially supported by Fondecyt grant 1-171058, Chile; NSERC, Canada; basal funds FB0001, Conicyt, Chile; and the Millenium Institute for Foundational Research on Data, Chile. A preliminary partial version of this article appeared in *Proc. DCC 2017* [16].

1 Introduction

An α-majority in a sequence $S[1..n]$ is a character that occurs more than αn times in S, where the threshold $\alpha \in (0, 1)$. Misra and Gries [25] proposed a two-pass algorithm for finding all α-majorities that runs in $O(1/\alpha)$ space and can be made to run in linear time [9]. In contrast, any algorithm that makes only a constant number of passes over S needs nearly linear space even to estimate the frequency of the mode well [2], where the mode of S is defined as its most frequent element. Thus, finding α-majorities is often considered a practical way to find frequent characters in large files and is important in data mining [12, 9, 23], for example.

For the *range α-majority query* problem, we are asked to preprocess S such that, given a query range $[i..j]$, we can efficiently report the α-majorities of $S[i..j]$, i.e., the symbols that occur more than $\alpha(j - i + 1)$ times in $S[i..j]$. Not surprisingly, this problem seems easier than the range mode query problem [20, 7], in which the query asks for the most frequent element in the query range. Karpinski and Nekrich [24] first considered the range α-majority query problem and proposed a solution that uses $O(n/\alpha)$ words to support queries in $O((\lg \lg n)^2/\alpha)$ time. Durocher *et al.* [10] presented the first solution that achieves optimal $O(1/\alpha)$ query time, and their structure occupies $O(n \lg(1/\alpha))$ words. Subsequent researchers have worked to make the space usage independent of α [17, 8, 5] and even to achieve compression [17, 5]. Among all these works, the most recent one is that of Belazzougui *et al.* [3, 5], who showed how to represent S using $(1 + \epsilon)nH_0 + o(n)$ bits for any constant $\epsilon > 0$ to answer range α-majority queries in $O(1/\alpha)$ time, where H_0 is the 0-th order empirical entropy of S. When more compression is desired, they also showed how to represent S in $nH_0 + o(n)(H_0 + 1)$ bits to support range α-majority in $O(f(n)/\alpha)$ time, for any $f(n) = \omega(1)$. Their solutions work for *variable* α, that is, α is not known at construction time; the value of α is given together with the range $[i, j]$ in each query. We refer readers to their most recent paper [5] for a more thorough survey.

Another line of work is that of *encodings*, which return the positions of the α-majorities in $S[i..j]$ without accessing S at all [28]. Existing encodings use $O(n \lg(1/\alpha))$ bits (which is optimal), possibly less than the entropy of S. The best encoding to date [18] achieves the optimal time, $O(1/\alpha)$. They mention that one can combine such an encoding with a representation of S using $nH_k + o(n \lg \sigma)$ bits and offering constant access time [13] in order to have a competitive static solution, at least for large enough α: the $O(n \lg(1/\alpha))$ bits of the encoding are $o(n \lg \sigma)$ whenever $\lg(1/\alpha) = o(\lg \sigma)$ (e.g., if $1/\alpha = O(\text{polylog}\, \sigma)$). A drawback of encodings is that they work only for fixed α; to have variable α one can build them for all the powers of 2 until reaching $1/\alpha$, which raises their space to $O(n \lg^2(1/\alpha))$ bits.

In the dynamic setting, we wish to maintain support for range α-majority queries under the following update operations on S: i) `insert`(c, i), which inserts symbol c between $A[i-1]$ and $A[i]$, shifting the symbols in positions i through n to positions $i+1$ through $n + 1$, respectively; ii) `delete`(c, i), which deletes $A[i]$, shifting the symbols in positions $i + 1$ through n to positions i through $n - 1$, respectively. Elmasry *et al.* [11] considered this setting, and designed an $O(n)$-word structure that can answer range α-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, supporting insertions and deletions in

$O(\frac{\lg n}{\alpha})$ amortized time. They obtained their results by reducing from another problem, where the data are colored points on the real line. Before their work, Karpinski and Nekrich [24] also considered the colored points problem. With the same reduction [11], the solutions by Karpinski and Nekrich can also be used to encode dynamic sequences, although the results are inferior to those of Elmasry *et al.* [11]. More precisely, their data structures, when combined with the reduction [11], can represent S in $O(n/\alpha)$ words of space, answer queries in time $O(\frac{\lg^2 n}{\alpha})$, and support insertions and deletions in $O(\frac{\lg^2 n}{\alpha})$ amortized time. Alternatively, they can increase the space cost to $O(\frac{n \lg n}{\alpha})$, while decreasing the query and update times to $O(\frac{\lg n}{\alpha})$ worst-case and amortized time, respectively. All the previous work for the dynamic case requires α to be a fixed value given at construction time.

A closely related problem is the *range α-minority query* problem, in which we preprocess a sequence S such that, given a query range $[i..j]$, we can efficiently report one *α-minority* of $S[i..j]$, i.e., a symbol that occurs at least once but not more than $\alpha(j-i+1)$ times in $S[i..j]$, if such a symbol exists, and otherwise return that there is no α-minority in the range. Chan *et al.* [8] studied this problem and designed an $O(n)$-word data structure that answers range α-minority queries in $O(1/\alpha)$ time. Belazzougui *et al.* [3, 5] further designed succinct data structures for range α-minority. They again presented two tradeoffs: they either represent S using $(1+\epsilon)nH_0 + o(n)$ bits for any constant $\epsilon > 0$ to answer range α-minority queries in $O(1/\alpha)$ time, or use $nH_0 + o(n)(H_0+1)$ bits and support range α-minority queries in $O(f(n)/\alpha)$ time, for any $f(n) = \omega(1)$. The solutions of both Chan *et al.* [8] and Belazzougui *et al.* [3, 5] work for variable α. No work has been done for dynamic range α-minority queries.

Our results. In this article we first consider the dynamic range α-majority problem for fixed α and improve the result of Elmasry *et al.* [11] in two key performance aspects: we reduce their space usage while also reducing their time on some more general queries. We describe a data structure that uses even less space than Belazzougui *et al.*'s static representation: $nH_k + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$, where σ is the alphabet size and $H_k \leq H_0 \leq \lg \sigma$ is the k-th order empirical entropy of S. At the same time, while still supporting updates in $O(\frac{\lg n}{\alpha})$ amortized time, we can reduce query times. Specifically, although we still answer range α-majority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, like Elmasry *et al.*, our data structure can receive a threshold $\beta \geq \alpha$ at query time and report the range β-majorities in $O(\frac{\lg n}{\beta \lg \lg n})$ time, rather than $O(\frac{\lg n}{\alpha \lg \lg n})$ time. This type of queries is called *range β-majority queries*. Gagie *et al.* [17] and Chan *et al.* [8] investigated reporting β-majorities in the static setting (i.e., variable α) but no one has previously investigated doing so in the dynamic setting. In summary, our time bounds are at least as good as those by Elmasry *et al.*, our space bound is better to a surprising degree, and our data structure can take advantage of being given a larger threshold at query time in order to answer queries more quickly.

We also design the first solution to the dynamic range α-minority query problem, for fixed α. We can represent S using $nH_k + 2n + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$ to answer range α-minority queries in $O(\frac{\lg n}{\alpha \lg \lg n})$ time, supporting symbol insertions and deletions in $O(\frac{\lg n}{\alpha \lg \lg n})$ amortized time. We show that our structure — and previous

ones — can be extended to report $\Theta(1/\alpha)$ α-minorities within the same space and time complexities, and in general to report m α-minorities by replacing $1/\alpha$ with $m + 1/\alpha$ in all time complexities.

As a byproduct of our main contributions, static versions of our dynamic data structures also use $nH_k + o(n \lg \sigma)$ bits of space ($+2n$ bits in the case of α-minorities), for any $k = o(\lg_\sigma n)$. They support range α-majority queries for variable α, or α-minority queries for fixed α, in time $O((1/\alpha) \lg \lg_w \sigma)$. This time is not far from the $O(1/\alpha)$ achieved by Gagie *et al.* [5] using $(1 + \epsilon)nH_0 + o(n)$ bits, for any constant $\epsilon > 0$, or the times in $(1/\alpha) \cdot \omega(1)$ they achieve within $nH_0 + o(n)(H_0 + 1)$ bits of space. The time $O(1/\alpha)$ is optimal for α-majority queries. The encoding-based static solution for α-majority queries [18] takes $nH_k + o(n \lg \sigma)$ bits and $O(1/\alpha)$ time for fixed α, as long as $\lg(1/\alpha) = o(\lg \sigma)$. For variable α, its range of applicability decreases to $\lg(1/\alpha) = o(\sqrt{\lg \sigma})$.

A preliminary partial version of this article appeared in *Proc. DCC 2017* [16]. Apart from a more complete and detailed presentation, this version includes the support for β-majority queries, the static data structure for α-majority queries, and the dynamic data structure for α-minority queries.

