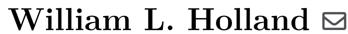
Succinct List Indexing in Optimal Time



William L. Holland

School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Abstract

An *indexed* list supports (efficient) access to both the offsets and the items of an arbitrarily ordered set under the effect of *insertions* and *deletions*. Existing solutions are engaged in a space-time trade-off. On the one hand, time efficient solutions are composed as a package of data structures: a linked-list, a hash table and a tree-type structure to support indexing. This arrangement observes a memory commitment that is outside the information theoretic lower bound (for ordered sets) by a factor of 12. On the other hand, the memory lower bound can be satisfied, up to an additive lower order term, trivially with an array. However, operations incur time costs proportional to the length of the array.

We revisit the list indexing problem by attempting to balance the competing demands of space and time efficiency. We prepare the first *succinct* indexed list that supports efficient query and update operations. To implement an ordered set of size n, drawn from the universe $\{1, \ldots, m\}$, the solution occupies $n(\log m + o(\log n))$ bits (with high probability) and admits all operations *optimally* in $\mathcal{O}(\log n / \log\log n)$ time.

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

Keywords and phrases Succinct Data Structures, Lists, Dynamic Data Structures

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.65

Funding This work was supported by an Australian Government Research Training Program RTP Scholarship.

Acknowledgements We acknowledge the Wurundjeri People of the Kulin Nations as traditional owners of the land on which we live and work.

1 Introduction

List indexing is a popular problem in the data structures literature from the previous century. The problem demands a representation of a list $\mathcal{L}$ that supports the following operations:

- `insert`$(\mathcal{L}, x, y)$: insert element y at the index succeeding element x.
- `delete`$(\mathcal{L}, x)$: delete element x from the list.
- `index`$(\mathcal{L}, i)$: return the element at index i in the list.
- `position`$(\mathcal{L}, x)$: return the index of element x in the list.

A list that supports these operations[1] is named an *indexed list*. Notably, the `position` query is a key inbuilt function in programming languages such as Python and Java. For a list encoding an arbitrarily *ordered* set $\mathcal{L} \subseteq [m]$ of size n, a lower bound of (amortized) $\Omega(\log n / \log\log n)$ time per operation is due to Fredman and Saks [4]. The bound is assembled in the cell probe model. Matching upper bounds, in the word RAM model, are provided by Dietz [3]. As is noted by Andersson [1], the solution requires item pointers be provided as arguments to the operations. Thus, the data structure of Dietz appears in three components; a hash-table to retrieve item pointers; the underlying linked-list; and the structure to support (fast) indexing. This configuration requires $\mathcal{O}(n \log m)$ bits of space with a constant factor around 12.

[1] We assume that the update sequence does not contain any repetitions. We elaborate on the implication of including repetitions in Appendix A.

© William L. Holland;
licensed under Creative Commons License CC-BY 4.0

33rd International Symposium on Algorithms and Computation (ISAAC 2022).
Editors: Sang Won Bae and Heejin Park; Article No. 65; pp. 65:1–65:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A data structure, supporting a particular query (or set of queries), is *succinct* if it accommodates a memory commitment “close to” the information-theoretic lower bound *and* admits the prescribed query operation(s) “efficiently”. Formally, if a minimum of $\mathcal{B}$ bits are required, in the information-theoretic sense, to store the data, a succinct data structure occupies $\mathcal{B} + o(\mathcal{B})$ bits of memory. Despite the affluent and diverse state of the field of succinct data structures [5, 9, 11, 14, 18], a succinct representation of an indexed list does not exist and we pursue the problem of finding such a representation.

The information-theoretic lower bound for encoding an ordered set is

$$B(m,n) = \log\left(\frac{m!}{(m-n)!}\right) = n(\log m - \mathcal{O}(1)) \tag{1}$$

bits. A low-memory solution can be constructed by positioning the items consecutively in an array that has $\log m$ bit cells. The per-item cost of the encoding is $\mathcal{O}(1)$ bits above the information-theoretic lower bound. However, updates to the list and `position` queries are slow, requiring $\mathcal{O}(n)$ time. This observation leads to an interesting question: can a list be encoded at close to $\log m$ bits per item and support dynamic indexing operations efficiently?

In response to this question, we first present a simple data structure, named the Princess List (PL), that significantly improves the update and query efficiency of the canonical linked-list at a small space overhead. We then detail an optimized implementation of the Princess List (named PL+) that occupies $n(\log m + o(\log n))$ bits of space, with high probability, and admits *optimal* query and update times, matching the prior state-of-the-art. The structure pays, per-item, a sublogarithmic number of bits above the information-theoretic lower bound for ordered sets. In the appendix we demonstrate an alternate solution that achieves the memory commitment with probability 1, but with update times close to optimal with high probability.

The Princess List

At a high level, the PL is a divide-and-conquer approach, where the list items are partitioned, via a hash function, into a collection of disjoint sublists. If the hash function that performs the partition maps on to a small range $[\sigma] = \{0, 1, \ldots, \sigma - 1\}$, the list can be mapped to a random *string* on the alphabet $[\sigma]$. Each character $c \in [\sigma]$ represents an order-preserved sublist $\mathcal{L}_c$ on the set $S_c = \{x \in \mathcal{L} \mid h(x) = c\}$ and each occurrence of a character represents a unique item. The PL is comprised of the random string $h(\mathcal{L})$ and the sublists $\mathcal{L}_c$.

An operation entails identifying, through the random string, the correct (and unique) sublist to update or query, followed by an execution of the operation on the small sublist. The random string maintains the inter-order between the sublists and, consequently, allows us to relate each subproblem to the superproblem. Thus, the dynamic string negotiates the divide-and-conquer strategy and, simultaneously, maintains the order of the full list. In other words, the string articulates both how we branch into subproblems and how we merge the sublists back together. With this set-up, queries are fast, comprising a constant number of queries on the string and a linear scan on a subproblem.

The high level structure, composed of the random string and the sublists, introduces three key challenges. First, as each item is represented by both a string character and an item identifier in the sublist, the representation of an item in each component must be concise. Item identifiers are expensive, requiring $\log m$ bits, and we exploit a random permutation of the universe to partition each identifier into both its hash character and a small unique identifier in the assigned sublist. This is a technique known as the quotient filter [17]. Consequently, the total information needed for both the reduced small-alphabet

string and the sublists is concise. Second, we need to maintain all the component data structures in a compact form. To store the small alphabet string we utilize an existing solution for succinct dynamic strings [15]. To reduce the size of the bit allocation for the sublists, we use a linked-list representation with $\omega(1)$ items packed into each node. This allows the per-item cost of the pointers to be sublogarithmic. Third, as we are required to perform linear scans on the sublists, we don't want the cardinality of any sublist to grow too large. To mitigate the impact of a large sublist, we introduce a threshold and store any sublist that exceeds the threshold under the non-succinct but time-optimal solution of Dietz [3].

