# Pushing the Information-Theoretic Limits of Random Access Lists

Traversing Cons Lists in $\left(1+\frac{1}{\sigma}\right)\lfloor\lg n\rfloor+\sigma+9$ Steps

EDWARD PETERS, Independent, USA
YONG QI FOO, National University of Singapore, Singapore
MICHAEL D. ADAMS, National University of Singapore, Singapore

Accessing an arbitrary element of a singly linked list or *cons* list requires traversing up to a linear number of pointers. The applicative random-access list is a data structure that behaves like a cons list except that accessing an arbitrary element traverses only a logarithmic number of pointers. Specifically, in a list of length $n$, an arbitrary element can be accessed by traversing at most $3\lceil\lg n\rceil-5$ pointers.

In this paper, we present a simple variation on random-access lists that improves this bound and requires traversing at most $2\lceil\lg(n+1)\rceil-3$ pointers. We then present a more complicated variation that improves this bound to $(1+\frac{1}{\sigma})\lfloor\lg n\rfloor+\sigma+9$ for any $\sigma\geq 1$. This shows that it is possible to get asymptotically close to the information-theoretically optimal bound of $\lceil\lg(n+1)\rceil-1$.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Data structures design and analysis**.

Additional Key Words and Phrases: random-access list, Myers list, purely functional data structures

**ACM Reference Format:**
Edward Peters, Yong Qi Foo, and Michael D. Adams. 2025. Pushing the Information-Theoretic Limits of Random Access Lists: Traversing Cons Lists in $\left(1+\frac{1}{\sigma}\right)\lfloor\lg n\rfloor+\sigma+9$ Steps. *Proc. ACM Program. Lang.* 9, ICFP, Article 267 (August 2025), 31 pages. https://doi.org/10.1145/3747536

## 1 Introduction

The singly linked list, also known as the *cons list*, is a cornerstone of functional programming and one of the simplest and most elegant data structures. As cons lists are just cons cells sequentially linked by pointers, operations like *cons* (prepending an element), *car* (accessing the first element), and *cdr* (accessing the list excluding the first element) require only $O(1)$ time and space. However, the simplicity of the cons list comes with a trade-off. Accessing an arbitrary list cell (*lookup*) demands traversing up to $n-1$ pointers for a list of length $n$. This becomes particularly cumbersome in applications with frequent random access, especially in environments where pointer traversals are costly, such as in distributed applications where pointer traversals may involve network access. In such cases, functional programmers might yearn for the efficient random access of arrays. Yet, arrays pose their own challenges in purely functional settings, where previous versions of the array after updates must be preserved and available.

A data structure that improves on the linear-time lookups of cons lists is the purely functional random-access list by Okasaki [24], which we call *Okasaki lists*. In Okasaki lists, *cons*, *car* and *cdr* can also be done in constant time and space, making them asymptotically equally efficient

Authors' Contact Information: Edward Peters, edwardstuyvesantpeters@gmail.com, Independent, Eugene, OR, USA; Yong Qi Foo, yongqi@nus.edu.sg, National University of Singapore, Singapore, Singapore; Michael D. Adams, https://michaeldadams.org, National University of Singapore, Singapore, Singapore.

ACM 2475-1421/2025/8-ART267
https://doi.org/10.1145/3747536

for stack-like behavior. Okasaki lists go beyond cons lists by supporting lookups in no more than $2\lceil \lg(n+1) \rceil + O(1)$[1] pointer traversals, where $n$ is the length of the list.

An alternative to both cons lists and Okasaki lists is the applicative random-access stack by Myers [23], which we call *Myers lists*. Myers lists are structured similarly to cons lists, but every cell in a Myers list has an additional `jump` pointer that points further down the list. The cells that these additional `jump` pointers point to are chosen such that *lookup* for a list of length $n$ requires traversing at most $3\lceil \lg n \rceil - 5$ pointers. Myers lists support constant time and space *cons*, *car* and *cdr* operations just like cons lists and Okasaki lists, and logarithmic-time lookups like Okasaki lists.

Although Okasaki lists and Myers lists perform *lookup* in logarithmic-time, there are subtle differences in the structural characteristics of *lookup* between them. An Okasaki list is essentially a cons list (the "spine") of complete binary trees where the Okasaki list elements are contained in the tree nodes. Querying an arbitrary tree node can be done in logarithmic time via only pointer traversals, but the returned tree node is not an Okasaki list, and tree nodes further down the list may not be reachable from that node (thus, more lookups cannot be done from that node). Alternatively, if *lookup* on an Okasaki list obtains an arbitrarily long *suffix* of the list (in which the leftmost tree root is the desired tree node), then it involves a sequence of *cdr* operations, both on the Okasaki list itself and on the cons list spine, to reconstruct the spine of the suffix. This can incur a logarithmic-space cost and change the pointer structure of the list significantly. In contrast, Myers lists, like cons lists, perform *lookup* by merely traversing pointers until the appropriate list cell is reached and returned. This cell is also the head of a Myers list, from which further lookups to arbitrary succeeding cells can be performed.