2 Preliminaries

In this section, we summarize some existing data structures that will be used in our solution. One such data structure is designed for the problem of maintaining a string S under `insert` and `delete` operations to support the following operations: `access`(i), which returns $S[i]$; `rank`(c, i), which returns the number of occurrences of character c in $S[1..i]$; and `select`(c, i), which returns the position of the i-th occurrence of c in S. The following lemma summarizes the currently best compressed solution to this problem, which also supports the extraction of an arbitrary substring in optimal time:

Lemma 1 ([26]). *A string of length n over an alphabet of size σ can be represented using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$ to support* `access`*,* `rank`*,* `select`*,* `insert` *and* `delete` *in $O(\lg n / \lg \lg n)$ time. It also supports the extraction of a substring of length l in $O(\lg n / \lg \lg n + l / \lg_\sigma n)$ time.*

Raman *et al.* [30] considered the problem of representing a dynamic integer sequence Q to support the following operations: `sum`(Q, i), which computes $\sum_{j=1}^{i} Q[j]$; `search`(Q, x), which returns the smallest i with `sum`$(Q, i) \geq x$; and `update`(Q, i, δ), which sets $Q[i]$ to $Q[i]+\delta$. One building component of their solution is a data structure for small sequences, which will also be used in our data structures:

Lemma 2 ([30]). *A sequence, Q, of $O(\lg^\epsilon n)$ nonnegative integers of $O(\lg n)$ bits each, where $0 \leq \epsilon < 1$, can be represented using $O(\lg^{1+\epsilon} n)$ bits to support* `sum`*,* `search`*, and* `update`(Q, i, δ) *where $|\delta| \leq \lg n$, in $O(1)$ time. This data structure can be constructed in $O(\lg^\epsilon n)$ time, and requires a precomputed universal table occupying $O(n^{\epsilon'})$ bits for any fixed $\epsilon' > 0$.*

3 Compressed Dynamic Range Majority Data Structures

In this section we design compressed dynamic data structures for range α-majority queries. We define three different types of queries as follows. Given an α-majority query with range $[i..j]$, we compute the size, r, of the query range as $j - i + 1$. If $r \geq L$, where $L = \lceil \frac{1}{\alpha}(\lceil \frac{\lg n}{\lg \lg n} \rceil)^2 \rceil$, then we say that this query is a *large-sized query*. The query is called a *medium-sized query* if $L' < r < L$, where $L' = \lceil \frac{1}{\alpha} \lceil \frac{\lg n}{\lg \lg n} \rceil \rceil$. If $r \leq L'$, then it is a *small-sized query*.

We represent the input sequence S using Lemma 1. This supports small-sized queries immediately: By Lemma 1, we can compute the content of the subsequence $S[i..j]$, where $[i..j]$ is the query range, in $O(\frac{\lg n}{\lg \lg n} + \frac{j-i+1}{\lg_\sigma n}) = O(\frac{\lg n}{\alpha \lg \lg n})$ time. We can then compute the α-majorities in $S[i..j]$ in $O(j - i + 1) = O(\frac{\lg n}{\alpha \lg \lg n})$ time using the algorithm of Misra and Gries [25]. Thus it suffices to construct additional data structures only for large- and medium-sized queries.

3.1 Supporting Large-Sized Range α-Majority Queries

To support large-sized queries, we construct a weight-balanced B-tree [1] T with branching parameter 8 and leaf parameter L. We augment T by adding, for each node, a pointer to the node immediately to its left at the same level, and another pointer to the node immediately to its right. These pointers can be maintained easily under updates, and will not affect the space cost of T asymptotically. Each leaf of T represents a contiguous subsequence, or *block*, of S, and the entire sequence S can be obtained by concatenating all the blocks represented by the leaves of T from left to right. Each internal node of T then represents a block that is the concatenation of all the blocks represented by its leaf descendants. We number the levels of T by $0, 1, 2, \ldots$ from the leaf level to the root level. Thus level a is higher than level b if $a > b$. Let v be a node at the l-th level of T, and let $B(v)$ denote the block it represents. Then, by the properties of weight-balanced B-trees, if v is a leaf, the length of its block, denoted by $|B(v)|$, is at least L and at most $2L - 1$. If v is an internal node, then $\frac{1}{2} \cdot 8^l \cdot L < |B(v)| < 2 \cdot 8^l \cdot L$. We also have that each internal node has at least 2 and at most 32 children.

We do not store the actual content of a block in the corresponding node of T. Instead, for each v, we store the size of the block that it represents, and in addition, compute and store information in a structure $C(v)$ called *candidate list* about symbols that can possibly be the α-majorities of subsequences that meet certain conditions. More precisely, let l be the level of v, u be the parent of v, and $SB(v)$ be the concatenation of the blocks represented by the node immediately to the left of u at level $l+1$, the node u, and the node immediately to the right of u at level $l+1$. Then $C(v)$ contains each symbol that appears more than αb_l times in $SB(v)$, where $b_l = \frac{1}{2} \cdot 8^l \cdot L$ is the minimum size of a block at level l. Since the maximum length of each block at level $l+1$ is $4b_{l+1} = 32b_l$, we have $|SB(v)| \leq 96b_l$, and thus $|C(v)| = O(1/\alpha)$. To show the idea behind the candidate lists, we say that two subsequences *touch* each other if their corresponding sets of indices in S are not disjoint. We then observe that, since the size of any block at level $l+1$ is greater than $8b_l$, any subsequence $S[i..j]$ touching

$B(v)$ is completely contained in $SB(v)$ if $r = j - i + 1$ is within $(b_l, 8b_l)$. Since each α-majority in $S[i..j]$ appears at least $\alpha r > \alpha b_l$ times, it is also contained in $C(v)$. Therefore, to find the α-majority in $S[i..j]$, it suffices to verify whether each element in $C(v)$ is indeed an answer; more details are to be given in our query algorithm later.

Even though it only requires $O(|SB(v)|)$ time to construct $C(v)$ [25], it would be costly to reconstruct it every time an update operation is performed on $SB(v)$. To make the cost of maintaining $C(v)$ acceptable, we only rebuild it periodically by adopting a strategy by Karpinski and Nekrich [24]. More precisely, when we construct $C(v)$, we store symbols that occur more than $\alpha b_l/2$ times in $SB(v)$. We also keep a counter $U(v)$ that we increment whenever we perform `insert` or `delete` in $SB(v)$. Only when $U(v)$ reaches $\alpha b_l/2$ do we reconstruct C_B, and then we reset $U(v)$ to 0. Since at most $\alpha b_l/2$ updates can be performed to $SB(v)$ between two consecutive reconstructions, any symbol that becomes an α-majority in $SB(v)$ any time during these updates must have at least $\alpha b_l/2$ occurrences in $SB(v)$ before these updates are performed. Thus we can guarantee that any symbol that appears more than αb_l times in $SB(v)$ is always contained in $C(v)$ during updates. The size of $C(v)$ is still $O(b_l/\alpha)$, and, as will be shown later, it only requires $O((\lg n)/\alpha)$ amortized time per update to S to maintain all the candidate lists.

We also construct data structures to speed up a top-down traversal in T. These data structures are defined for the *marked* levels of T, where the k-th marked level is level $k\lceil(1/6)\lg\lg n\rceil$ of T for $k = 0, 1, \ldots$. Given a node v at the k-th marked level, the number of its descendants at the $(k-1)$-st marked level is at most $32^{\lceil(1/6)\lg\lg n\rceil - 1} \leq 32^{(1/6)\lg\lg n} = \lg^{5/6} n$. Thus, the sizes of the blocks represented by these descendants, when listed from left to right, form an integer sequence, $Q(v)$, of at most $\lg^{5/6} n$ entries. We represent $Q(v)$ using Lemma 2, and store a sequence of pointers $P(v)$, in which $P(v)[i]$ points to the i-th leftmost descendant at the $(k-1)$-st marked level.

We next prove the following key lemma regarding an arbitrary subsequence $S[i..j]$ of length greater than L, which will be used in our query algorithm:

Lemma 3. *If $r = j - i + 1 > L$, then each α-majority element in $S[i..j]$ is contained in $C(v)$ for any node v at level $l = \lceil \frac{1}{3}\lg\frac{2r}{L} - 1\rceil$ whose block touches $S[i..j]$.*

Proof. Let u be v's parent. Then $S[i..j]$ also touches u, and u is at level $l+1$. Let u_1 and u_2 be the nodes immediately to the left and right of u at level $l+1$, respectively.

Let b_l and b_{l+1} denote the minimum block size represented by nodes at level l and $l+1$ of T, respectively. Then, by the properties of weight-balanced B-trees, if $l > 0$, $b_l = \frac{1}{2}\cdot 8^l \cdot L = \frac{1}{2}\cdot 8^{\lceil\frac{1}{3}\lg\frac{2r}{L}-1\rceil}\cdot L < \frac{1}{2}\cdot 8^{\frac{1}{3}\lg\frac{2r}{L}}\cdot L = r$. When $l = 0$, $b_l = L < r$. Thus, we always have $b_l < r$. Therefore, any α-majority of $S[i..j]$ occurs more than $\alpha r > \alpha b_l$ times in $S[i..j]$.

On the other hand, $b_{l+1} = \frac{1}{2}\cdot 8^{\lceil\frac{1}{3}\lg\frac{2r}{L}\rceil}\cdot L \geq \frac{1}{2}\cdot 8^{\frac{1}{3}\lg\frac{2r}{L}}\cdot L = r$. Since $S[i..j]$ touches $B(u)$, this inequality means that $S[i..j]$ is entirely contained in either the concatenation of $B(u_1)$ and $B(u)$, or the concatenation of $B(u)$ and $B(u_2)$. In either case, $S[i..j]$ is contained in $SB(v)$. Since any α-majority of $S[i..j]$ occurs more than αb_l times in $S[i..j]$, it also occurs more than αb_l times in $SB(v)$. As $C(v)$ includes any symbol that appears more than αb_l times in $SB(v)$, any α-majority of $S[i..j]$ is contained in $C(v)$. □

We now describe our query and update algorithms, and analyze space cost.

Lemma 4. *Large-sized range α-majority queries can be supported in $O(\frac{\lg n}{\alpha \lg \lg n})$ time.*

Proof. Let $[i..j]$ be the query range, $r = j - i + 1$ and $l = \lceil \frac{1}{3} \lg \frac{2r}{L} - 1 \rceil$. We first look for a node v at level l whose block touches $S[i..j]$. The obvious approach is to perform a top-down traversal of T to look for a node at level l whose block contains position i. During the traversal, we make use of the information about the lengths of the blocks represented by the nodes of T to decide which node at the next level to descend to, and to keep track of the starting position in S of the block represented by the node that is currently being visited. More precisely, suppose we visit node u at the current level as we have determined previously that $B(u)$ contains $S[i]$. We also know that the first element in $B(u)$ is $S[p]$. Let $u_1, u_2, \ldots, u_d$ denote the children of u, where $d \leq 32$. To decide which child of u represents a block that contains $S[i]$, we retrieve the lengths of all $|B(u_k)|$'s, and look for the smallest q such that $p + \sum_{k=1}^{q} |B(u_k)| > i$. Node u_q is then the node at the level below whose block contains $S[i]$, and the starting position of its block in S is $p + \sum_{k=1}^{q-1} |B(u_k)|$. As $d \leq 32$ and we store the length of the block that each node represents, these steps use constant time.