1.1 Contribution and outline

We present the first succinct indexed list with update time and query time performance identical to the state-of-the-art. We first introduce a simple and novel solution to list indexing (the PL) that lowers the traversal cost on a linked-list (§3). We then curate an optimized instance of our simple solution (the PL+) that achieves the attributes available in the following theorem (§4).

▶ **Theorem 1.** *For any constant $\gamma > 0$, an ordered set of n items, drawn from the universe $[m]$, where $m = \text{poly}(n)$, can be stored in $n(\log m + o(\log n))$ bits with probability $1 - \mathcal{O}(1/n^{\gamma})$ and support* `index` *and* `position` *in $\mathcal{O}(\log n/ \log\log n)$ time and* `insert` *and* `delete` *in $\mathcal{O}(\log n/ \log\log n)$ amortized time.*

This constitutes our main result and contribution. The significance here is that our data structure performs all operations optimal in run time – equal to the prior state-of-the-art – while spending close to $\log m$ bits per item. As a comparison, we achieve better asymptotic performance than balanced binary search trees for updates and access queries with additional support for indexing on ordered sets *and* a succinct representation (over ordered sets) with high probability. In addition, to obtain a succinct representation with probability one, we demonstrate a modified construction (detailed in the appendix) that secures the following properties.

▶ **Theorem 2.** *For any constant $\gamma > 0$, an ordered set of n items, drawn from the universe $[m]$, where $m = \text{poly}(n)$, can be stored in $n(\log m + o(\log n))$ bits and support* `index` *in $\mathcal{O}(\log n/ \log\log n)$ time,* `position` *in $\mathcal{O}(\log n)$ time with probability $1 - \mathcal{O}(1/n^{c})$ and any sequence of $\mathcal{O}(n)$ updates (*`insert` *or* `delete`*) takes $\mathcal{O}(n \log n)$ time with probability $1 - \mathcal{O}(1/n^{\gamma})$.*

One constraint for the data structure is that the universe size is a polynomial in the problem size. Therefore, as currently stated, our result will not hold on small problem instances. However, as small problem sizes are less interesting, this is only a minor concern . In these instances we can afford to store a non-optimal representation and, indeed, this is likely preferable. The constraint is a condition of our hash family [20]. Other succinct data structures, such as Backyard Cuckoo Hashing [2], which utilize the latter result, also inherit this condition.

1.2 The rank-select problem

List indexing has proximity to the well known *rank-select problem* [6, 7]. For the latter, a solution constitutes a succinct representation of a sequence C (where repetitions are allowed), drawn from a universe (or alphabet) Σ, that supports the following operations.

- $C[i]$: return the character at index i.
- $\texttt{rank}_b(C, i)$: given $i \in \{0, \dots, n-1\}$ and $b \in \Sigma$ return $|\{j \in \{0, \dots, i\} \mid C[j] = b\}|$, *i.e.*, the number of occurrences of character b in the subsequence $C[0 \dots i]$.
- $\texttt{select}_b(C, j)$: given $j \in \{0, \dots, n-1\}$ and $b \in \Sigma$ return $\min\{x \mid \texttt{rank}_b(C, x) = j\}$, *i.e.*, the index of the jth occurrence of b in C.

The `select` operation is equivalent to the `position` query and an access $C[i]$ is exactly the `index` query. Thus, a compressed rank-select data structure for dynamic sequences [10, 13, 15, 16] would act as a solution to the list indexing problem. The two problems are very close and, indeed, the rank-select problem inherits the time lower bounds of the list indexing problem. The main difference between the two problems is the assumption about the universe size. For the rank-select problem it is reasonable to focus on texts drawn from a small universe of characters. However, the solutions become intractable for problems on larger universes and, in particular, where repetitions do not occur. For example, with $m = |\Sigma|$, a state-of-the-art solution by Navarro and Nekrich has a memory allocation with a redundancy term of $\mathcal{O}(m \log n)$ [15]. In the list indexing problem, where $m \geq n$, this leads to a non-succinct representation. Further, a solution by Munro and Nekrich [13] that supports "arbitrarily large alphabets" meets a similar fate. While the memory allocation is quoted as $H_k(C) + o(n \log m)$ bits[2], it appears to require (implicitly) that $m \leq n$. The issue is that an auxiliary data structure is stored for each character in Σ. Resolving this issue is not as simple as assigning 0 bits for non-occurring characters as this introduces the need for a search structure. Even a state-of-the-art dynamic succinct dictionary [2] would push the memory allocation over a succinct allowance. A succinct indexed list remedies this restriction on the universe size and is an open problem that we address here.

To help distinguish our problem from this influential strain of prior work, it is best to think of an indexed list as a dictionary. In this context, it would make little sense to use a compressed sequence as a solution to the problem. Before we progress with the exposition, we introduce some background around strings, the hashing scheme we engage and the existing solutions to list indexing

2 Background

We proceed in the (unit cost) word RAM model of computation, with word size $w = \Theta(\log m)$. Consequently, items from the universe can be stored in $\mathcal{O}(1)$ machine words and bitwise operations on words can be performed in constant time. We use the superscript notation $\texttt{index}^M$, $\texttt{position}^M$, $\texttt{insert}^M$, $\texttt{delete}^M$ to refer (unambiguously) to query and update algorithms on a list under the representation of the data structure M. We drop the superscript M and the argument $\mathcal{L}$ when both are obvious from the context.

2.1 Strings

In addition to the `rank` and `select` queries, a dynamic string supports the following update operations:

- $\texttt{insertSt}(C, b, i)$: insert character $b \in \Sigma$ at index i in string C.
- $\texttt{deleteSt}(C, i)$: delete character at index i in string C.

[2] $H_k(S)$ is the kth order empirical entropy of the string S. This term is somewhat redundant in our setting; as repetitions are not allowed, the empirical entropy is high.

The fundamental dynamic string implementation is the wavelet tree. It emerged in the context of text compression [8], as a tool for compressing suffix arrays, and has since seen application in a diverse range of problems. The power of the wavelet tree rests in its capacity to support both the compression of strings and fast query and update operations. The state-of-the-art wavelet tree is advanced by Navarro and Nekrich and acknowledges the following properties.

▶ **Lemma 3** ([15]). *For $\varphi \in (0,1)$, a dynamic string of length n on an alphabet of size σ can be stored in $n \log \sigma + \mathcal{O}(n \log \sigma / \log^{1-\varphi} n) + \mathcal{O}(\sigma \log n)$ bits and support the queries* `rank` *and* `select` *in $\mathcal{O}(\varphi^{-2} \log n / \log\log n)$ time and* `insertSt` *and* `deleteSt` *in $\mathcal{O}(\varphi^{-2} \log n / \log\log n)$ amortized time.*

2.2 *k*-wise independent hashing

The division of the list into disjoint sublists is coordinated by a random source. Our compact representation utilizes existing hash families with *limited* independence, small description and constant time evaluation.

A family of functions is k-wise independent if, for a function $f[m] \to [r]$ selected uniformly at random from the family, the image of any k-tuple, $(f(x_1), \ldots, f(x_k))$, is uniformly distributed in $[r]^k$. There exists no family of k-wise independent hash functions that meets our requirements – that is, small description and constant time evaluation. However, the construction of Siegel [20] is a good approximation and is sufficient for our purposes.