This structural difference between Okasaki lists and Myers lists comes with further tradeoffs. Purely functional updates of arbitrary Okasaki list elements can be done in logarithmic time and space, unlike Myers lists and cons lists which require linear time and space. On the other hand, the structural characteristics of lookups in Myers lists allow them to efficiently support *range* or *path queries* (querying some property of a contiguous sequence of elements) [20] by embedding additional information in the pointer fields of each cell. For example, if the outgoing pointers of each cell in a Myers list store the sum of the elements they traverse, then the sum of an arbitrary contiguous sequence of elements can be obtained in logarithmically many pointer traversals. In addition, the fact that two heads can be *cons*-ed onto the same tail without modifying the tail's structure (tail-sharing) makes Myers lists ideal for path queries over trees [1]. Myers lists have been more broadly used for static analysis [5], and applications that perform computations over intervals [2, 11] can be adapted to use Myers lists.

However, the $3\lceil \lg n \rceil - 5$ lookup performance of Myers lists is well above the information-theoretic lower bound of $\lceil \lg(n+1) \rceil - 1$ for traversals on linked data structures with two pointers. For instance, starting from the root of a perfect binary tree with $n$ nodes, accessing any other node requires traversing no more than $\lceil \lg(n+1) \rceil - 1$ pointers. Thus, the motivating question of this paper is: *how far can we push the theoretical lookup performance of random-access lists*?

A trivial approach might involve increasing the number of outgoing pointers in each cell of Okasaki lists or Myers lists from 2 to $b$, thereby changing the base of the logarithm from 2 to $b$, which is equivalent to reducing lookup costs by a constant factor of $\lg b$. However, this increases space usage and does not provide novel theoretical or algorithmic insights. Instead, our objective is to design a new random-access list that (1) improves theoretical lookup performance without increasing the number of outgoing pointers per cell, and (2) retains the structural characteristics of Myers lists. This structure thus optimizes workloads (a) requiring purely functional list structures

[1] In this paper, lg denotes $\log_2$.

```
type 'a cell = { value: 'a; next: 'a cell; length: int; jump: 'a cell }

val init   : 'a -> 'a cell   ;; val cdr    : 'a cell -> 'a cell        ;;
val length : 'a cell -> int  ;; val cons   : 'a -> 'a cell -> 'a cell  ;;
val car    : 'a cell -> 'a   ;; val lookup : 'a cell -> int -> 'a cell ;;
```

Fig. 1. The `cell` type and interface for Myers lists.

with cons-list-like properties (tail-sharing, etc.), (b) are dominated by lookups, and (c) where pointer traversals may be expensive relative to other computations.

To achieve this goal, this paper explores modifications to Myers lists that improves its lookup performance. First, we describe an interface for working with Myers lists (Section 2). This is followed by the main contributions of this paper:

(1) Section 3 reviews traditional Myers lists [23] and provides a recursive construction, yielding a structure similar to the one presented by Okasaki [24]. However, unlike Myers and Okasaki, this paper leverages the recursive structure of Myers lists to perform a new algorithmic-complexity analysis via recurrence equations, instead of using skew-binary numbers.
(2) Section 4 introduces a novel random-access list by making minor structural modifications to Myers lists, improving its lookup complexity to $2\lceil \lg(n+1) \rceil - 2$ pointers, effectively matching that of Okasaki lists while retaining the properties stated in our objectives.
(3) Section 5 introduces a variation on the list presented in Section 4, that, while significantly more complex structurally, further improves the lookup performance to $(1+1/\sigma)\lfloor \lg n \rfloor + \sigma + 9$ pointers. This new random-access list shows that it is possible to design a data structure whose lookup performance comes arbitrarily close to the information-theoretical limit of $\lceil \lg(n+1) \rceil - 1$, while retaining the properties stated in our objectives.

Finally, Section 6 compares the performance of these variations, Section 7 discusses related work and Section 8 concludes. Appendix A contains an embedded artifact and links to the online artifact [26]. All code presented in this paper is written in OCaml.

## 2 Myers Lists

To set up the discussion for the rest of this paper, in this section, we provide a high-level overview of the interface for working with Myers lists. We first describe the cell structure of individual Myers list cells (Section 2.1), then show the API for working with Myers lists and the implementations of some of these functions (Section 2.2), and finally summarize the main modifications we will make to the original Myers list to improve lookup performance (Section 2.3).

### 2.1 Myers List Cells

Cons lists consist of cells, where each cell has a `car` field storing the value of that cell, and a `cdr` pointer that points to the next sequential cell in the cons list. Similarly, Myers lists also consist of cells, where each cell has the type in Figure 1, and (1) `value` is the value stored in the cell, (2) `next` points to the next cell in the list, (3) `length` is the number of cells in the list after and including this cell, and (4) `jump` points further along the Myers list in a way that allows traversal in logarithmically many steps. `value` and `next`, respectively, correspond to `car` and `cdr` of a cons list cell. `length` is also used to decide between taking the `next` or `jump` pointers during a lookup; it is also possible to store the length of the entire list at the head only instead of the length at every cell, although this places some computational overhead on finding the shortest path between two cells.

```
let init (x: 'a): 'a cell =
  let rec c: 'a cell = { value = x; next = c; length = 1; jump = c } in c
let length (xs: 'a cell): int = xs.length
let car (xs: 'a cell): 'a = xs.value
let cdr (xs: 'a cell): 'a cell = xs.next
```

Fig. 2. Function implementations that are shared across all Myers lists variants.

```
let cons (x: 'a) (xs: 'a cell): 'a cell =
  { value = x; next = xs; length = xs.length + 1; jump = ... }
let rec lookup (xs: 'a cell) (len: int): 'a cell =
  if xs.length = len then xs
  else if ... then lookup xs.jump len
  else lookup xs.next len
```

Fig. 3. Implementation templates of cons and lookup; ellipses (. . .) depend on the Myers list variant.