However, if we follow the approach described in the previous paragraph, we would use $O(\lg n)$ time in total, as T has $O(\lg n)$ levels. Thus we make use of the additional data structures stored at marked levels to speed up this process. If there is no marked level between the root level and l, then the top down traversal only descends $O(\lg \lg n)$ levels, requiring $O(\lg \lg n)$ time only. Otherwise, we perform the top-down traversal until we reach the highest marked level. Let x be the node we visit at the highest marked level. As $Q(x)$ stores the lengths of the blocks at the next marked level, we can perform a `search` operation in $Q(x)$ and then follow an appropriate pointer in $P(x)$ to look for the node y at the second highest level that contains $S[i]$, and perform a `sum` operation in $Q(x)$ to determine the starting position of $B(y)$ in S. These operations require constant time. We repeat this process until we reach the lowest marked level above level l, and then we descend level by level until we find node v. As there are $O(\lg n / \lg \lg n)$ marked levels, the entire process requires $O(\lg n / \lg \lg n)$ time.

By Lemma 3, we know that the α-majorities of $S[i..j]$ are contained in $C(v)$. We then verify, for each symbol, c, in $C(v)$, whether it is indeed an α-majority by computing its number, m, of occurrences in $S[i..j]$ and comparing m to αr. As $m =$ $\texttt{rank}(c, j) - \texttt{rank}(c, i-1)$, m can be computed in $O(\lg n / \lg \lg n)$ time by Lemma 1. As $|C(v)| = O(1/\alpha)$, it requires $O(\frac{\lg n}{\alpha \lg \lg n})$ time in total to find out which of these symbols should be included in the answer to the query. Therefore, the total query time is $O(\frac{\lg n}{\lg \lg n} + \frac{\lg n}{\alpha \lg \lg n}) = O(\frac{\lg n}{\alpha \lg \lg n})$. □

Lemma 5. *The data structures described in Section 3.1 can be maintained in $O(\frac{\lg n}{\alpha})$ amortized time under update operations.*

Proof. We show only how to support `insert`; the support for `delete` is similar.

To perform $\texttt{insert}(c, i)$, we first perform a top down traversal to look for the node v at level 0 whose block contains $S[i]$. During this traversal, we descend level by level as in Lemma 4, but we do not use the marked levels to speed up the process. For

each node u that we visit, we increment the recorded length of $B(u)$. In addition, we update the counters U stored in the children of u and in the children of the two nodes that surround u. There are a constant number of these nodes, and they can all be located in constant time by following either the edges of T, or the pointers between two nodes that are next to each other at the same level where we augment T.

When incrementing the counter U of each node, we find out whether the candidate list of this node has to be rebuilt. To reconstruct the candidate list of a node x at level l, we first compute the starting and ending positions of $SB(x)$ in S. This can be computed in constant time because, during the top down traversal, we have already computed the starting and ending positions of $B(u)$ in S, and the three nodes whose blocks form $SB(x)$, as well as the sizes of these three blocks, can be retrieved by following a constant number of pointers starting from u. We then extract the content of $SB(x)$. As $|SB(x)| \leq 96b_l$ (see discussions earlier in this section) and $b_l \geq L$, by Lemma 1, $SB(x)$ can be extracted from S in $O(b_l)$ time. We next compute all the symbols that appear in $SB(x)$ more than $\alpha b_l/2$ times in $O(b_l)$ time [25], and these are the elements in the reconstructed $C(x)$. Since the counter $U(x)$ has to reach $\alpha b_l/2$ before $C(x)$ has to be rebuilt, the amortized cost per update is $O(1/\alpha)$.

If u is at a marked level, we perform a `search` operation in $O(1)$ time to locate the entry of $Q(u)$ that corresponds to the node at the next lower marked level whose block contains i, and perform an `update`, again in $O(1)$ time, to increment the value stored in this entry. So far we have used $O(1/\alpha)$ amortized time for each node we visit during the top-down traversal. Since T has $O(\lg n)$ levels, the overall cost we have calculated up to this point is $O((\lg n)/\alpha)$ amortized time.

When a node, z, at level l of T splits into two nodes z_1 and z_2, where z_1 is to the left of z_2, we construct $C(z_1)$ and $C(z_2)$ in $O(b_l)$ time. In addition, for any node y that is a child of z_1 or z_2, or a child of the node immediately to the left of z_1 or the right of z_2 at the same level, we reconstruct $C(y)$ in $O(b_l)$ time. As there are a constant number of such nodes, all these structures can be reconstructed in $O(b_l)$ time. If l is a marked level, but it is not the lowest marked level, we also build $Q(z_1)$, $Q(z_2)$, $P(z_1)$, and $P(z_2)$. We also have to rebuild $P(z')$ and $Q(z')$, where z' is the lowest ancestor of z that is on a marked level. All this takes $O(\lg^{5/6} n) = o(b_l)$ time. By the properties of a weight-balanced B-tree, after a node at level l has been split, it requires at least $\frac{1}{2} \cdot 8^l \cdot L = b_l$ insertions before it can be split again. Therefore, we can amortize the cost of reconstructing these data structures over the insertions between reconstructions, and each `insert` is thus charged with $O(1)$ amortized cost. As each `insert` may cause one node at each level of T to split, the overall cost charged to an `insert` operation is thus $O(\lg n)$.

Finally, update operations may cause the value of L to change. For this to happen, the value of $\lceil \frac{\lg n}{\lg \lg n} \rceil$ must change, and this requires $\Omega(n)$ updates. When this happens, we rebuild our data structure in $O(n \lg n)$ time: we can easily precompute the structures for each level of T in linear time and there are $O(\lg n)$ levels. Thus, such rebuilding incurs $O(\lg n)$ amortized time for each update. To summarize, `insert` can be supported in $O((\lg n)/\alpha)$ amortized time. □

Lemma 6. *The data structures described in Section 3.1 occupy $o(n \lg \sigma)$ bits.*

Proof. As T has $O(n/L)$ nodes, the structure of T, pointers between nodes at the same level, as well as counters and block lengths stored with the nodes, occupy $O(n/L \times \lg n) = O(\frac{\alpha n(\lg\lg n)^2}{\lg n})$ bits in total. Each candidate list can be stored in $O((\lg\sigma)/\alpha)$ bits, so the candidate lists stored in all the nodes use $O(n/L \times (\lg\sigma)/\alpha) = O(\frac{n\lg\sigma(\lg\lg n)^2}{\lg^2 n})$ bits in total. The size of the structures $Q(v)$ and $P(v)$ can be charged to the pointed nodes, so there are $O(n/L)$ entries to store. As each entry of $Q(v)$ uses $O(\lg n)$ bits, all the $Q(v)$s occupy $O(n/L \times \lg n) = O(\frac{\alpha n(\lg\lg n)^2}{\lg n})$ bits. The same analysis applies to $P(v)$. Therefore, the data structures described in this section use $O(\frac{\alpha n(\lg\lg n)^2}{\lg n} + \frac{n\lg\sigma(\lg\lg n)^2}{\lg^2 n}) = o(n\lg\sigma)$ bits. □

3.2 Supporting Medium-Sized Range α-Majority Queries

We could use the same structures designed in Section 3.1 to support medium-sized queries if we simply set the leaf parameter of T to be L' instead of L, but then the resulting data structures would not be succinct. To save space, we build a data structure $D(v)$ for each leaf node v of T. Our idea for supporting medium-sized queries is similar to that for large-sized queries, but since the block represented by a leaf node of T is small, we are able to simplify the idea and the data structures in Section 3.1. Such simplifications allow us to maintain a multi-level decomposition of $B(v)$ in a hierarchy of lists instead of in a tree, which are further laid out in one contiguous chunk of memory for each leaf node of T, to avoid using too much space for pointers.

We now describe this multi-level decomposition of $B(v)$, which will be used to define the data structure components of $D(v)$. As we define one set of data structure components in $D(v)$ for each level of this decomposition, we use $D(v)$ to refer to both the data structure that we build for $B(v)$ and the decomposition of $B(v)$. To distinguish a level of $D(v)$ from a level of T, we number each level of $D(v)$ using a non-positive integer. At level $-l$, for $l = 0, 1, 2, \ldots, \lceil\lg(L/L')-1\rceil$, $B(v)$ is partitioned into *miniblocks* of length between $L/2^l$ and $L/2^{l-1}$. Note that the level 0 decomposition contains simply one miniblock, which is $B(v)$ itself, as the length of any leaf block in T is between L and $2L$ already. We define $m_l = L/2^l$, which is the minimum length of a miniblock at level $-l$. As $L' < m_{\lceil\lg(L/L')-1\rceil} \leq 2L'$, the minimum length of a miniblock at the lowest level, i.e., level $-\lceil\lg(L/L')-1\rceil$, is between L' and $2L'$.