▶ **Theorem 4** ([20]). *Let $S \subseteq U = [m]$ be a set of $n = k^{\mathcal{O}(1)}$ elements. For any constants $\varepsilon, c > 0$ there is a RAM algorithm constructing a random family $\mathcal{H}$ of functions in $o(n)$ time (provided $m = \text{poly}(n)$) and $o(k)$ words of space, such that:*

- *with probability $1 - \mathcal{O}(1/n^c)$, $\mathcal{H}$ is k-wise independent;*
- *there is a RAM data structure of $\mathcal{O}(k^{1+\varepsilon})$ words representing its functions such that a function can be evaluated in constant time. The data structure can be initialized to a random function in $\mathcal{O}(k^{1+\varepsilon})$ time.*

Significantly, the tail probabilities of k-wise independent random variables can be bound with a Chernoff-like result. The result we employ comes from Schmidt *et. al* [19].

▶ **Theorem 5** ([19] Theorem 5.III.b). *If X is the sum of k-wise independent random variables, each confined to the interval $[0,1]$, with $\mu = \text{E}[X]$ and $k = \lfloor \delta\mu e^{-1/3} \rfloor$, then $\Pr[|X - \mu| \geq \delta\mu] \leq e^{-\delta\mu/3}$.*

Negatively related random variables

Furthermore, in our analysis, we encounter instances of sums of indicator random variables that are not k-wise independent, but do hold the property of being *negatively related*. Happily, Jansen established that Chernoff bounds apply to sums of negatively related random variables [12] and we utilize this result in our analysis.

2.3 Prior work

The literature on list indexing assumes that item pointers are readily available (doubtless through a hash table) as arguments to the operations. A non-optimal or naive solution entails a straightforward application of a balanced (or height-bounded) binary tree. List items are stored, in order from left to right, in the leaves of the tree and internal nodes store a count of the number of leaves (or, the size of the sublist) in the subtree rooted at the internal node. An `index`(i) query begins at the root and, following the logic of the counts stored at internal

nodes, can branch into the sublist containing the correct index. Conversely, `position`(x) begins at a leaf and accumulates a global index on a leaf-to-root path; the count of any left sibling at an internal node gets aggregated to the current index. Updates require recourse to a balancing criteria and may entail some restructuring. All operations take $\mathcal{O}(\log n)$ time (worst-case or amortized depending on the choice of tree) and the tree requires $\mathcal{O}(n \log m)$ bits to store.

This idea was extended by Dietz with what are now fairly standard techniques [3]. The binary tree is replaced with a weight balanced B-tree with branching factor $\Theta(\log^{\varphi} m)$, for $\varphi \in (0, 1)$. As before, internal nodes count the number of leaves in the subtree they root. On a leaf-to-root path, a `position` query accumulates the counts, at each internal node, of all *left* siblings. To remove dependence on the branching factor, the sum of the counts of left siblings is evaluated on a *partial sums* data structure. The latter stores an array of integers $A[1 \dots b]$ and admits the following procedures.

- `add`(i, δ): perform $A[i] \leftarrow A[i] + \delta$, where $\delta = \log^{\mathcal{O}(1)} m$.
- `sum`(j): return $\sum_{i \leq j} A[i]$.

On problem size $b = \mathcal{O}(\log^{\varphi} m)$, both operations can be performed in $\mathcal{O}(1)$ amortized time. Thus, the cost of a `position` query is bound by the height $(\mathcal{O}(\log_{\log^{\varphi} m} n))$ of the tree. Efficient navigation for `index`(i) queries is provided by the additional operation:

- `select_ps`(i): return the smallest j such that `sum`$(j) \geq i$.

Although the partial sums structure of Dietz does not explicitly solve the abstraction of `select` queries of prefix sums[3], a constant time solution, on small problem sizes, is provided by Raman *et al.* [18]. Thus, the cost of both query operations is bound by the height of the tree and is thereby optimal at $\mathcal{O}(\log n / \log \log n)$. Due to rebuilding requirements on both the partial sums structure and the weight balanced B-tree, update costs are amortized. The solution requires $\mathcal{O}(n \log m)$ bits.

▶ **Lemma 6** ([3]). *The list indexing problem can be solved by a data structure, occupying $\mathcal{O}(n \log m)$ bits, that supports* `index` *and* `position` *in $\mathcal{O}(\log n / \log \log n)$ time and* `insert` *and* `delete` *in $\mathcal{O}(\log n / \log \log n)$ amortized time.*

Andersson and Petersson define an approximate version of the problem [1]. The `position` query is permitted to err with a user defined relative error and the `index` query can retrieve *any* item from a neighborhood, of size proportional to the relative error, surrounding the correct index. The added flexibility allows the authors to remove the dependence on the problem size from the update and query costs. For relative error parameter $\varepsilon \in (0, 1)$, queries can be evaluated in constant worst-case time and updates in amortized $\mathcal{O}(\varepsilon^{-2})$.

3 The Princess List: a simple solution to list indexing

The linked-list, obliged to traverse all preceding nodes in the ordered set, performs the operation `index`(i) in $\mathcal{O}(i)$ time. To reduce this expense, the PL employs a divide-and-conquer strategy that partitions the list into a collection of disjoint sublists. A search proceeds on a single sublist and thereby reduces the expense of the linear scan. To allocate items to sublists, we map the list, via a hash function h, onto a small alphabet $\Sigma = [\sigma]$ and represent the mapped list

$$h(\mathcal{L}) \cong \langle h(\mathcal{L}[0]), \dots, h(\mathcal{L}[|\mathcal{L}| - 1]) \rangle$$

---

[3] They augment the weight balanced B-tree with additional pointers that implicitly implement the functionality.

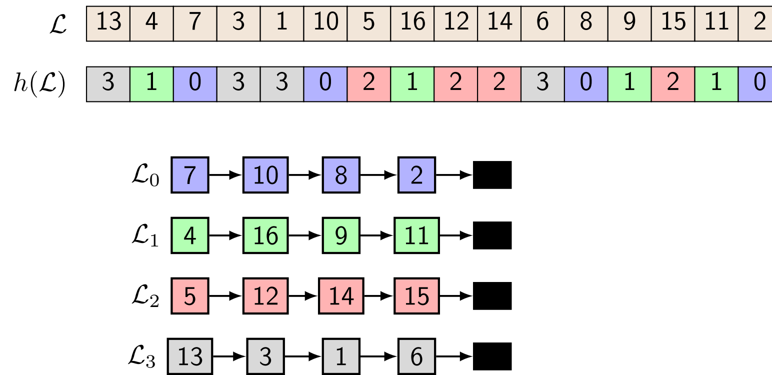


Figure 1 The Princess List. The list $\mathcal{L}$ of items from $U \cong \{1, \dots, 16\}$ is hashed, via the function $h : U \to \{0, \dots, 3\}$, into the string $h(\mathcal{L})$. Items mapping to character c are stored in the (linear) sublist $\mathcal{L}_c$. To evaluate $\texttt{index}(9)$: identify $h(\mathcal{L})[9] = 2$; calculate $\texttt{rank}_2(h(\mathcal{L}), 9) = 2$; and perform $\texttt{index}(\mathcal{L}_2, 2) = 14$ with a linear scan.

with a dynamic string. Under this regime, multiple items may map to the same character and we store such a set of items under a (order-preserved) list representation. The random string $h(\mathcal{L})$ organises the branching into subproblems and, simultaneously, preserves the order of the full list. The range of the hash function permits control over the size of the sublists. An image of the PL arrangement, which outlines, at a high level, the division of the problem into subproblems and the support of the string $h(\mathcal{L})$ towards managing the relation between subproblems, is available in Figure 1.