## 2.2 Myers List API

Figure 1 shows the functions that form the public interface for working with Myers lists. Similar to their analogues for cons lists, all of the functions in the API do not perform mutation on data; Myers lists are *purely functional* data structures. The functions described in the API are as one would typically expect of cons lists: (1) `init x` initializes a list of length 1 with `x` as its only element, (2) `length xs` returns the length of the list where `xs` is the list head, (3) `car xs` retrieves the first element of the list starting at `xs`, (4) `cdr xs` obtains the list where the first cell of `xs` is omitted (this assumes `xs` has `length` greater than 1, otherwise `cdr xs` just returns `xs`), (5) `cons x xs` prepends `x` onto `xs`, and (6) `lookup xs len` retrieves the *list cell* in `xs` whose length is `len` (assuming `len` is within bounds). Just like cons lists, Myers lists do not support efficient implementations for purely functional *append* (appending an element to the list), *concat* (list concatenation) or *update* (updating a list element at a particular cell).

Figure 2 shows the implementations that are shared by all variants of Myers lists presented in this paper for some functions in the API. `length`, `car` and `cdr` conveniently access the fields of the list cell. To simplify our presentation, we use the convention that all Myers lists are non-empty, and as shown in the implementation of `init`, a cell with `length` 1 has its `next` and `jump` pointers point to itself. Empty Myers lists can be supported with small modifications to the interface and implementation; we describe these in the supplementary material of this paper.

## 2.3 Our Focus

The goal of this paper is to design variants of the original Myers list [23] to reduce the number of pointer dereferences performed by `lookup`. This is done solely by modifying where the `jump` pointer of each cell points to. This results in new list structures that require fewer pointer traversals for *lookup*, i.e., shorter paths between two list cells. A consequence of this is that `cons` and `lookup` look like the generic implementation templates shown in Figure 3, where each variant of Myers lists we present in this paper fills in the ellipses (. . .) differently.

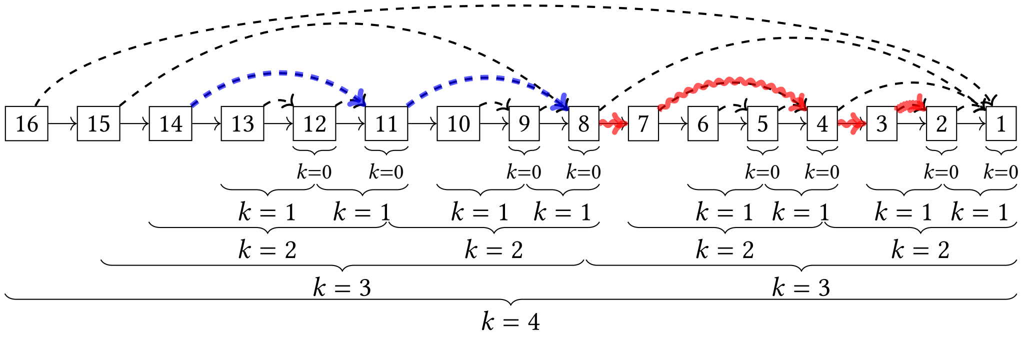


Fig. 4. An example basic Myers list of length 16. Each box represents a cell. Cell `value` fields are not shown. The number in the box is the `length` of that cell. Braces indicate the degree $k$ of the region containing those cells. Solid arrows ⟶: `next` pointers. Dashed arrows ⇢: `jump` pointers. Thick arrows ⇢ ⇝: path of the traversal from cell 14 to cell 2.

## 3 Basic Myers Lists

This section provides an overview of the original Myers lists, which we call *basic Myers lists*. We refer readers unfamiliar with basic Myers lists to the original paper [23] for more details. However, while Myers [23] uses an iterative construction, we present a recursive construction (Section 3.1). This recursive construction allows us to view Myers lists as trees (Section 3.2). We then describe how pointers are traversed during a *lookup* (Section 3.3), and we use the recursive construction of Myers lists to analyze its lookup performance via recurrence equations (Section 3.4).

### 3.1 Construction of Basic Myers Lists

Myers [23] describes an iterative construction rule (repeated application of `cons`) which we also re-state in the supplementary material. The essence of the iterative construction is that the jump pointer of a cell `c` is assigned using the following rule:

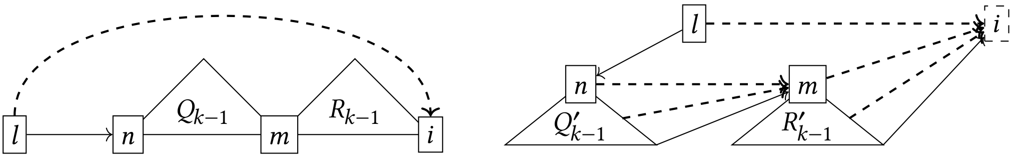
Let `xs` be `c.next`, `ys` be `xs.jump`, and `zs` be `ys.jump`. Then,

$$\texttt{c.jump} = \begin{cases} \texttt{zs} & \text{if } \texttt{xs.length} - \texttt{ys.length} = \texttt{ys.length} - \texttt{zs.length} \\ \texttt{xs} & \text{otherwise} \end{cases}$$

This rule of assigning `jump` pointers, essentially, uniquely characterizes the structure of basic Myers lists. Although this rule causes a fraction of cells in a Myers list to have their `jump` pointer being identical to their `next` pointer, it is conceptually simple and makes `cons` easy to implement, while still allowing *lookup* to be efficient.