For each miniblock M at level $-l$ of $D(v)$, we define its *predecessor*, $\texttt{pred}(M)$, as follows: If M is not the leftmost miniblock at level $-l$ of $D(v)$, then $\texttt{pred}(M)$ is the miniblock immediately to its left at the same level. Otherwise, if v is not the leftmost leaf ($\texttt{pred}(M)$ is null otherwise), let v_1 be the leaf immediately to the left of v in T, and $\texttt{pred}(M)$ is defined to be the rightmost miniblock at level $-l$ of $D(v_1)$. Similarly, we define the *successor*, $\texttt{succ}(M)$, of M as the miniblock immediately to the right of M at level $-l$ of $D(v)$ if such a miniblock exists. Otherwise, $\texttt{succ}(M)$ is the leftmost miniblock at level $-l$ of $D(v_2)$ where v_2 is the leaf immediately to the right of v in T if v_2 exists, or null otherwise. Then, the candidate list, $C(M)$, of M contains each symbol that occurs more than $\alpha m_l/2$ times in the concatenation of M, $\texttt{pred}(M)$ and $\texttt{succ}(M)$. To maintain $C(M)$ during updates, we use the same strategy in Section 3.1 that is used to maintain $C(v)$. More specifically, we store a counter $U(M)$ so that

we can rebuild $C(M)$ after exactly $\alpha m_l/4$ update operations have been performed to M, $\texttt{pred}(M)$ and $\texttt{succ}(M)$. Whenever we perform the reconstruction, we include in $C(M)$ each symbol that occurs more than $\alpha m_l/4$ times in the concatenation of M, $\texttt{pred}(M)$ and $\texttt{succ}(M)$. Since $|\texttt{pred}(M)| + |M| + |\texttt{succ}(M)| \leq 6m_l$, the number of symbols included in $C(M)$ is at most $24/\alpha$.

The precomputed information for each miniblock M includes $|M|$, $C(M)$, and $U(M)$. These data for miniblocks at the same level, $-l$, of $D(v)$ are chained together in a doubly linked list $L_l(v)$. $D(v)$ then contains these $O(\lg(L/L')) = O(\lg\lg n)$ lists. We cannot, however, afford storing each list in the standard way using pointers of $O(\lg n)$ bits each, as this would use too much space. Instead, we lay them out in a contiguous chunk of memory as follows: We first observe that the number of miniblocks at level $-l$ of $D(v)$ is less than $2L/(L/2^l) = 2^{l+1}$. Thus, the total number of miniblocks across all levels is less than $2 \cdot 2^{\lceil \lg(L/L')-1\rceil+1} - 1 < 4L/L'$. We then use an array $A(v)$ of $\lceil 4L/L' \rceil$ fixed-size *slots* to store $D(v)$, and each slot stores the precomputed information of a miniblock.

To determine the size of a slot, we compute the maximum number of bits needed to encode the precomputed information for each miniblock M. $C(M)$ can be stored in $\lceil 24/\alpha \rceil \cdot \lceil \lg \sigma \rceil$ bits. As M has less than $2L$ elements, its length can be encoded in $\lceil \lg(2L) \rceil$ bits. The counter $U(M)$ can be encoded in $\lceil \lg(\alpha m_l/4) \rceil < \lceil \lg(\alpha L/2) \rceil \leq \lceil \lg(L/2) \rceil$ bits. The two pointers to the neighbours of M in the linked list can be encoded as the indices of these miniblocks in the memory chunk. Since there are $\lceil 4L/L' \rceil$ slots, each pointer can be encoded in $\lceil \lg \lceil 4L/L' \rceil \rceil$ bits. Therefore, we set the size of each slot to be $\lceil 24/\alpha \rceil \cdot \lceil \lg \sigma \rceil + 2\lceil \lg L \rceil + 2\lceil \lg \lceil 4L/L' \rceil \rceil$ bits.

We prepend this memory chunk with a header. This header encodes the indices of the slots that store the head of each $L_l(v)$. As there are $\lceil \lg(L/L') \rceil$ levels and each index can be encoded in $\lceil \lg \lceil 4L/L' \rceil \rceil$ bits, the header uses $\lceil \lg(L/L') \rceil \cdot \lceil \lg \lceil 4L/L' \rceil \rceil$ bits. Clearly our memory management scheme allows us to traverse each doubly linked list $L_l(v)$ easily. When miniblocks merge or split during updates, we need to perform insertions and deletions in the doubly linked lists. To facilitate these updates, we always store the precomputed information for all miniblocks in $D(v)$ in a prefix of $A(v)$, and keep track of the number of used slots of $A(v)$. When we perform an insertion into a list $L_l(v)$, we use the first unused slot of A to store the new information, and update the header if the newly inserted list element becomes the head. When we perform a deletion, we copy the content of the last used slot (let M' be the miniblock that corresponds to it) into the slot corresponding to the deleted element of $L_l(v)$. We also follow the pointers encoded in the slot for M' to locate the neighbours of M' in its doubly linked list, and update pointers in these neighbours that point to M'. If M' is the head of a doubly linked list (we can determine which list it is using $|M'|$), we update the header as well. The following lemma shows that our memory management strategy does, indeed, save space:

Lemma 7. *The data structures described in Section 3.2 occupy $o(n \lg \sigma)$ bits.*

Proof. We first analyze the size of the memory chunk storing $D(v)$ for each leaf v of T. By our analysis in previous paragraphs, we observe that the header of this chunk uses $O((\lg\lg n)^2)$ bits. Each slot of $A(v)$ uses $O(\frac{\lg \sigma}{\alpha} + \lg\lg n)$ bits, and $A(v)$ has

$O(\lg n/\lg\lg n)$ entries. Therefore, $A(v)$ occupies $O(\frac{\lg\sigma\lg n}{\alpha\lg\lg n}+\lg n)$ bits. Hence the total size of the memory chunk of each leaf of T is $O(\frac{\lg\sigma\lg n}{\alpha\lg\lg n}+\lg n)$ bits. As there are $O(n/L)$ leaves in T, the data structures described in this section use $O(\frac{n\lg\sigma\lg\lg n}{\lg n}+\frac{\alpha n(\lg\lg n)^2}{\lg n}) = o(n\lg\sigma)$ bits. □

We now show how to support query and update operations.

Lemma 8. *Medium-sized range α-majority queries can be supported in $O(\frac{\lg n}{\alpha\lg\lg n})$ time.*

Proof. Let $[i..j]$ be the query range and let $r = j-i+1$. We first perform a top down traversal in T to locate the leaf, v, that represents a block containing $S[i]$ in $O(\frac{\lg n}{\lg\lg n})$ time using the approach described in the proof of Lemma 4. In this process, we can also find the starting position of $B(v)$ in S.

We next make use of $D(v)$ to answer the query as follows. Let $l = \lceil\lg(L/r)-1\rceil$. As $m_l = L/2^{\lceil\lg(L/r)-1\rceil}$, we have $m_l/2 \le r < m_l$. We then scan the list $L_l(v)$ to look for a miniblock, M, that contains $S[i]$ at level $-l$. This can be done by first locating the head of $L_l(v)$ from the header of the memory chunk that stores $D(v)$, and then performing a linear scan, computing the starting position of each miniblock in $L_l(v)$ along the way. As $L_l(v)$ has at most $O(L/L') = O(\frac{\lg n}{\lg\lg n})$ entries, we can locate M in $O(\frac{\lg n}{\lg\lg n})$ time. Since $m_l > r$, $S[i..j]$ is either entirely contained in the concatenation of $\texttt{pred}(M)$ and M, or in the concatenation of M and $\texttt{succ}(M)$. Thus each α-majority of $S[i..j]$ must occur more than $\alpha r > \alpha m_l/2$ times in the concatenation of $\texttt{pred}(M)$, M and $\texttt{succ}(M)$. Therefore, each α-majority of $S[i..j]$ is contained in $C(M)$. We can then perform `rank` operations in S to verify whether each symbol in $C(M)$ is indeed an α-majority of $S[i..j]$. As $C(M)$ has $O(1/\alpha)$ symbols, this process requires $O(\frac{\lg n}{\alpha\lg\lg n})$ time. The total query time is hence $O(\frac{\lg n}{\alpha\lg\lg n})$. □

Lemma 9. *The data structures described in Section 3.2 can be maintained in $O(\frac{\lg n}{\lg\lg n}+\frac{\lg\lg n}{\alpha})$ amortized time under update operations.*

Proof. We show only how to support `insert`; the support for `delete` is similar.

To perform $\texttt{insert}(c,i)$, we first perform a top down traversal in T to locate the leaf, v, that represents a block containing $S[i]$ in $O(\frac{\lg n}{\lg\lg n})$ time. We then increment the recorded lengths of all the miniblocks that contain $S[i]$. We also increment the counters U of these miniblocks, as well as the counters of their predecessors and successors. All the miniblocks whose counters should be incremented are located in $D(v)$, $D(v_1)$ and $D(v_2)$, where v_1 and v_2 are the leaves immediately to the left and right of v in T. At each level $-l$, we scan each doubly linked list $L_l(v)$, $L_l(v_1)$ and $L_l(v_2)$ to locate these miniblocks. Since $D(v)$, $D(v_1)$ and $D(v_2)$ have $O(\frac{\lg n}{\lg\lg n})$ miniblocks in total over all levels, it requires $O(\frac{\lg n}{\lg\lg n})$ to find these miniblocks and update them.

The above process can find all these miniblocks, as well as their starting and ending positions in S. It may be necessary to reconstruct the candidate list of these miniblocks. Similarly to the analysis in the proof of Lemma 5, the candidate list of each of these miniblocks can be maintained in $O(1/\alpha)$ amortized time. Since there are

$O(\lg\lg n)$ levels in $D(v)$, $D(v_1)$ and $D(v_2)$, and only a constant number of miniblocks needing rebuilding at each level, $O((\lg\lg n)/\alpha)$ amortized time will be required to reconstruct all of them.