Fixing notation, let M denote the list data structure used to implement the sublists $\mathcal{L}_\Sigma$ and $\mathsf{PL}^M(\mathcal{L}, \sigma)$ name the subsequent Princess List representation. To evaluate $\texttt{index}(i)$, the program begins by identifying the relevant sublist through the operation $c = h(\mathcal{L})[i]$. Therefore, the item at location i in $\mathcal{L}$ belongs to sublist $\mathcal{L}_c$. Subsequently, the global location i needs to be translated to a local location on the sublist. This is achieved by a $\texttt{rank}$ query; $i' = \texttt{rank}_c(h(\mathcal{L}), i) - 1$ denotes the position of $\mathcal{L}[i]$ in the sublist $\mathcal{L}_c$. The query is completed by the operation $\texttt{index}^M(\mathcal{L}_c, i')$, which, under a linked-list representation, stipulates a linear scan and requires $\mathcal{O}(i')$ time. Conversely, to evaluate $\texttt{position}(x)$, the program starts at the sublist $\mathcal{L}_{h(x)}$. The value $j = \texttt{position}^M(\mathcal{L}_{h(x)}, x) + 1$ provides the rank of the character occurrence $h(x)$ that corresponds to the item x. The query is completed by determining the global index of the jth occurrence of $h(x)$ in $h(\mathcal{L})$ through the query $\texttt{position}(x) = \texttt{select}_{h(x)}(h(\mathcal{L}), j)$. Pseudo-code for the query operations is available in Figure 2.

The update $\texttt{insert}(x, y)$ begins by identifying the index of insertion in both the string $i = \texttt{position}^{\mathsf{PL}}(\mathcal{L}, x)$ and the relevant subproblem $i' = \texttt{rank}_{h(y)}(h(\mathcal{L}), i)$. Character $h(y)$ is then inserted at index $i + 1$ in the dynamic string and item y is placed after item $\texttt{position}^M(\mathcal{L}_{h(y)}, i')$ in sublist $\mathcal{L}_{h(y)}$. The $\texttt{delete}(x)$ operation proceeds in a similar fashion; it locates the index of the placeholder of x in $h(\mathcal{L})$ and identifies the relevant subproblem $c = h(x)$. The character at this index is then removed from $h(\mathcal{L})$ and x is removed from $\mathcal{L}_c$. Pseudo-code for the update operations is available in Figure 3.

The efficiencies of the operations depend on the efficiency of the dynamic string $h(\mathcal{L})$ and the size of the sublists. By Lemma 3, all string queries can be supported in $\mathcal{O}(\log n / \log\log n)$ time and string updates in $\mathcal{O}(\log n / \log\log n)$ amortized time. Each sublist has expected size n/σ. The amount by which sublist sizes deviate from their expectation depends on both σ and the randomness of the underlying hashing scheme.

```
Procedure index^PL(i)
    c ← h(L)[i]
    i' ← rank_c(h(L), i) − 1
    return index^M(L_c, i')
```

```
Procedure position^PL(x)
    c ← h(x)
    j ← position^M(L_c, x) + 1
    return select_c(h(L), j)
```

Figure 2 Query operations for the $\mathsf{PL}^M(\mathcal{L}, \sigma)$ representation of the list $\mathcal{L}$. The data structure M is used to implement the sublists.

```
Procedure delete^PL(x)
    i ← position^PL(x)
    c ← h(x)
    delete^M(L_c, x)
    deleteSt(h(L), i)
    return
```

```
Procedure insert^PL(x, y)
    i ← position^PL(x)
    c ← h(y)
    i' ← rank_c(h(L), i)
    z ← position^M(L_c, i')
    insert^M(L_c, z, y)
    insertSt(h(L), c, i + 1)
    return
```

Figure 3 Update operations for the $\mathsf{PL}^M(\mathcal{L}, \sigma)$ representation of the list $\mathcal{L}$.

3.1 Towards a compact representation

In itself, the PL does not articulate a succinct form. Thus, before moving to the main result, we need to construct the bridge between the general form and an optimal encoding. This is achieved in three parts. First, we need to paramteterize the PL by choosing an alphabet size. The verdict on the parameter choice is provided by the $\Omega(\log n / \log\log n)$ time lower bound for the operations of an indexed list [4]. We require *linear* scans when operating on sublists. Thus, it is desirable that subproblems have size $\mathcal{O}(\log n / \log\log n)$. As sublists have expected cardinality n/σ, the latter implies an alphabet size of

$$\sigma = \Omega\left(n \cdot \frac{\log\log n}{\log n}\right). \tag{2}$$

Second, when following a standard linked-list implementation, the memory allocation of the subproblems trespasses over $n(\log m + o(\log n))$ bits. Therefore, an alternate list representation is required at the sublists. Multiple directions are available and we opt for packing a super-constant number of items in the nodes of a linked-list to reduce the per-item cost of the pointers. Third, in light of the space allocated to the dynamic string – which is $\omega(1)$ bits per character by equation (2) – we cannot afford to store the full key of each item. The key length can be reduced with a technique, standard in succinct dictionaries [17], named the quotient filter. A random permutation $\pi : [m] \to [m]$ is employed. For item x, the leftmost $\log \sigma$ bits of $\pi(x)$ specify the subproblem to which x is a member and the rightmost $(\log m - \log \sigma)$ bits of $\pi(x)$ are stored in the list. In this manner, the representation of the item is split between a character in the dynamic string *and* an identifier in the allocated sublist. Permutations are necessary to avoid collisions in the sublists. The remainder of the paper illustrates the details of the bridge outlined above. We refer to this construction, an optimal instance of the PL, as PL+.

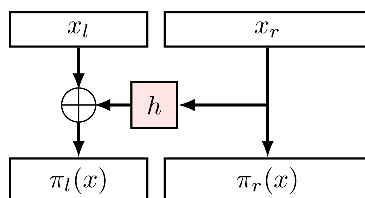


Figure 4 One-round Feistal permutation [2].