An example basic Myers list of length 16 is shown in Figure 4. Removing `jump` pointers in Figure 4 reveals the structure of a cons list. The `jump` pointers of Myers lists allow us to traverse through Myers lists more quickly. For example, to traverse to cell 2 from cell 14, instead of traversing sequentially along the `next` pointers, we can take the pointers marked by the thick arrows. The distinction between the different thick arrows shown in Figure 4 will be described in Section 3.2.

Although Myers [23] provides an iterative construction, the resulting structure of Myers lists exhibit a recursive pattern. As shown in Figure 4, the braces highlight regions that repeat the same pointer structure. For example, the cells and pointers between cell 14 and cell 8 exactly match (modulo renumbering) those between cell 8 and cell 1. We assign each of the sub-regions a degree $k$



(a) Flat view of construction rule, as discussed in Section 3.1.

(b) Tree view of construction rule, as discussed in Section 3.2. $Q'_{k-1}$ and $R'_{k-1}$ are the cells of $Q_{k-1}$ and $R_{k-1}$ excluding their last cells, respectively.

Fig. 5. Construction rule for region of Myers lists of degree $k$.

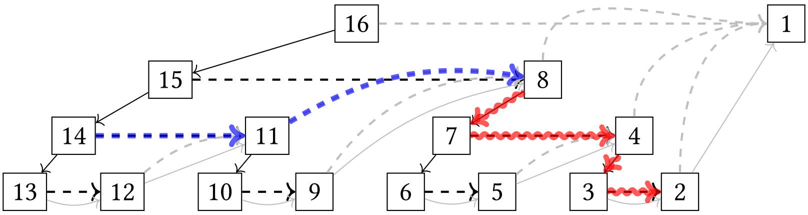


Fig. 6. The Myers list shown in Figure 4 but laid out as a tree. Black arrows ⟶ ⇢ are pointers taken during the inward phase of a traversal. For the traversal from cell 14 to cell 2, the outward phase is shown as thick dashed arrows ⇢, and the inward phase is shown as thick squiggly arrows ⇝.

that indicates how many levels of recursion it contains. For example, the cells and pointers between cell 14 and cell 8 have degree $k = 3$.

We define the degree of regions explicitly using the recursive construction rule shown in Figure 5a:

(1) A region has degree 0 if it is a single cell.
(2) A region has degree $k > 0$ if it consists of
  (a) two regions $Q_{k-1}$ and $R_{k-1}$, both of degree $k-1$, such that the last cell of $Q_{k-1}$ (shown as cell $m$ in Figure 5a) is also the first cell of $R_{k-1}$ and
  (b) an additional cell $l$ not part of $Q_{k-1}$ or $R_{k-1}$ whose `next` pointer points to the first cell of $Q_{k-1}$ (cell $n$ in Figure 5a) and `jump` pointer points to the last cell of $R_{k-1}$ (cell $i$ in Figure 5a).

This rule constructs regions of Myers lists of only length $2^k$ for some $k$. Then, a Myers list of length $2^k$ is just a region of length $2^k$ whose rightmost cell has `length` 1. Finally, Myers lists of other lengths can be obtained by taking the appropriately long suffix of any Myers list using `lookup`.

### 3.2 Myers Lists as Trees

The key idea that we will frequently use in this paper is to view Myers lists as trees due to their recursive structure, which gives us greater intuition on how lookups can be performed in logarithmically many pointers. The way to do so is shown in Figure 5b: given a region $P_k$ with sub-regions $Q_{k-1}$ and $R_{k-1}$, all the cells excluding the last cell of $P_k$ can be viewed as a tree rooted by its first cell, and $Q_{k-1}$ and $R_{k-1}$ are its left and right subtrees, respectively.

An example of this is shown in Figure 6, which is the same as the Myers list shown in Figure 4, rearranged to better show its tree structure. Cells 2 to 16 can be viewed as a tree rooted at cell 16, and its left and right subtrees are rooted at cells 15 and 8, respectively.

### 3.3 Lookups in Basic Myers Lists

As described in Section 1, *lookup* on a Myers list involves only a traversal across `next` and `jump` pointers until the appropriate cell is reached. Myers [23] describes an algorithm for *lookup* which we also re-state in the supplementary material. In essence, *lookup* can be expressed as the following (assume `len` is within bounds):

$$\texttt{lookup xs len} = \begin{cases} \texttt{xs} & \text{if } \texttt{xs.length} = \texttt{len} \\ \texttt{lookup xs.jump len} & \text{if } \texttt{xs.jump.length} \geq \texttt{len} \\ \texttt{lookup xs.next len} & \text{otherwise} \end{cases}$$

An example path taken by a *lookup* is shown in Figure 4. The thick arrows show the pointers traversed when performing *lookup* on the list starting at cell 14, targeting the list starting at cell 2. At cells 14, 11, 7 and 3, the `jump` pointer does not bring us past cell 2, and thus the `jump` pointer is traversed. At cells 8 and 4, only the `next` pointer can be traversed to bring us to cell 2.