An insertion may also cause a miniblock M to split. As in the proof of Lemma 5, we compute the candidate lists and other required information for the miniblocks created as a result of the split in time linear in the length of M, and amortize the cost over the insertions that lead to the split. As the number of these insertions is also proportional to the length of M, the amortized cost is again $O(1)$. As there can possibly be a split at each level of $D(v)$, it requires $O(\lg\lg n)$ amortized time to handle them. Finally, when the value of L' changes, we rebuild all the data structures designed in this section. Since these data structures are constructed for $O(\lg\lg n)$ levels and the structures for each level can be rebuilt in linear time, this process incurs $O(\lg\lg n)$ amortized time. Therefore, the total time required to support `insert` is $O(\frac{\lg n}{\lg\lg n} + \frac{\lg\lg n}{\alpha})$. □

Combining Lemma 1 and Lemmas 4-9, we obtain our first result, when the structure is queried for α-majorities.

Theorem 1. *For any $0 < \alpha < 1$, a sequence of length n over an alphabet of size σ can be represented using $nH_k + o(n\lg\sigma)$ bits for any $k = o(\lg_\sigma n)$ to answer range α-majority queries in $O(\frac{\lg n}{\alpha\lg\lg n})$ time, and to support symbol insertions and deletions in $O(\frac{\lg n}{\alpha})$ amortized time.*

4 Supporting β-Majorities

Theorem 1 supports range α-majority queries, where α is chosen at construction time. We now enhance our data structure to find range β-majorities, for any $\beta \geq \alpha$ given at query time together with the interval $[i..j]$. While it is easy to answer those queries in time $O(\frac{\lg n}{\alpha\lg\lg n})$, our goal is to reach time $O(\frac{\lg n}{\beta\lg\lg n})$. Updates are still carried out in amortized time $O(\frac{\lg n}{\alpha})$.

Although we have not used this in previous sections, note that we can focus our attention in the case $\beta > 1/\sigma$, since otherwise we can directly check the range of S for each of the σ symbols c, reporting those where $\texttt{rank}(c,j) - \texttt{rank}(c,i-1) > \beta r$, all in time $O(\frac{\sigma\lg n}{\lg\lg n}) = O(\frac{\lg n}{\beta\lg\lg n})$. Thus, at construction time we can set α to $1/\sigma$ if α turns out to be smaller. This implies, in particular, that all our $\frac{1}{\alpha}$ in the complexities can be replaced by $\min(\frac{1}{\alpha}, \sigma)$. We will also use the fact that $\lg\frac{1}{\alpha} = O(\lg\sigma) \cap O(\lg n)$. Similarly, it makes sense to consider $\alpha \leq 1/2$ only, as otherwise we use the solution for $\alpha = 1/2$ and report only the true α-majorities found, within the same complexity.

4.1 Large and Medium-Sized Intervals

For large and medium-sized intervals, it is not difficult to answer β-majority queries within the desired time. Note that, in those cases, the crux of the solution is to verify a list of candidates, $C(v)$ in the block v (for large intervals) or $C(M)$ in the miniblock M (for medium-sized intervals), both of size $O(1/\alpha)$. It is sufficient that those lists are sorted by decreasing frequency of the elements and that we stop verifying them

when reaching an element with frequency below βr. Since $r > b_l$ in large intervals and $r \geq m_l/2$ in medium-sized intervals, there can be only $O(1/\beta)$ such candidates and we solve the query in time $O(\frac{\lg n}{\beta \lg\lg n})$.

We can maintain those list only approximately sorted, however. When the lists are created, we do sort them by decreasing frequency, but the elements can later change their frequencies upon updates. The order in which we store the symbols is not modified upon updates until the lists are rebuilt. Because frequencies can change only by a maximum of $\gamma = \alpha b_l/2$ (for large intervals) or $\gamma = \alpha m_l/4$ (for medium-sized intervals) before we rebuild the lists, we can safely stop verifying when the frequency we compute on the fly drops below $\beta r - \gamma$, since this guarantees that the next element cannot have a current frequency over βr. Since $r > b_l$ for large intervals and $r \geq m_l/2$ for medium-sized intervals, it holds that $\beta r - \gamma > \beta b_l/2$ for large intervals and $\beta r - \gamma \geq \beta m_l/4$ for medium-sized intervals, and therefore the resulting complexity is in both cases $O(\frac{\lg n}{\beta \lg\lg n})$.

4.2 Small Intervals

The small ranges, which were solved by brute force, pose a more difficult problem, because now we cannot afford scanning a block of S of size $O(\frac{\lg n}{\alpha \lg\lg n})$. To handle small ranges, we add further structures to our tree leaves, which contain L' to $2L'$ elements for $L' = \lceil \frac{1}{\alpha} \lceil \frac{\lg n}{\lg\lg n} \rceil \rceil$. The leaves will be further partitioned into halves repeatedly in $\lg(1/\alpha)$ levels, until reaching size between L^* and $2L^*$, for $L^* = \lceil \frac{\lg n}{\lg\lg n} \rceil$.

These additional levels, numbered $-l^*$ for $l = 0, \ldots, \lfloor \lg(L'/L^*) \rfloor$, are organized much as the miniblocks of Section 3.2. Indeed, our highest level, -0^*, is the same level, $-\lceil \lg(L/L') \rceil$, as the deepest one of Section 3.2. The main difference is that, in the new levels, not only the sizes m_{l^*} are halved as we descend, but also the majority thresholds are doubled: we use the value $\alpha_{l^*} = \alpha \cdot 2^{l^*}$ to define the candidate lists $C(M)$ at level $-l^*$. In our last level, $-l^* = -\lfloor \lg(L'/L^*) \rfloor$, it holds that $\alpha_{l^*} = \Theta(1)$ (precisely, $\alpha_{l^*} > 1/2$) and $m_{l^*} = O(\frac{\lg n}{\lg\lg n})$ (precisely, $L^* \leq m_{l^*} \leq 2L^*$).

Because, at each level $-l^*$, $C(M)$ can store at most $24/\alpha_{l^*} = 24/(\alpha\, 2^{l^*})$ elements, the miniblocks of different levels $-l^*$ are of different size. We then do not store the information of all the miniblocks descending from a leaf block in a single chunk of memory, as done in Section 3.2. Instead, we stratify the storage of miniblocks per level $-l^*$: For each leaf block, we have an array of $O(\lg \frac{1}{\alpha})$ entries, one per level $-l^*$, to memory areas of miniblocks of that level descending from the leaf block. Within each memory area, the slots are of the same size, as in Section 3.2.

We also impose further structure to the linked lists of miniblocks inside each memory area: the list nodes are not anymore linked, but they are the leaves of a B-tree of arity B to $2B$, for $B = \sqrt{\lg n}$. Since the list at level $-l^*$ has $2^{l^*} < \frac{1}{\alpha} + 1$ elements, the B-tree is of height $O(\lg(1/\alpha)/\lg\lg n)$. Each B-tree node stores the up to $2\sqrt{\lg n}$ subtree sizes (measured in terms of number of positions of S stored in all the subtree leaves) using Lemma 2, which allows routing the search for a given position in S in constant time per B-tree node. To facilitate memory management, we have one memory area for the B-tree nodes and another for the list nodes, so that memory slots are of the same size within each area.

Finally, we use a new arrangement to store the lists $C(M)$ in these miniblocks. Instead of representing the candidate symbols directly, we store one position of $\texttt{pred}(M) \cdot M \cdot \texttt{succ}(M)$ where the symbol appears. The actual symbol can then be obtained with an access to S in time $O(\frac{\lg n}{\lg\lg n})$. Further, we sort all the $O(1/\alpha_{l^*})$ positions of the candidates as follows: The primary criterion for the sort is $\lceil \lg(1/f) \rceil$, where f is the relative frequency of the element in $\texttt{pred}(M) \cdot M \cdot \texttt{succ}(M)$. The secondary criterion, when the first produces ties, is the increasing order of the positions in $\texttt{pred}(M) \cdot M \cdot \texttt{succ}(M)$ we use to represent the symbols.

Therefore, the list $C(M)$ is partitioned into $O(\lg n)$ *chunks* of symbols with the same quantized frequency, $q_f = \lceil \lg(1/f) \rceil$, and the positions stored are increasing within each chunk. Those chunks are then represented as the differences between consecutive positions using γ-codes [6], and a difference of zero is used to signal the end of a chunk. By Jensen's Inequality, the number of bits required to represent k differences that add up to m is $O(k \lg(m/k))$.[6] Since there are at most 2^{q_f} elements with quantized frequency q_f, their chunk is represented with $O(2^{q_f}(\lg(m) - q_f))$ bits. Adding up to relative frequency $f^* = \alpha_{l^*}/24$ (i.e., the minimum for a candidate stored in $C(M)$), the total space in bits is at most

$$\sum_{q_f=0}^{q_f=\lg\lceil 24/\alpha_{l^*}\rceil} 2^{q_f}(\lg m - q_f) = 2\left\lceil \frac{24}{\alpha_{l^*}} \right\rceil \left(\lg m - \lg \left\lceil \frac{24}{\alpha_{l^*}} \right\rceil + 1 \right) - \lg m - 2 = O\left(\frac{\lg(\alpha_{l^*} m)}{\alpha_{l^*}} \right),$$

and since in our case $m = O(m_{l^*}) = O(\frac{\lg n}{\alpha 2^{l^*} \lg\lg n}) = O(\frac{\lg n}{\alpha_{l^*} \lg\lg n})$, the total space to represent $C(M)$ is $O((1/\alpha_{l^*}) \lg\lg n)$.