4 PL+: state-of-the-art wavelet trees and packed linked-lists

Items are allocated to sublists with Siegel's hash family [20]. Therefore, by Theorem 4, with probability $1 - \mathcal{O}(1/n^\gamma)$, item allocations are k-wise independent, for $k^{\mathcal{O}(1)} = n$ and any constant $\gamma > 0$. Further, the hash function is evaluated in constant time and is stored in $o(n)$ bits. For our construction, we require that $\log n = \Theta(\log m) = \Theta(w)$. In other words, it is assumed that the universe size is a polynomial in the problem size.

Our structure, PL+, is an instantiation of the $\mathsf{PL}^{\mathsf{PackUnpack}}(\mathcal{L}, \Theta(n \log\log n/\log n))$ form, where PackUnpack refers to a type of linked-list that we detail below. To implement the dynamic string, PL+ appoints the wavelet tree of Navarro and Nekrich [15]. As the ideal alphabet size changes with n, the structure is periodically rebuilt to conform with the configuration of asymptotic constaints of the alphabet. We set σ to a power-of-two (for efficiency) such that

$$\sigma \in \left[\frac{1}{2}n \cdot \frac{\log\log n}{\log n}, 2n \cdot \frac{\log\log n}{\log n}\right] \tag{3}$$

Therefore, a rebuild happens, in the worst-case, every $\Omega(n)$ updates. A simple rebuild can be performed by constructing a new string under the updated alphabet. This takes $\mathcal{O}(n \log n/\log\log n)$ time by Lemma 3. In addition, selecting a new hash function from the constructed hash family takes $o(n)$ time. Periodically, but not with every rebuild, we will also have to reconstruct the hash family. The latter occurrence takes $o(n)$ time. Thus, rebuilding contributes $\mathcal{O}(\log n/\log\log n)$ amortized time to each update.

4.1 A random permutation

Without loss of generality, and for ease of demonstration, we assume that m is a power-of-two. The permutation is generated from a collection of one-round Feistal permutations. We use them in a manner similar to Arbitman *et al.* [2], where existing hash functions, and the properties they inhabit, are recruited to generate the randomness. Let $\mathcal{H}_\sigma$ be a class of $h : [m/\sigma] \to [\sigma]$ functions. Let x_l denote the leftmost $\log\sigma$ bits of a key x and x_r denote the rightmost $\log(m/\sigma)$ bits. A permutation $\pi_h : [m] \to [m]$ is defined for each instance $h \in \mathcal{H}_\sigma$:

$$\pi_h(x) = (\pi_l(x), \pi_r(x)) = (x_l \oplus h(x_r), x_r).$$

This is a permutation as every pair of keys x and y, where $x_r = y_r$, receive the same hash value, but map to a different bucket under the $\oplus$ operation. An image of the permutation is provided in Figure 4. With the permutation π, an item x is allocated to a subproblem with the random character $c = x_l \oplus h(x_r)$ and is stored under the representation x_r in $\mathcal{L}_c$. In this manner, items are allocated to subproblems according to the randomness prescribed by our hashing scheme.

4.2 The sublists

From Lemma 3 (with the parameter φ set to some constant), the dynamic string $h(\mathcal{L})$ occupies

$$n \log \sigma + o(n \log n) + \mathcal{O}(\sigma \log n) = n(\log \sigma + o(\log n)) \tag{4}$$

bits. Thus, for a list implementation that is asymptotically close to $n \log m$ bits, the remaining budget available to the sublists is $n(\log m - \log \sigma + o(\log n))$. As stated above, an item is stored in sublist $\mathcal{L}_{\pi_l(x)}$ with the $(\log m - \log \sigma)$ (permuted) bits of $\pi_r(x)$. This representation is in line with the budget stated above. It remains to demonstrate how the collection of sublists can be stored efficiently, with low redundancy.

A sublist is stored in a *packed* linked-list. A node in the linked-list contains at most $\log \log n$ items stored in a packed array. If the keys are represented in K bits, the contents of each node occupy $K \cdot \log \log n$ bits. Only the last node in the list is permitted to have less than $\log \log n$ items. Consequently, the nodes are packed as tight as possible. Updates and queries can be performed on linear traversals. Note that the update operations require the shifting of items prior to insertion and after deletion.

As we perform linear scans (in the tradition of the linked-list), the runtime of operations depends on the cardinality of the sublists. Thus, for optimality, sublist cardinalities cannot exceed $\mathcal{O}(\log n / \log \log n)$. Unfortunately, in all likelihood, some sublists will contain $\omega(\log n / \log \log n)$ items. To mitigate this outcome, we track the cardinalities of each sublist and, for a large enough constant $C > 0$, if a sublist cardinality exceeds $C \log n / \log \log n$, we implement the sublist with an *uncompressed* solution. The latter could be the time optimal solution put forth by Dietz [3] (see Lemma 6). With this set-up all operations require $\mathcal{O}(\log n / \log \log n)$ time to complete. We refer to this list data structure, which resorts to an uncompressed solution when its cardinality exceeds a specified threshold $C \cdot \log n / \log \log n$, as a pack-unpack list with threshold parameter C ($\mathsf{PackUnpack}(C)$).

4.3 Memory allocation for sublists

The proofs for the remainder of this section are located in Appendix B.2. The key with PL+ is to ensure that the number of items that belong to uncompressed sublists does not grow too large. To provide an upper bound on the latter, we compute both the maximum cardinality of the sublists and bound the number of sublists with cardinalities that exceed our threshold. We begin with the former. The upcoming pair of lemmas rely on the properties of k-wise independence. Recall that the construction of the hash family fails with probability $\mathcal{O}(1/n^\gamma)$, for any constant $\gamma > 0$, and the analysis of the probabilities must account for this occurrence. We use the notation $\xi_\gamma = \mathcal{O}(1/n^\gamma)$ to refer to this error term incurred by our choice of hash family.

▶ **Lemma 7.** *For $\gamma > 0$,*

$$\max_{c \in [\sigma]} |\mathcal{L}_c| \leq C_1 \log n \tag{5}$$

with probability at least $1 - e^{-\mathcal{O}(C_1) \log n} - \xi_\gamma$.

The proof is a standard rehearsal of bounding the maximum load in a balls-in-bins problem with a Chernoff bound. We now bound the number of sublists that exceed the threshold.

▶ **Lemma 8.** *For $C_1 = \mathcal{O}(\log n)$, C set to a sufficiently large constant and any $\gamma > 0$,*

$$|\{c \mid |\mathcal{L}_c| \geq C \cdot \log n / \log\log n\}| \leq \frac{n}{C_1 \log n \cdot \log\log n} \tag{6}$$

with probability at least $1 - e^{-n/(\mathcal{O}(C_1)\log n \log\log n)} - \xi_\gamma$.

Let Q_C be the set of items that belong to unpacked lists when using threshold parameter C. The cardinality $|Q_C|$ is less than the product of the max load and the number of sublist cardinalities that exceed the threshold. Therefore, with the combination of Lemmas 7 and 8, and an appropriate choice of C_1, we arrive at the following result.