The tree view of basic Myers lists allows us to conceptually divide a traversal between two cells `src` and `dst` into two phases:

- *Outward phase*: a traversal to the last cell of one of the enclosing regions of `src`, thereby traversing *out* of a subtree containing `src`. This phase consists of following `jump` pointers, shown as dashed arrows in Figure 4 and Figure 6.
- *Inward phase*: a descent *into* `dst` from the root of some tree containing `dst`. This phase consists of following the pointers shown as black arrows in Figure 6.

An example of this is shown for the traversal from cell 14 to cell 2 in Figure 4 and Figure 6, where the outward phase is shown as thick dashed arrows, while the inward phase is shown as thick squiggly arrows. The outward phase traverses out of the subtree rooted at cell 15, arriving at cell 8. The inward phase then descends into the tree rooted at cell 8, arriving at cell 2.

Being able to divide traversals into two phases, coupled with the tree view of Myers lists, allows us to analyze the lookup performance of Myers lists via recurrence equations, instead of the analysis via skew binary numbers as performed by Myers [23].

### 3.4 Analysis of Lookups in Basic Myers Lists

Now that we have built some intuition behind traversals between two cells of Myers lists, in this subsection, we reproduce the result by Myers [23], giving an upper bound for the number of pointers traversed when accessing an arbitrary element in a Myers list. This is the main result of this subsection, shown as Theorem 3.1.

Theorem 3.1. *Given a Myers list $L$ of length $n$ containing cells* `src` *and* `dst` *where* `src.length` $\geq$ `dst.length`*, the number of pointers traversed to get from* `src` *to* `dst` *is no greater than* $\lceil \lg n \rceil$ *for all* $1 \leq n \leq 4$*, and* $3\lceil \lg n \rceil - 5$ *for all* $n > 4$.

In Section 3.2, we showed that regions of Myers lists can be viewed as trees. As such, we first relate the degree of a region formed by a list and the number of cells in it. This is shown as Lemma 3.2. Its proof is in the supplementary material; the intuition behind it is that the degree $k$ of a region corresponds to one more than the height of the tree it can be viewed as.

Lemma 3.2. *Given a basic Myers list $L$ of length $n$, the smallest basic Myers list $L'$ that contains $L$ as a suffix and forms a region of degree $k$ satisfies $k = \lceil \lg n \rceil$.*

The goal is to give an upper bound for the number of pointers traversed between arbitrary cells in a Myers list $L$ of length $n$. Given Lemma 3.2, the approach we take is to do the same for a Myers list of degree $k = \lceil \lg n \rceil$ that contains $L$ as a suffix. Since every suffix of a Myers list is also a Myers list, this is also an upper bound for $L$. The reason for this approach is that, as we have shown in Section 3.2, Myers lists that form complete regions assemble into perfect binary trees, which are amenable to analysis via recurrence equations.

*3.4.1 Preliminaries for Analysis of Lookups in Basic Myers Lists.* For the rest of this subsection, we assume $P_k$ is a region of a Myers list of degree $k$. In addition, we write $\mathit{tree}(P_k)$ to denote every cell except the last cell of $P_k$—in other words, if $k > 0$, $\mathit{tree}(P_k)$ are the cells of $P_k$ that can be viewed as a tree. Finally, following the conventions in Section 3.1 and Section 3.2, every region $P_k$ of degree $k > 0$ also contains two sub-regions of degree $k - 1$, which we denote as $Q_{k-1}$ and $R_{k-1}$.

*3.4.2 Performance of Lookups in Basic Myers Lists.* As described in Section 3.2, a traversal between two arbitrary cells in a Myers list consists of an outward phase and an inward phase. Thus, we analyze each phase before combining these results to give an upper bound of the path length for any traversal. The upper bounds of the lengths of the outward and inward phases are shown as Lemma 3.3 and Lemma 3.4, respectively.

Lemma 3.3. *Let $O_k$ be the upper bound for the number of pointers traversed to get from an arbitrary cell $x$ in $\mathit{tree}(P_k)$ to the last cell of $P_k$. For all $k > 0$, $O_k = k$.*

Proof. We proceed by induction on $k$. When $k = 1$, $O_1 = 1$ since there is only one cell in $\mathit{tree}(P_1)$, and taking the `next` pointer brings us to the last cell of $P_1$. When $k > 1$, there are three possible locations of $x$:

*Case* 1. $x$ is the first cell of $P_k$. In this case, the `jump` pointer takes us directly to the last cell, therefore requiring only 1 pointer.

*Case* 2. $x$ is in $\mathit{tree}(Q_{k-1})$. This requires no more than $O_{k-1}$ pointer traversals to get to the last cell in $Q_{k-1}$, which is also the first cell of $R_{k-1}$. From the first cell of $R_{k-1}$, the `jump` pointer takes us directly to the last cell of $R_{k-1}$, which is the last cell of $P_k$. As such, this path has no more than $O_{k-1} + 1$ pointers.

*Case* 3. $x$ is in $\mathit{tree}(R_{k-1})$, thus requiring no more than $O_{k-1}$ pointer traversals.