Lemma 10. *The data structures described in Section 4 occupy $o(n \lg \sigma)$ bits.*

Proof. The analysis is analogous to that of Lemma 7. The number of miniblocks at level $-l^*$ of each array $A(v)$ is at most $2L/(L'/2^{l^*}) = O(\frac{\lg n}{\lg\lg n} \cdot 2^{l^*+1})$. The size of the miniblocks includes the space to store the list of candidates, $O((1/\alpha_{l^*}) \lg\lg n)$ bits, plus a constant number of $(\lg L)$-bit counters and pointers, which require $O(\lg\lg n + \lg \frac{1}{\alpha})$ further bits. The B-tree nodes, stored in another memory area, require $O(B \lg L)$ bits per node, but have $O(1/B)$ nodes per miniblock M, thus their space is already covered in our formula. All this adds up to $O((1/\alpha_{l^*}) \lg\lg n + \lg \frac{1}{\alpha})$ bits per miniblock, which multiplied by the number of miniblocks at level $-l^*$ of $A(v)$ yields $O(\frac{\lg n}{\alpha} + \frac{\lg n \lg \frac{1}{\alpha}}{\lg\lg n} \cdot 2^{l^*})$ bits. Summing up this space over all the $\lg \frac{1}{\alpha}$ levels $-l^*$, we obtain $O(\frac{\lg n \lg \frac{1}{\alpha}}{\alpha} + \frac{\lg n \lg \frac{1}{\alpha}}{\alpha \lg\lg n}) = O(\frac{\lg n \lg \frac{1}{\alpha}}{\alpha})$ bits. Finally, multiplying this space by the $O(n/L)$ leaves, we obtain $O(\frac{n \lg \frac{1}{\alpha} (\lg\lg n)^2}{\lg n}) = o(n \lg \sigma)$ bits.

We also have $O(\lg \frac{1}{\alpha})$ global pointers for the memory areas of each level $-l^*$, which multiplied by the $O(n/L)$ leaves yields $O(\frac{\alpha n \lg \sigma (\lg\lg n)^2}{\lg n}) = o(n \lg \sigma)$ bits in total. □

Now we show how to support range β-majority queries with this structure.

[6] Since γ-codes can only represent positive numbers and we want to use zero to signal end of chunks, we will always use the code for $x + 1$ to represent the number x. This adds only $O(k)$ extra bits.

Lemma 11. *Small-sized range β-majority queries, for any $\beta \geq \alpha$, can be supported in $O(\frac{\lg n}{\beta \lg\lg n})$ time.*

Proof. After we arrive at the corresponding leaf block v in time $O(\frac{\lg n}{\lg\lg n})$ as in the proof of Lemma 8, we choose the level $-l^*$ according to $r = j-i+1$: it must hold that $m_{l^*}/2 \leq r < m_{l^*}$, i.e., $\frac{\lg n}{2r\lg\lg n} \leq \alpha_{l^*} < \frac{\lg n}{r\lg\lg n}$. This level is appropriate to apply the same reasoning of Lemma 8, and it exists whenever $2L^* \leq r < L'$. On the other hand, we need that $\alpha_{l^*} \leq \beta$ in order to ensure that the candidates stored in $C(M)$ (which include all the possible α_{l^*}-majorities) include all the possible β-majorities. Since $\alpha_{l^*} < \frac{\lg n}{r\lg\lg n}$, it suffices that $\frac{\lg n}{r\lg\lg n} \leq \beta$ to have $\alpha_{l^*} \leq \beta$. This condition is equivalent to $r \geq \frac{\lg n}{\beta\lg\lg n}$. We can always assume this condition to be true, since otherwise we can use Lemma 1 to extract $S[i..j]$ and find its majorities in time $O(\frac{\lg n}{\beta\lg\lg n})$ without the help of any other data structure.

We then traverse the B-tree of level $-l^*$ so as to find the appropriate miniblock M. The traversal takes time $O(\frac{\lg(1/\alpha)}{\lg\lg n}) = O(\frac{\lg n}{\lg\lg n})$. Once we arrive at the proper miniblock M, we scan the successive chunks of $C(M)$. Since the frequencies in the next chunk (at the moment of list construction) could not be more than those in the current chunk, and since some frequency may have increased by at most $\alpha_{l^*}m_{l^*}/4$ since the last reconstruction, we can safely stop the scan when the highest frequency seen in the current chunk does not exceed $\beta r - \alpha_{l^*}m_{l^*}/4$.

Let us analyze the cost we incur to scan up to this threshold. Since frequencies can also decrease by up to $m_{l^*}/4$ until the next reconstruction, an element with current frequency over $\beta r - \alpha_{l^*}m_{l^*}/4$ must have had frequency over $\beta r - \alpha_{l^*}m_{l^*}/2 \geq (\beta - \alpha_{l^*})m_{l^*}/2$ when the list was built, and thus its relative frequency was $f \geq (\beta-\alpha_{l^*})/12$. Its quantized frequency was therefore $q_f \leq \lceil \lg(12/(\beta - \alpha_{l^*}))\rceil$, and thus we might have to process up to $2^{q_f+1} = O(\frac{1}{\beta-\alpha_{l^*}})$ elements before covering its chunk. This is $O(1/\beta)$ if $\beta \geq 2\alpha_{l^*}$; otherwise we might use the argument that the whole list is of size $O(1/\alpha_{l^*}) = O(1/\beta)$ anyway. Therefore, we try out each candidate using `rank` on S in time $O(\frac{\lg n}{\beta\lg\lg n})$ and complete the query. □

Finally, we show that we can still maintain the structure within the original time.

Lemma 12. *The data structures described in Section 4 can be maintained in amortized time $O(\frac{\lg^2(1/\alpha)}{\lg\lg n} + \frac{1}{\alpha} + \frac{\lg n}{\lg\lg n})$ under update operations.*

Proof. For large and medium blocks, the only difference is that we must sort the candidate lists $C(v)$ and $C(M)$ by decreasing frequency. This is not difficult because we already spend time $O(b_l)$ (for large ranges) and $O(m_l)$ (for medium-sized ranges) in building the lists. The frequencies range over a universe of the same size, thus we can sort them within the same times, $O(b_l)$ or $O(m_l)$, using radix sort.

The maintenance procedure for the levels $-l^*$ is very similar to that of miniblocks described in Lemma 9. One difference is that, when changes in a miniblock M occurs, we must update its size upwards in its B-tree. This adds $O(\frac{\lg(1/\alpha)}{\lg\lg n})$ time, because Lemma 2 allows us update each B-tree node in constant time. Node splits and merges require $O(B)$ time, but these amortize to $O(1)$. Finally, a single update requires modifying the B-trees in all the $\lg(1/\alpha)$ levels $-l^*$, for a total update cost of $O(\frac{\lg^2(1/\alpha)}{\lg\lg n})$.

The amortized cost to reconstruct a list $C(M)$ at level $-l^*$ is $O(1/\alpha_{l^*})$, including the special sorting and encoding we use. Since a single update is reflected in all the levels, we must add up this cost for all $-l^*$, yielding $O(1/\alpha)$. Updates also need time $O(\frac{\lg n}{\lg\lg n})$ to reach the desired miniblock. □

We now have all the elements to prove our main result. Note that $O(\frac{\lg n}{\alpha})$ encompasses all the update costs for the three range sizes.

Theorem 2. *For any $0 < \alpha < 1$, a sequence of length n over an alphabet of size σ can be represented using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$ to answer range β-majority queries for any $\beta \geq \alpha$ in $O(\frac{\lg n}{\beta \lg\lg n})$ time, and to support symbol insertions and deletions in $O(\frac{\lg n}{\alpha})$ amortized time.*

4.3 A Static Variant

A static variant of our solutions uses blocks and miniblocks of fixed size, so one can access in constant time the desired block at the corresponding level l, $-l$, or $-l^*$, and then try out the prefix of $O(1/\beta)$ stored candidates that covers all the possible β-majorities. All that precomputed data amounts to $o(n \lg \sigma)$ bits of space, even when built for the minimum α of interest, $\max(1/n, 1/\sigma)$, as shown in Lemmas 6, 7, and 10. Using a sequence representation that uses $nH_k + o(n \lg \sigma)$ bits [4, Thm. 11] and answers access queries in time $O(1)$ and rank queries in time $O(\lg \lg_w \sigma)$, where $w = \Omega(\lg n)$ is the RAM word size in bits, we can solve β-majority queries in time $O((1/\beta) \lg \lg_w \sigma)$.

Since the structure has $O(\lg n)$ levels and each level is built in linear time as described in Lemmas 5, 9, and 12, the construction time is $O(n \lg n)$. The sequence representation we use [4, Thm. 11] is built in linear time.

Interestingly, since update times are irrelevant in this case, the asymptotic time and space complexities of our data structure do not depend on α. Thus, our structure can be built directly for the minimum relevant value of α, $\max(1/n, 1/\sigma)$, and then it can be queried for any value of β (if $\beta \leq \max(1/n, 1/\sigma)$, we just try out all the σ possible candidates using `rank` on $S[i..j]$). Thus, this data structure can be used to answer range α-majorities for variable α, which is even more powerful than the range β-majority query. The following theorem presents our result, which is stated as a solution to the range α-majority problem for variable α.

Theorem 3. *On a RAM machine of $w = \Omega(\lg n)$ bits, a sequence of length n over an alphabet of size σ can be represented using $nH_k + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$ to answer range α-majority queries for any $0 < \alpha < 1$ defined at query time, in $O((1/\alpha) \lg \lg_w \sigma)$ time. The structure is built in $O(n \lg n)$ time.*

5 Finding α-Minorities

We now introduce the first dynamic structure to find α-minorities in array ranges. We build on the idea of Chan *et al.* [8], who find $A = 1 + \lfloor 1/\alpha \rfloor$ distinct elements

in $S[i..j]$ and try them out one by one, since one of those must be a minority (there may be no minority if there are less than A distinct elements in $S[i..j]$). A succinct static structure based on this idea [3] uses an $O(n)$-bit range minimum query data structure [14], of which no dynamic succinct version exists.

We use a different dynamic arrangement that can be implemented in succinct space. We partition S into *pieces*, which contain A to $3A$ *distinct* elements, except when S contains a single piece with less than A distinct elements. The following property is the key to find an α-minority in time $O(\frac{\lg n}{\alpha \lg \lg n})$.