▶ **Lemma 9.** *For a sufficiently large constant C and $\gamma > 0$, $|Q_C| \leq n/\log\log n$ with probability at least $1 - \xi_\gamma$.*

Each item in an unpacked list requires $\mathcal{O}(\log n)$ bits to store. Therefore, this result implies that, with high probability, items belonging to unpacked lists occupy a total of $\mathcal{O}(n \log n/\log\log n)$ bits. This allocation is dominated by the space occupied by the packed lists and leads to the following result.

▶ **Lemma 10.** *For any $\gamma > 0$, if each item occupies K bits, the memory allocation of the sublists, with sufficiently large threshold parameter C, aggregates to $n(K + o(\log n)) + o(n)K$ bits with probability at least $1 - \mathcal{O}(1/n^\gamma)$.*

5 Space and time efficiency of PL+

The memory allocation for PL+ is determined by the implementations of the string $h(\mathcal{L})$ and the sublists $\{\mathcal{L}_c\}$. Therefore, with the combination of Lemma 3 and Lemma 10, we arrive at a bound on the memory allocation of PL+.

▶ **Lemma 11.** *For any $\gamma > 0$, for $m = \text{poly}(n)$, to store a list of n items, PL+ occupies $n(\log m + o(\log n))$ bits of space with probability at least $1 - \mathcal{O}(1/n^\gamma)$.*

As the alphabet size is a function of the number of items, PL+ needs to be rebuilt periodically. By relation (3), this occurs every $\Omega(n)$ updates. The change in alphabet also affects the permutation and we periodically require a *different* initialization of the hash family to support the scheme. As discussed above, the string can be rebuilt in $\mathcal{O}(n \log n/\log\log n)$ time. For a crude bound – which assumes that all items are uncompressed – the sublists can be rebuilt in $\mathcal{O}(n \log n/\log\log n)$ time. Therefore, the overall cost of the rebuild is $\mathcal{O}(\log n/\log\log n)$ amortized time per update.

▶ **Lemma 12.** *On a list of n items, PL+ performs* `index` *and* `position` *in $\mathcal{O}(\log n/\log\log n)$ time and updates* (`insert` *or* `delete`) *complete in $\mathcal{O}(\log n/\log\log n)$ amortized time.*

Combined, Lemmas 11 and 12 curate our main result, which we restate below.

▶ **Theorem 1.** *For any constant $\gamma > 0$, an ordered set of n items, drawn from the universe $[m]$, where $m = \text{poly}(n)$, can be stored in $n(\log m + o(\log n))$ bits with probability $1 - \mathcal{O}(1/n^\gamma)$ and support* `index` *and* `position` *in $\mathcal{O}(\log n/\log\log n)$ time and* `insert` *and* `delete` *in $\mathcal{O}(\log n/\log\log n)$ amortized time.*

The error term in Theorem 1 is dominated by the likelihood that the hash family fails. On the condition that the construction of the random hash family is successful, the desired bit allocation holds with probability $1 - (1/n)^{\omega(1)}$. The fail rate applies to each update. Therefore our result holds, with high probability, on any sequence of operations that is polynomial in length. Notably, the structure incurs, per-item, a sublogarithmic number of bits above the information theoretic bound and, taken with the efficient query and update operations, constitutes a *succinct* representation: by equation (1),

$$\begin{aligned} n(\log m + o(\log n)) &= B(m,n) + \mathcal{O}(n) + n \cdot o(\log n) \\ &= (1 + o(1))B(m,n). \end{aligned}$$

The significance of the result is that the runtime of all operations is optimal, matching the previous best, and we reduce the memory commitment to a succinct representation with high probability.

To remove the probabilistic expression surrounding the memory allocation, we can just commit to using packed linked-lists for *all* sublists irrespective of their cardinalities. The outcome of this set-up is described in Theorem 2 and we provide the necessary detail and analysis in Appendix C.

6 Conclusion

The PL is an uncomplicated approach to list indexing. The algorithms are simple and offer immediate improvements over solutions that employ linear traversals. The data structure admits an instance that is close to optimal in space with respect to ordered sets. This is the first succinct indexed list with optimal time query and update operations. There are two directions for further work. In all prior work, update operations are amortized. Thus, even in the non-succinct case, designing worst-case update operations is an open problem. Further, achieving a non-probabilistic memory allocation with optimal time operations would conclude this line of work.

References

1 Arne Andersson and Ola Petersson. Approximate indexed lists. *Journal of Algorithms*, 29(2):256–276, 1998.
2 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *FOCS*, pages 787–796. IEEE, 2010.
3 Paul F Dietz. Optimal algorithms for list indexing and subset rank. In *Workshop on Algorithms and Data Structures*, pages 39–46. Springer, 1989.
4 Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *STOC*, pages 345–354, 1989.
5 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, pages 326–337. Springer, 2014.
6 Alexander Golynski, J Ian Munro, and S Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, volume 6, pages 368–373, 2006.
7 Rodrigo González and Gonzalo Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410(43):4414–4422, 2009.
8 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.
9 Torben Hagerup. Highly succinct dynamic data structures. In *International Symposium on Fundamentals of Computation Theory*, pages 29–45. Springer, 2019.

10 Meng He and J Ian Munro. Succinct representations of dynamic strings. In *International Symposium on String Processing and Information Retrieval*, pages 334–346. Springer, 2010.

11 William L Holland, Anthony Wirth, and Justin Zobel. Recency queries with succinct representation. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, 2020.

12 Svante Janson. Large deviation inequalities for sums of indicator variables. *arXiv preprint*, 2016. `arXiv:1609.00533`.

13 J Ian Munro and Yakov Nekrich. Compressed data structures for dynamic sequences. In *Algorithms-ESA 2015*, pages 891–902. Springer, 2015.

14 J Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation (ISAAC 2018)*, 2018.

15 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In *SODA*, pages 865–876. SIAM, 2013.

16 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):1–39, 2014.

17 Rasmus Pagh. Low redundancy in static dictionaries with o (1) worst case lookup time. In *ICALP*, pages 595–604. Springer, 1999.

18 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *WADS*, pages 426–437. Springer, 2001.

19 Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

20 Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM Journal on Computing*, 33(3):505–543, 2004.

A Allowing for Repetitions

If we allow for repetitions, there are some precedents (from Python or Java) on how to modify the behavior of the operations. For example, the `position` function could return the lowest index of an item occurrence. The question is; how does this affect the theoretical performance?

Repetitions reduce the entropy of the list and the size of the lower bound. Certainly there are mechanisms that can leverage the reduction in entropy in the sublists (i.e. we can apply compression techniques). However, there is a point (in terms of the number of repetitions) at which this becomes a dynamic sequences problem and we would shift towards using the dynamic string of Munro and Nekrich [13].

The key difference between our version of the problem and that of the original propblem proposed by Dietz [3], is that the updates and the `position` query take pointers (or “records”) as inputs. It would make less sense to have repeated records in the list if we access them via pointers. Thus it is sensible to think of these records – each associated with a key in the list – as distinct from each other. Given that the index query returns a key but not a record, we think, even under the Dietz version of the problem, it is natural to think of the keys as distinct.

Further, we think our formalization of indexing an *ordered set* is what makes this an interesting problem.