Therefore, when $k > 1$, $O_k = \max(1, O_{k-1} + 1, O_{k-1})$. By the induction hypothesis, $O_{k-1} = k - 1$. Thus, $O_k = \max(1, O_{k-1} + 1, O_{k-1}) = \max(1, k, k - 1) = k$. Hence, $O_k = k$ for all $k > 0$. □

Lemma 3.4. *Let $I_k$ be the upper bound for the number of pointers traversed to get from the first cell of $P_k$ to an arbitrary cell $x$ in $\mathit{tree}(P_k)$. For all $k > 0$, $I_k = 2k - 2$.*

Proof. We proceed by induction on $k$. When $k = 1$, $I_1 = 0 = 2(1) - 2$ because $\mathit{tree}(P_1)$ has only one cell. When $k > 1$, there are three possible locations of $x$:

*Case* 1. $x$ is the first cell of $P_k$, thus the path has 0 pointers.

*Case* 2. $x$ is in $\mathit{tree}(Q_{k-1})$. We take the `next` pointer of the first cell of $P_k$ to the first cell $c$ of $Q_{k-1}$. The path from $c$ to $x$ has no more than $I_{k-1}$ pointers. Thus, in this case, path consists of no more than $1 + I_{k-1}$ pointers.

*Case* 3. $x$ is in $\mathit{tree}(R_{k-1})$. From the first cell of $P_k$ we can take the `next` pointer then the `jump` pointer to the first cell $c$ of $R_{k-1}$. The path from $c$ to $x$ has no more than $I_{k-1}$ pointers. Thus, in this case, the path consists of no more than $2 + I_{k-1}$ pointers.

Therefore, when $k > 1$, $I_k = \max(0, 1 + I_{k-1}, 2 + I_{k-1})$. By the induction hypothesis, $I_{k-1} = 2(k-1) - 2 = 2k - 4$. As such, $I_k = \max(0, 1 + I_{k-1}, 2 + I_{k-1}) = \max(0, 1 + 2k - 4, 2 + 2k - 4) = 2k - 2$. Hence, $I_k = 2k - 2$ for all $k > 0$. □

Now that the upper bounds for the lengths of the outward and inward phases of a traversal have been found in Lemma 3.3 and Lemma 3.4, respectively, we prove Lemma 3.5, providing an upper bound for the number of pointers traversed between *any* two cells.

Lemma 3.5. *Let $T_k$ be the upper bound for the number of pointers traversed to get from an arbitrary cell* `src` *to an arbitrary cell* `dst` *where both* `src` *and* `dst` *are in $P_k$ and* `src.length` $\geq$ `dst.length`*. For all $k \geq 0$, $T_k = \max(k, 3k - 5)$, i.e. $T_k = k$ if $0 \leq k \leq 2$, $3k - 5$ otherwise.*

Proof. We proceed by induction on $k$. When $k = 0$, $T_0 = 0 = \max(0, 3(0) - 5)$ since $P_0$ only has one cell. When $k = 1$, $T_1 = 1 = \max(1, 3(1) - 5)$ since $P_1$ only has two cells. When $k > 1$, we can group the locations of `src` and `dst` into four cases:

*Case* 1. `src` is the first cell in $P_k$. `dst` can either be in *tree*$(P_k)$ or be the last cell in $P_k$. Therefore, the path from `src` to `dst` consists of no more than $\max(I_k, 1)$ pointers.

*Case* 2. `dst` is the last cell in $P_k$. Therefore, the path consists of no more than $O_k$ pointers.

*Case* 3. Both `src` and `dst` are in *tree*$(Q_{k-1})$, or both are in *tree*$(R_{k-1})$. Since the two cells are in the same subregion of degree $k - 1$, the path consists of no more than $T_{k-1}$ pointers.

*Case* 4. `src` is in *tree*$(Q_{k-1})$ and `dst` is in *tree*$(R_{k-1})$. The path consists of a traversal from `src` to the last cell of $Q_{k-1}$ (which is also the first cell of $R_{k-1}$) in no more than $O_{k-1}$ pointers, then descending to `dst` in no more than $I_{k-1}$ pointers. As such, this path has no more than $O_{k-1} + I_{k-1}$ pointers.

Therefore, when $k > 1$, $T_k = \max(I_k, 1, O_k, T_{k-1}, O_{k-1} + I_{k-1})$. By Lemma 3.3 and Lemma 3.4, for all $k > 0$, $O_k = k$ and $I_k = 2k - 2$. As such, $T_k = \max(I_k, 1, O_k, T_{k-1}, O_{k-1} + I_{k-1}) = \max(2k - 2, 1, k, T_{k-1}, 3k - 5) = \max(k, T_{k-1}, 3k - 5)$. By the induction hypothesis, $T_{k-1} = \max(k - 1, 3(k - 1) - 5) = \max(k - 1, 3k - 8)$. This means that $T_k = \max(k, k - 1, 3k - 8, 3k - 5) = \max(k, 3k - 5)$. Hence, $T_k = \max(k, 3k - 5)$ for all $k \geq 0$. □

Overall, given a Myers list $L$ of length $n$, by Lemma 3.2, the smallest Myers list $L'$ containing $L$ as a suffix and forms a region of degree $k$ satisfies $k = \lceil \lg n \rceil$. By Lemma 3.5, the path length between two arbitrary cells in $L'$ is no greater than $T_{\lceil \lg n \rceil} = \max(\lceil \lg n \rceil, 3\lceil \lg n \rceil - 5)$. Since $L$ is a suffix of $L'$, the path length between any arbitrary cells in $L$ is also no greater than $\max(\lceil \lg n \rceil, 3\lceil \lg n \rceil - 5)$, thereby proving Theorem 3.1.