Lemma 13. *If $S[i..j]$ overlaps one or two pieces only and it has an α-minority, then this minority element is one of the distinct elements in those pieces. If $S[i..j]$ contains a piece with at least A distinct elements, then one of the distinct elements in that contained piece is a minority in $S[i..j]$.*

Proof. If $S[i..j]$ overlaps one or two pieces only, then all of its distinct elements are also distinct elements in some of those overlapped pieces, so the result holds. If $S[i..j]$ contains a piece with A distinct elements, then one of those must be an α-minority of $S[i..j]$, since not all of them can occur more than $\alpha \cdot (j-i+1)$ times in $S[i..j]$. □

Our data structure is formed by a compressed dynamic representation of S using $nH_k + o(n \lg \sigma)$ bits (Lemma 1) plus two dynamic bitvectors that add $2n + o(n)$ bits:

1. $P[1..n]$, where $P[r] = 1$ iff a new piece starts at $S[r]$.
2. $C[1..n]$, where each distinct element in each piece has one arbitrary occurrence at position r (within the piece) marked with $C[r] = 1$.

The dynamic bitvectors support the operations `access`, `rank`, `select`, `insert`, and `delete`, in time $O(\lg n / \lg \lg n)$ (see [27, Lem. 8.1] or [21]).

To find an α-minority in $S[i..j]$, we use `rank` and `select` on P to determine the first and the last piece overlapped by $[i..j]$. More precisely, we compute the starting position of the first of these pieces as $x = \texttt{select}_1(P, \texttt{rank}_1(P, i))$ and the ending position of the last as $y = \texttt{select}_1(P, \texttt{rank}_1(P, j) + 1) - 1$. If there are one or two pieces overlapped by $[i..j]$, that is, $\texttt{rank}_1(P, y) - \texttt{rank}_1(P, x-1) \leq 2$, we try out all their at most $6A$ distinct elements as follows: For $k = 1, 2, \ldots$, we find their k-th distinct element, c, in $S[i..j]$ using the formula $c = S[p]$ for $p = \texttt{select}_1(C, \texttt{rank}_1(C, x - 1) + k)]$. We then compute $\texttt{rank}_c(S, j) - \texttt{rank}_c(S, i-1)$ to count how many times c occurs in $S[i..j]$ and thus determine whether c is an α-minority. We repeat this process until we find and return an α-minority, or until $p > y$, in which case we report that there is no α-minority in the query range. If $S[i..j]$ overlaps 3 pieces or more, we choose its leftmost contained piece (the left and right endpoints of this piece are $\texttt{select}_1(P, \texttt{rank}_1(P, i-1)+1)$ and $\texttt{select}_1(P, \texttt{rank}_1(P, i-1)+2)-1$, respectively), and do as before to obtain its A to $3A$ candidates and count their occurrences in S. This process yields, in time $O(\frac{\lg n}{\alpha \lg \lg n})$, an α-minority of $S[i..j]$, if there is one.

5.1 Handling Updates

To insert a new symbol c at position i in S, we first do the insertion in S, and also insert a 0 in $P[i]$ and $C[i]$. We then find the piece $P[x..y]$ where $P[i]$ belongs,

using $x = \texttt{select}_1(P, \texttt{rank}_1(P, i))$ and $y = \texttt{select}_1(\texttt{rank}_1(P, i) + 1) - 1$. Finally, if $\texttt{rank}_c(S, y) - \texttt{rank}_c(S, x - 1) = 1$, then c is a new distinct symbol in the piece and we must mark it, with $C[i] \leftarrow 1$.

This completes the insertion process unless we exceed the maximum number of distinct element in the piece, that is, $\texttt{rank}_1(C, y) - \texttt{rank}_1(C, x-1) > 3A$. In this case, we *repartition* the piece into pieces of size A to $3A$.

The repartitioning proceeds as follows. We locate the first occurrences of the distinct elements in the piece, using `rank` and `select` on C and S: For $k = 1, 2, \ldots$, the kth distinct symbol is $c_k = S[i_k]$, with $i_k = \texttt{select}_1(C, \texttt{rank}_1(C, x - 1) + k)$. The first occurrence of c_k in the piece is then $p_k = \texttt{select}_{c_k}(S, \texttt{rank}_{c_k}(S, x - 1) + 1)$. We unmark its position in C, $C[i_k] \leftarrow 0$ (note that i_k needs not be the first occurrence, p_k, of the kth distinct element).

Once we have collected all the first positions p_k of the $O(A)$ distinct elements in the piece, we use the classic algorithm that computes order statistics in linear time (i.e., $O(A)$) on the positions p_k to find the $(A+1)$st smallest of them, p. The first new piece, with exactly A distinct elements, then goes from x to $p-1$, so we set $P[p] \leftarrow 1$ and mark in C the A positions we had located, $C[p_k] \leftarrow 1$ for the A values $p_k < p$.

We now create the next new pieces from $S[p..y]$. To do this, we simply replace those values $p_k < p$ with their first occurrence in $S[p..y]$ using `rank` and `select` on S again, compute the $(A+1)$st smallest position again, and so on, until covering the whole original overflowed piece $S[x..y]$.

This process generates a number of pieces with A distinct elements, except the last one, which may have fewer. In this case, we merge the last two pieces built, into one that will have less than $2A$ distinct elements. Those are found by repeating the generation of the penultimate piece, this time not marking only the A smallest positions p_k, but including them all. Overall, each new piece is built in time $O(\frac{\lg n}{\alpha \lg \lg n})$.

To delete $c = S[i]$, we remove the position i from sequences S, P, and C. If it holds that $P[i] = 1$ before removing it, then we had deleted the mark of the beginning of the piece, so we reset $P[i] \leftarrow 1$ again after removing $P[i]$. If it holds that $C[i] = 1$ before removing it, we must see if there is another occurrence of c in its piece. We compute the piece endpoints x and y as for the insertion, and then see if $\texttt{rank}_c(S, y) - \texttt{rank}_c(S, x - 1) \geq 1$. If so, we set another occurrence of c in C, for example the first, $C[\texttt{select}_c(S, \texttt{rank}_c(S, x - 1) + 1)] \leftarrow 1$. Otherwise, we have lost a distinct element in the piece and must see if we still have sufficiently many distinct elements, that is, if $\texttt{rank}_1(C, y) - \texttt{rank}_1(C, x - 1) \geq A$. If this is true, we finish.

Otherwise, we have less than A distinct elements in the piece, so we merge it with the previous or next piece (if none exists, then S has only one piece, which can have less than A distinct elements). The merged piece has at least A distinct elements, but it might have up to $4A - 1$ and thus overflow. The merging then consists of removing the intermediate 1 in P that separates the two pieces and running our repartitioning process described above. The cost will be, again, $O(\frac{\lg n}{\alpha \lg \lg n})$ per piece generated.

5.2 Analysis

Our partitioning process may require time proportional to the length of the piece, which can be arbitrarily longer than $3A$. Consider for example $A = 2$ and the piece $(abc)^n def$. Inserting a g at the end produces $\frac{3}{2}n + 2$ pieces of length 2. We can show, however, that the amortized cost of a repartitioning is within the desired time bounds. We will measure the cost in terms of number of operations over the sequences, each of which costs $O(\frac{\lg n}{\lg\lg n})$.

Let us define a potential function $\phi = n - A \cdot (m - 1)$, where m is the number of pieces at the present moment. It always holds $\phi \geq 0$, even when we start with $n = 1$ element and $m = 1$ piece. Then an insertion or a deletion without overflow or underflow modifies ϕ by $\Delta\phi = \pm 1$, which does not affect the asymptotic cost. A repartitioning, instead, takes a piece and produces $t > 1$ pieces out of it. The actual cost of generating each piece, ignoring constants, is A operations on the sequences; therefore the total cost of the operation is $A \cdot t$. On the other hand, the difference in potential is $\Delta\phi = A \cdot (1 - t)$ (n does not change while repartitioning). Therefore, the amortized cost of the repartitioning is A.

The case of an underflow is similar: we first join two pieces, which increases ϕ by A. We then repartition the resulting piece into t, which costs $A \cdot t$ and changes the potential by $\Delta\phi = A \cdot (1 - t)$. In total, the amortized cost of an underflow is $2A$.

Since $A = O(1/\alpha)$ and the operations we are counting cost $O(\frac{\lg n}{\lg\lg n})$, the amortized cost of the operations `insert` and `delete` is $O(\frac{\lg n}{\alpha \lg\lg n})$.

Theorem 4. *For any $0 < \alpha < 1$, a sequence of length n over an alphabet of size σ can be represented using $nH_k + 2n + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$ to answer range α-minority queries in $O(\frac{\lg n}{\alpha \lg\lg n})$ time, and to support symbol insertions and deletions in $O(\frac{\lg n}{\alpha \lg\lg n})$ amortized time.*

By using the idea in static form, we also obtain the first solution for α-minority queries using high-order entropy space. Here the bitvectors take constant time to answer `access`, `rank` and `select` queries, and the static sequence representation [4, Thm. 11] yields time $O(\lg \lg_w \sigma)$ per operation. The construction is easily done in linear time.

Theorem 5. *On a RAM machine of $w = \Omega(\lg n)$ bits, a sequence of length n over an alphabet of size σ can be represented using $nH_k + 2n + o(n \lg \sigma)$ bits for any $k = o(\lg_\sigma n)$, to answer range α-minority queries in $O((1/\alpha) \lg \lg_w \sigma)$ time. The structure is built in $O(n)$ time.*

The static result, however, is not a merit of our novel piece partitioning. In fact, it is not hard to obtain the same space and time from previous work [3], by using the base sequence representation we use [4, Thm. 11] (the base representation chosen by Belazzougui et al. [3] compresses to nearly $nH_0 \geq nH_k$, but it yields better query time, $O(1/\alpha)$).