B Relegated Proofs

B.1 Memory Allocation for Sublists

▶ **Lemma 7.** *For* $\gamma > 0$*,*

$$\max_{c\in[\sigma]} |\mathcal{L}_c| \leq C_1 \log n \tag{5}$$

with probability at least $1 - e^{-\mathcal{O}(C_1)\log n} - \xi_\gamma$*.*

Proof. Fix a sublist c. Let X_i be an indicator for the event that the ith item added to $\mathcal{L}$ is placed in sublist c. It holds that $\mathrm{E}[X_i] = 1/\sigma$. The cardinality $X = \sum_{i=1}^{n} X_i$ of sublist c is the sum of k-wise independent random variables (for $k = n^\alpha, \alpha \in (0,1)$). Therefore, by Theorem 5, with $\mu = \mathrm{E}[X] \in [\log n/(2\log\log n), 2\log n/\log\log n]$ by Equation (3),

$$\begin{aligned}\mathsf{Pr}[X \geq C_1 \log n] &\leq \mathsf{Pr}[X \geq C_1 \log\log n/2 \cdot \mu] \\ &\leq \mathsf{Pr}[|X - \mu| \geq (C_1 \log\log n/2 - 1)\cdot \mu] \\ &\leq e^{-\mathcal{O}(C_1)\cdot \log n},\end{aligned}$$

by our choice of k and Theorem 4. The third line holds on the condition that $C_1 \geq 3\log\log n$. A union bound absorbs the probability that the construction of the hash family fails and we arrive at the stated result. ◀

▶ **Lemma 8.** *For* $C_1 = \mathcal{O}(\log n)$*,* C *set to a sufficiently large constant and any* $\gamma > 0$*,*

$$|\{c \mid |\mathcal{L}_c| \geq C \cdot \log n/\log\log n\}| \leq \frac{n}{C_1 \log n \cdot \log\log n} \tag{6}$$

with probability at least $1 - e^{-n/(\mathcal{O}(C_1)\log n \log\log n)} - \xi_\gamma$*.*

Before we proceed with the proof, we need an additional result. Let $X_{i,c}$ indicate that item i is allocated to sublist c and $X_c = \sum_{i=1}^{n} X_{i,c}$ denote the sublist cardinalities for $c \in [\sigma]$. Further, we define

$$Y_c = \begin{cases} 1 & \text{if } X_c > C \cdot \log n/\log\log n \\ 0 & \text{otherwise} \end{cases}$$

to indicate the event that sublist c exceeds the threshold. The task is to bound the sum $\sum_{c=1}^{\sigma} Y_c$. However, the random variables $\{Y_c\}$ are not independent. Fortunately, the $\{Y_c\}$ are *negatively related* and a Chernoff bound can be applied to acquire a concentration result on the sum of the indicators [12]. Negatively related random variables are defined as follows.

▶ **Definition 13** ([12])**.** *The indicator random variables* $\{Y_i\}_{i\in[\sigma]}$ *(defined on some probability space) are negatively related if for each* $j \leq \sigma$ *there exists further random variables* $\{J_{i,j}\}_{i\in[\sigma]}$ *defined on the same probability space such that:*

- *the definition of the random vector* $\{J_{i,j}\}_{i\in[\sigma]}$ *equals the conditional distribution of* $\{Y_i\}_{i\in[\sigma]}$ *given* $Y_j = 1$*;*
- $J_{i,j} \leq Y_i, \forall i \neq j$*.*

▶ **Lemma 14.** *The random variables* $\{Y_c\}_{c\in[\sigma]}$ *are negatively related.*

Proof. We can imagine this as an experiment where n balls are thrown into σ bins where the bin locations for the balls are n^α-wise independent for $\alpha \in (0,1)$. Let X_c denote the load of bin c and let $L = C \cdot \log n/\log\log n$. The indicator Y_c is 1 if $X_c > L$. To demonstrate that the indicators $\{Y_c\}$ are negatively related we construct the distributions $\{J_{c,j}\}_{c\in[\sigma]}$ for $j \in [\sigma]$ that satisfy the two properties of Definition 13. Fix a $j \in [\sigma]$. The random variables $\{J_{c,j}\}$ take the following form. First throw the n balls into σ bins as above. If $X_j > L$, then set $J_{c,j} = Y_c$ for all $c \in [\sigma]$. Otherwise, pick $\lfloor(L - X_j + 1)\rfloor$ balls uniformly at random from bins in $[\sigma] \setminus \{j\}$ and place them in bin j. Let X_c^* denote the modified bin loads. Now, $J_{c,j} = 1$ if $X_c^* > L$. As the bin loads X_c^* are conditioned on $X_j > L$, the indicators have the correct distribution. Further, $J_{c,j} \leq Y_c$ as we only remove items from bins. Thus, the $\{Y_c\}$ are negatively related. ◀

Now that we can apply a Chernoff bound, we can complete the proof of Lemma 8.

Proof of Lemma 8. As X is the sum of k-wise independent random variables, by Theorem 5, our choice of k and the observation that $\mu = \mathrm{E}[X] \leq 2\log n/\log\log n$, the following holds:

$$\begin{aligned}\Pr[Y_c = 1] &= \Pr[X_c > C \cdot \log n/\log\log n] \\ &\leq \Pr[|X_c - \mu| > (C/2 - 1)\cdot\mu] \\ &\leq e^{-\mathcal{O}(C)\mu}.\end{aligned}$$

Let $Y = \sum_{c=1}^{\sigma} Y_c$ denote the number of sublists that exceed the cardinality threshold.

$$\mathrm{E}[Y] \leq e^{-\mathcal{O}(C)\mu}\cdot\sigma \leq e^{-\mathcal{O}(C)\log n/\log\log n}\cdot\frac{n\log\log n}{\log n} \leq \frac{n}{(\log n)^{\mathcal{O}(C)}}. \tag{7}$$

The indicators $\{Y_c\}$ are not independent. However, by Lemma 14, they are *negatively related* and we are permitted to apply a Chernoff bound on their sum. Therefore, for $\delta = n/(C_1 \cdot 2\mathrm{E}[Y]\log n\log\log n)$,

$$\Pr[Y > n/(C_1\cdot\log n\log\log n)] \leq \Pr\big[|Y - \mathrm{E}[Y]| > \delta\mathrm{E}[Y]\big] \leq e^{-n/(\mathcal{O}(C_1)\log n\log\log n)}$$

on the condition that $\delta \geq 1$. The latter occurs if

$$1 \leq \delta = \frac{n}{C_1\cdot 2\mathrm{E}[Y]\log n\log\log n} \leq \frac{(\log n)^{\mathcal{O}(C)}}{2C_1\log\log n}$$

The second inequality comes from Equation (7) and holds for $C_1 = \mathcal{O}(\log n)$ and sufficiently large C. Adding the error rate of the hash family, via the union bound, completes the proof. ◀

▶ **Lemma 9.** *For a sufficiently large constant C and $\gamma > 0$, $|Q_C| \leq n/\log\log n$ with probability at least $1 - \xi_\gamma$.*