More interestingly, following the literature, we have considered the problem of spotting *one* α-minority, but we could output any number $m = O(1/\alpha)$ of them

within the same space and time: if $A = m + \lfloor 1/\alpha \rfloor$, then there are m minorities among any A distinct elements in $S[i..j]$. By using exactly the same described data structure with this new value of A we can output m α-minorities, or all of them if there are fewer. We could also use larger values of m, but then the query and update times become $O(\frac{m \lg n}{\lg \lg n})$ in the dynamic case and the query time becomes $O(m \lg \lg_w \sigma)$ in the static case.

The alternative (static) solutions for range α-minority queries [8, 3] can be extended similarly. They use a data structure that allows listing all the distinct elements in a range in real time. If they list $m + \lfloor 1/\alpha \rfloor$ distinct elements instead of $1 + \lfloor 1/\alpha \rfloor$, they also obtain m α-minorities. Note that, since they take time $O(m + 1/\alpha)$, they are worst-case optimal for listing $m = \Omega(1/\alpha)$ α-minorities because the output size can be m. Unlike the case of listing one α-minority, whose complexity is unknown, listing $m = \Omega(1/\alpha)$ α-minorities has a natural and reachable lower bound of $\Theta(m)$.

We cannot, instead, find β-minorities for some $\beta \geq \alpha$ in time less than $O(\frac{\lg n}{\alpha \lg \lg n})$. If the range $S[i..j]$ completely contains a piece, then it suffices to test $1 + \lfloor 1/\beta \rfloor$ elements of the contained piece in order to find a β-minority, because all those distinct elements appear in $S[i..j]$. Otherwise, however, the range $S[i..j]$ is contained in one or the concatenation of two pieces, and we do not know which of the $O(A) = O(1/\alpha)$ distinct elements stored for each piece appears in $S[i..j]$. In that case, we may have to test them all until finding a β-minority inside $S[i..j]$.

6 Conclusions and Open Problems

In this article, we have designed the first compressed data structure for dynamic range α-majority. To achieve this result, our key strategy is to perform a multi-level decomposition of the sequence S and, for each block of S, precompute a candidate set that includes all the α-majorities of any query range of the right size that touches this block. Thus, when answering a query, we need not find a set of blocks whose union forms the query range, as is required in the solution of Elmasry *et al.* [11]. Instead, we only look for a single block that touches the query range. This simpler strategy allows us to achieve compressed space.

Furthermore, we generalize our solution to design the first dynamic data structure that can maintain S in the same space and update time, to support the computation of the β-majorities in a given query range for any $\beta \in [\alpha, 1)$ in $O(\frac{\lg n}{\beta \lg \lg n})$ time. Note that here β is given with the queries, and only α is fixed and given beforehand. This type of query is more general than range α-majority queries and was only studied in the static case before [17, 5].

Finally, we design the first dynamic data structure for the range α-minority query problem, and this data structure is also compressed. Even simple static solutions [8] based on range minimum queries are difficult to dynamize. We find a new, simple data structure that is easy to maintain upon updates and gives sufficient information to find α-minorities in time $O(\frac{\lg n}{\alpha \lg \lg n})$. We also extend the solution so as to output up to m α-minorities in time $O((m + 1/\alpha)\frac{\lg n}{\lg \lg n})$; the extension is applicable to the other (static) solutions of this problem [8, 3], and makes them optimal when listing

$m = \Omega(1/\alpha)$ α-minorities. Solving β-minorities in time proportional to $1/\beta$, for any $\beta \geq \alpha$ given at query time, is an open problem.

For constant α, our query time $O(\frac{\lg n}{\lg \lg n})$ for α-majority is optimal within polylogarithmic update time, as it matches the lower bound of the simpler operation *majority* [22, Prop. 3], which considers the particular case of binary alphabets, ranges of the form $S[1..i]$, and $\alpha = 1/2$. Another obvious lower bound is $\Omega(1/\alpha)$, as it is the output size in the worst case. Combining both lower bounds, we obtain $\Omega(\frac{\lg n}{\lg \lg n} + \frac{1}{\alpha})$ query time. It is not clear whether a dynamic α-majority data structure can reach that bound within polylogarithmic update time, even without compression.

Elmasry *et al.* [11, Lem. 1] give a finer lower bound for dynamic α-majorities that includes α in the formula. For polylogarithmic update time and computer words with a polylogarithmic number of bits, the bound is $\Omega(\frac{\lg(\min(\alpha,1-\alpha)n)}{\lg \lg n})$, reaching the maximum with constant α.

Another interesting question is whether the bounds change when allowing only insertions or only deletions for the updates. These operations have been considered together since the beginning (e.g., proving lower bounds on bit flips and then reducing flips to insertion/deletion pairs [15, 22]). Situations where only insertions occur, for example, are common in practice, and thus this question is relevant.

No lower bounds are known for listing m α-minorities, other than the trivial $\Omega(m)$. As explained, we have shown how existing static data structures can be extended to work in time $O(1/\alpha + m)$, which makes them optimal for $m = \Omega(1/\alpha)$, but the gap for $m = o(1/\alpha)$ remains open.

Finally, we show that static versions of our data structures for α-majority and α-minority also use $nH_k + o(n \lg \sigma)$ bits of space ($+2n$ bits in the case of minorities) and answer queries in time $O((1/\alpha) \lg \lg_w \sigma)$. The best static solutions answer queries in time $O(1/\alpha)$ (which is optimal for queries that may return $\Theta(1/\alpha)$ elements), but either use more than $nH_0 \geq nH_k$ bits of space [5] or are compressed only for $\lg(1/\alpha) = o(\lg \sigma)$ [18].

No satic structure using $nH_k + o(n \lg \sigma)$ bits answering range α-majority or α-minority queries in $O(1/\alpha)$ time exists. Further, one may aim at reaching output-sensitive time for those problems, that is, $O(m)$ if they output m results. This seems unlikely, however: Gawrychowski and Nicholson [19] show that solving α-majorities in output-sensitive time would disprove the set intersection conjecture [29].

Acknowledgements

We thank the reviewers for their useful remarks.

References

1. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM Journal on Computing 32(6), 1488–1508 (2003)
2. Beame, P., Jayram, T.S., Rudra, A.: Lower bounds for randomized read/write stream algorithms. In: Proc. 39th Annual ACM Symposium on Theory of Computing (STOC). pp. 689–698 (2007)
3. Belazzougui, D., Gagie, T., Navarro, G.: Better space bounds for parameterized range majority and minority. In: Proc. 12th Annual Workshop on Algorithms and Data Structures (WADS). pp. 121–132 (2013)

4. Belazzougui, D., Navarro, G.: Optimal lower and upper bounds for representing sequences. ACM Transactions on Algorithms 11(4), article 31 (2015)
5. Belazzougui, D., Gagie, T., Munro, J.I., Navarro, G., Nekrich, Y.: Range majorities and minorities in arrays. CoRR abs/1606.04495 (2016)
6. Bell, T.C., Cleary, J., Witten, I.H.: Text Compression. Prentice Hall (1990)
7. Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. Theory of Computing Systems 55(4), 719–741 (2014)
8. Chan, T.M., Durocher, S., Skala, M., Wilkinson, B.T.: Linear-space data structures for range minority query in arrays. Algorithmica 72(4), 901–913 (2015)
9. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Proc. 10th Annual European Symposium on Algorithms (ESA). pp. 348–360 (2002)
10. Durocher, S., He, M., Munro, J.I., Nicholson, P.K., Skala, M.: Range majority in constant time and linear space. Information and Computation 222, 169–179 (2013)
11. Elmasry, A., He, M., Munro, J.I., Nicholson, P.K.: Dynamic range majority data structures. Theoretical Computer science 647, 59–73 (2016)
12. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.D.: Computing iceberg queries efficiently. In: Proc. 24rd International Conference on Very Large Data Bases (VLDB). pp. 299–310 (1998)
13. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theoretical Computer Science 372(1), 115–121 (2007)
14. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
15. 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)
16. Gagie, T., He, M., Navarro, G.: Compressed dynamic range majority data structures. In: Proc. 27th Data Compression Conference (DCC). pp. 260–269 (2017)
17. Gagie, T., He, M., Munro, J.I., Nicholson, P.K.: Finding frequent elements in compressed 2d arrays and strings. In: Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 295–300 (2011)
18. Gawrychowski, P., Nicholson, P.K.: Optimal query time for encoding range majority. In: Proc. 15th International Symposium on Algorithms and Data Structures (WADS). pp. 409–420 (2017)
19. Gawrychowski, P., Nicholson, P.K.: Optimal query time for encoding range majority. CoRR abs/1704.06149 (2017)
20. Greve, M., Jørgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds and approximations for range mode. In: Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP). pp. 605–616 (2010)
21. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 334–346. LNCS 6393 (2010)
22. Husfeldt, T., Rauhe, T.: New lower bound techniques for dynamic partial sums and related problems. SIAM Journal on Computing 32(3), 736–753 (2003)
23. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Transactions on Database Systems 28, 51–55 (2003)
24. Karpinski, M., Nekrich, Y.: Searching for frequent colors in rectangles. In: Proc. 20th Canadian Conference on Computational Geometry (CCCG). pp. 11–14 (2008)
25. Misra, J., Gries, D.: Finding repeated elements. Science of Computer Programming 2(2), 143–152 (1982)
26. Munro, J.I., Nekrich, Y.: Compressed data structures for dynamic sequences. In: Proc. 23rd Annual European Symposium on Algorithms (ESA). pp. 891–902 (2015)
27. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. ACM Transactions on Algorithms 10(3), article 16 (2014)
28. Navarro, G., Thankachan, S.V.: Optimal encodings for range majority queries. Algorithmica 74(3), 1082–1098 (2016)
29. Patrascu, M., Roditty, L.: Distance oracles beyond the Thorup-Zwick bound. SIAM Journal on Computing 43(1), 300–311 (2014)
30. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Proc. 7th International Workshop on Algorithms and Data Structures (WADS). pp. 426–437 (2001)