Proof. By design, $|Q_C| \leq n/\log\log n$ if inequalities (5) & (6) both hold. By Lemmas 7 and 8, with $C_1 = \Theta(\log n)$, in conjunction with a union bound, both the inequalities hold with probability $1 - (1/n)^{\omega(1)} - \xi_\gamma$. This completes the proof. ◀

▶ **Lemma 10.** *For any $\gamma > 0$, if each item occupies K bits, the memory allocation of the sublists, with sufficiently large threshold parameter C, aggregates to $n(K + o(\log n)) + o(n)K$ bits with probability at least $1 - \mathcal{O}(1/n^\gamma)$.*

Proof. The structure is comprised of at most σ packed linked-lists. These linked-lists contain at most $n/\log\log n + \sigma$ nodes. Each node contains $\log\log n$ items of K bits and two $\mathcal{O}(\log n)$ bit pointers. This accumulates to a bit commitment of size

$$\begin{aligned}
&\left(\frac{n}{\log\log n} + \sigma\right)(K \cdot \log\log n + \mathcal{O}(\log n)) \\
&= nK + n \cdot o(\log n) + \sigma K \cdot \log\log n + \sigma\mathcal{O}(\log n) \\
&= n(K + o(\log n)) + K\Theta\left(\frac{n\log^2\log n}{\log n}\right) + \Theta(n\log\log n) \\
&= n(K + o(\log n)) + o(n)K. \qquad (8)
\end{aligned}$$

By Lemma 9, the structure contains at most $n/\log\log n$ uncompressed items with probability $1 - \xi_\gamma = 1 - \mathcal{O}(1/n^\gamma)$. Each uncompressed item occupies $\mathcal{O}(\log n)$ bits. Thus, combined, uncompressed items occupy

$$n/\log\log n \cdot \mathcal{O}(\log n) = n \cdot o(\log n)$$

bits. This allocation is absorbed by the allocation for compressed items (Equation (8)). ◀

B.2 Performance for PL+

▶ **Lemma 11.** *For any $\gamma > 0$, for $m = \mathrm{poly}(n)$, to store a list of n items, PL+ occupies $n(\log m + o(\log n))$ bits of space with probability at least $1 - \mathcal{O}(1/n^\gamma)$.*

Proof. For the permutation $\pi : [m] \to [m]$, the leftmost $\log\sigma$ bits specify the subproblem and the rightmost $\lceil\log m\rceil - \log\sigma$ bits are stored in the sublist. Thus, the space occupied by the sublists is determined by Lemma 10 with key size $K = \lceil\log m\rceil - \log\sigma$. By equation (4), the cost of the dynamic string is $n(\log\sigma + o(\log n))$ bits. Aggregated with the sublists, we get a memory commitment (measured in bits) of

$$\begin{aligned}
&n(\log\sigma + o(\log n)) + nK + n \cdot o(\log n) + o(n)K \\
&= n(\log\sigma + o(\log n)) + n\log(m/\sigma) + o(n)\log(m/\sigma) \\
&= n(\log m + o(\log n)) + o(n) \\
&= n(\log m + o(\log n)),
\end{aligned}$$

as required. The forth term in line 2 is $o(n)$ on the stated condition $m = \mathrm{poly}(n)$. The failure probability is inherited from Lemma 10. ◀

▶ **Lemma 12.** *On a list of n items, PL+ performs* `index` *and* `position` *in $\mathcal{O}(\log n/\log\log n)$ time and updates (*`insert` *or* `delete`*) complete in $\mathcal{O}(\log n/\log\log n)$ amortized time.*

Proof. By Lemma 3, taking parameter φ as constant, all queries to the random string take $\mathcal{O}(\log n/\log\log n)$ time and update operations require $\mathcal{O}(\log n/\log\log n)$ amortized time. By Lemma 6, all operations on the uncompressed implementation for PackUnpack take $\mathcal{O}(\log n/\log\log n)$ time. The query $\texttt{index}^{\text{PL+}}(i)$ demands two operations on the random string and an $\texttt{index}^{\text{PackUnpack}}$ query on a sublist. The local $\texttt{index}^{\text{PackUnpack}}(\mathcal{L}_c, i')$ requires either a linear scan of length $\mathcal{O}(\log n/\log\log n)$ or an index operation on a fast uncompressed solution. The $\texttt{position}^{\text{PL+}}$ query requires one constant time hash evaluation, a `select` query and $\texttt{position}^{\text{PackUnpack}}$ query on the subproblem. Similar to the case of the `index` query, all subroutines take $\mathcal{O}(\log n/\log\log n)$ time and the claim for `position` holds.

An update is obtained by a combination of an update to the dynamic string and an update to the subproblems. The former operation has an amortized cost of $\mathcal{O}(\log n/\log\log n)$ and the runtime of the latter, similar to the query operations, requires a constant number of (possibly amortized) $\mathcal{O}(\log n/\log\log n)$ time subroutines. Periodic rebuilding adds amortized $o(\log n/\log\log n)$ time to each update. This completes the proof. ◀

C Alternative implementation for PL+

In our main result we achieve a succinct representation with high probability. The structure can be modified such that a succinct representation occurs with probability one: we commit to using packed linked lists (pack) and tolerate sublists with cardinality $\omega(\log n/\log\log n)$. Further, the local `index`$^{\mathsf{Pack}}(\mathcal{L}_c, i)$ query can take advantage of knowing *which* node the offset i belongs to. For every packed linked-list, we store a dynamic array of pointers, which we name *skip* pointers, where the jth pointer points to the jth node in the chain. This permits constant time access to a specified node. Consequently, the `index`$^{\mathsf{Pack}}(\mathcal{L}_c, i)$ performs small $\mathcal{O}(\log\log n)$ traversals on a portion of the sublist $\mathcal{L}_c$. The skip pointers add a negligible $\sigma \cdot \mathcal{O}(\log n)$ bits to the memory allocation of the structure. The runtimes of the other operations depend on the cardinality of the engaged sublist. The latter is at most $\mathcal{O}(\log n)$ with probability $1 - \mathcal{O}(1/n^{\gamma})$. Note that Lemma 10 accounts for σ packed lists. Thus, we arrive at the following result.

▶ **Theorem 2.** *For any constant $\gamma > 0$, an ordered set of n items, drawn from the universe $[m]$, where $m = \text{poly}(n)$, can be stored in $n(\log m + o(\log n))$ bits and support* `index` *in $\mathcal{O}(\log n/\log\log n)$ time,* `position` *in $\mathcal{O}(\log n)$ time with probability $1 - \mathcal{O}(1/n^{c})$ and any sequence of $\mathcal{O}(n)$ updates* (`insert` *or* `delete`) *takes $\mathcal{O}(n \log n)$ time with probability $1 - \mathcal{O}(1/n^{\gamma})$.*

As the expected cardinality of a sublist is $n/\sigma = \mathcal{O}(\log n/\log\log n)$, the *expected* runtime of `position` and the expected amortized runtime of the updates is $\mathcal{O}(\log n/\log\log n)$.