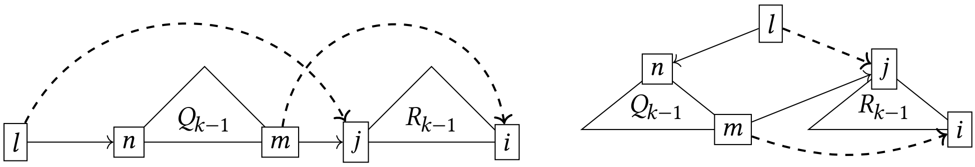
## 4 Improved Myers Lists

This section presents a modification to the construction rule for Myers lists that improves the worst case path length from $3 \lceil \lg n \rceil - 5$ to $2 \lceil \lg (n + 1) \rceil - 3$. We call this new data structure *improved Myers lists*. The construction rule for improved Myers lists is shown in Figure 7. Following the same format in Section 3, Section 4.1 presents how improved Myers lists are constructed, Section 4.2 describes lookups in improved Myers lists, and Section 4.3 analyzes the performance of lookups in improved Myers lists.

### 4.1 Construction of Improved Myers Lists

Section 3.4 shows that the upper bounds of the path lengths for the outward and inward phases are $k$ and $2k + O(1)$, respectively. The inward phase cost is greater because getting from the root of a tree to the root of its right subtree requires traversing two pointers. By modifying the list so that the inward phase of a traversal requires only $k$ pointer traversals, the total cost of lookups can be brought down to $2k + O(1)$ pointers. A modification to Myers lists that allows this is to have the `jump` pointer of the first cell of every region point to the first cell of its second sub-region. This is so that accessing the root of either subtree is done with one pointer traversal.[2]

[2] Though we present our changes as a straightforward design narrative, discovering this involved experimentation with many different construction rules. At one point we enumerated as many construction rules as we could think of and analyzed each. This is the one for which the worst case is the best. The ease with which different construction rules could be enumerated is a strength of the recursive construction rule compared to the skew-binary numbers as presented in [23].



(a) Flat view of construction rule.

(b) Tree view of construction rule.

Fig. 7. Construction rule for region of improved Myers lists of degree $k$.

Improved Myers lists consist of cells of the same type as shown in Figure 1. However, we use the new recursive construction rule shown in Figure 7 to characterize the structure of improved Myers lists. The differences between this new construction rule and the one from Section 3.1 are:

(1) the `jump` pointer of the first cell of a region $P_k$ now points to the first cell of the second subregion $R_{k-1}$ instead of the last cell of $P_k$, and
(2) subregions $Q_{k-1}$ and $R_{k-1}$ no longer share cells.

This new construction rule allows regions to neatly assemble into perfect binary trees via the tree view of the construction rule, shown in Figure 7b. Essentially, subregions $Q_{k-1}$ and $R_{k-1}$ form the left and right subtrees of a region. Unlike basic Myers lists, accessing the root of the right subtree from the root of a tree can be done by traversing a single pointer.

An example of an improved Myers list of degree 3 and length 15 is shown in Figure 8. Note that removing grey arrows from Figure 8b reveals a perfect binary tree.

Just like for basic Myers lists, the recursive construction rule gives rise to a straightforward analysis of its improved lookup performance. Although an iterative construction rule is not necessary for this analysis, nonetheless, we provide an implementation of `cons` for improved Myers lists in the supplementary material.

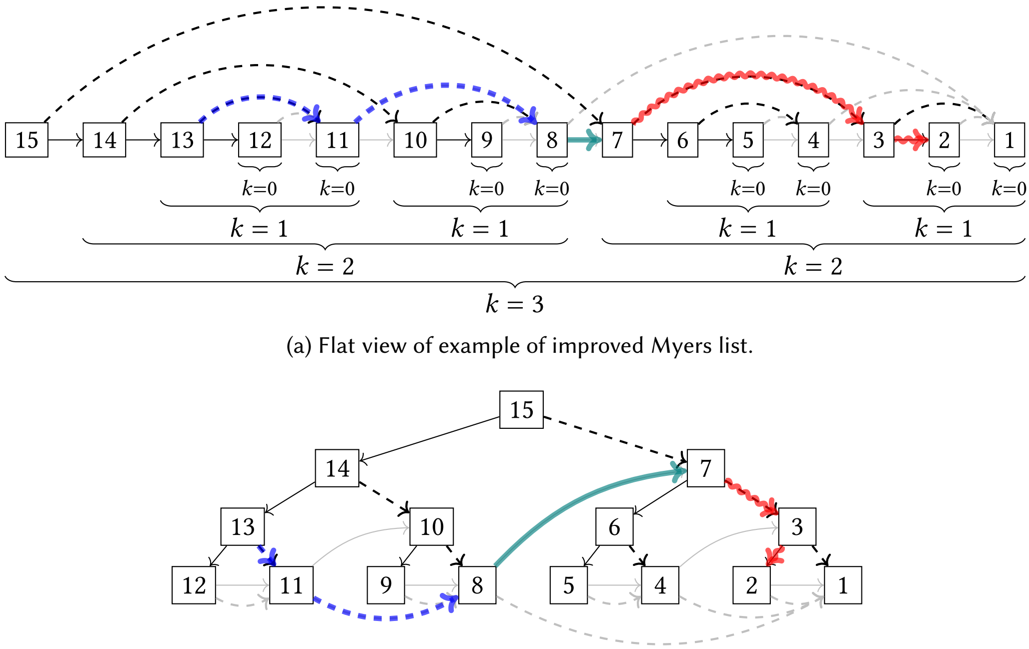
### 4.2 Lookups in Improved Myers Lists

The rule and implementation of *lookup* for improved Myers lists is the same as that of basic Myers lists as described in Section 3.3. Although improved Myers lists are structurally different to basic Myers lists, we can still divide a traversal between two arbitrary cells `src` and `dst` into phases in a similar manner:

- *Outward phase*: a traversal to the last cell of one of the regions containing `src`, i.e., traversing to the rightmost leaf of one of the subtrees containing `src`. This is done by descending via `jump` pointers to the leaf layer, then taking `jump` pointers to the desired rightmost leaf.
- From the rightmost leaf, taking the `next` pointer brings us to the root of some subtree containing `dst`.
- *Inward phase*: a descent to `dst`.

For example, the traversal from cell 13 to cell 2 in Figure 8 consists of (1) the outward phase, which is the descent to cell 8 via pointers marked by thick dashed arrows, (2) the `next` pointer to cell 7 marked by a thick solid arrow, and (3) the inward phase, which is the descent to cell 2 marked by thick squiggly arrows.

The inward phase of a traversal can now be viewed as a descent into a perfect binary tree, providing some intuition behind its reduced cost.



(a) Flat view of example of improved Myers list.

(b) The improved Myers list shown in Figure 8a but laid out as a tree.

Fig. 8. An example improved Myers list of length 15. The traversal from cell 13 to cell 2 consists of (1) the outward phase shown as thick dashed arrows ⇢ to cell 8, (2) the `next` pointer shown as a thick solid arrow → to cell 7, and finally (3) the inward phase shown as thick squiggly arrows ⇝ to cell 2.

### 4.3 Analysis of Lookups in Improved Myers Lists

As with Section 3.4, we analyze the number of pointers traversed when performing lookups in an improved Myers lists. The structure and techniques used in this analysis are the same as in Section 3.4. Hence, this subsection is brief. The main result of this subsection is Theorem 4.1.

Theorem 4.1. *Given an improved Myers list $L$ of length $n$ containing arbitrary cells* `src` *and* `dst` *such that* `src.length` $\geq$ `dst.length`*, the number of pointers traversed to get from* `src` *to* `dst` *is* 0 *for $n = 1$ and no greater than $2\lceil \lg(n+1) \rceil - 3$ for all $n > 1$.*

First, we relate the degree of regions and the length of improved Myers lists, shown as Lemma 4.2. We relegate its proof to the supplementary material to save space; the intuition behind it is that an improved Myers list of degree $k$ can be viewed as a perfect binary tree of height $k$.

Lemma 4.2. *Given an improved Myers list $L$ of length $n$, the smallest improved Myers list $L'$ that contains $L$ as a suffix and forms a region of degree $k$ satisfies $k = \lceil \lg(n+1) \rceil - 1$.*

*4.3.1 Preliminaries for Analysis of Lookups in Improved Myers Lists.* For the rest of the analysis, we assume $P_k$ is a region of an improved Myers list of degree $k$. Following the conventions in Section 4.1, every region $P_k$ of degree $k > 0$ also contains two sub-regions of degree $k-1$, which we denote by $Q_{k-1}$ and $R_{k-1}$.

*4.3.2 Performance of Lookups in Improved Myers Lists.* Unlike Section 3.4, we analyze the inward phase path length first (Lemma 4.3), since the outward phase (Lemma 4.4) also involves a descent into the leaf layer. Then, like Section 3.4, after analyzing the path lengths of the outward and inward phases, we get the path length of any traversal (Lemma 4.5), allowing us to prove Theorem 4.1.

Lemma 4.3. *Let $I_k$ be the upper bound for the number of pointers traversed to get from the first cell of $P_k$ to an arbitrary cell $x$ in $P_k$. For all $k \geq 0$, $I_k = k$.*

Proof. We proceed by induction on $k$. When $k = 0$, $I_0 = 0$ since $P_0$ consists of one cell. When $k > 0$, there are three possible locations of $x$:

*Case* 1. $x$ is the first cell of $P_k$, and thus the path has 0 pointers.

*Case* 2. $x$ is in $Q_{k-1}$. We take the `next` pointer to the first cell in $Q_{k-1}$, then take up to $I_{k-1}$ pointers to reach $x$, for a total of up to $1 + I_{k-1}$ pointers.

*Case* 3. $x$ is in $R_{k-1}$. We take the `jump` pointer to the first cell in $R_{k-1}$, then take up to $I_{k-1}$ pointers to reach $x$, for a total of up to $1 + I_{k-1}$ pointers.

Therefore, when $k > 0$, $I_k = \max(0, 1 + I_{k-1})$. By the induction hypothesis, $I_{k-1} = k - 1$, thus $I_k = \max(0, 1 + I_{k-1}) = \max(0, 1 + k - 1) = k$. Hence, $I_k = k$ for all $k \geq 0$. □

Lemma 4.4. *Let $O_k$ be the upper bound for the number of pointers traversed to get from an arbitrary cell $x$ in $P_k$ to the last cell of $P_k$. For all $k \geq 0$, $O_k = k$.*

Proof. We proceed by induction on $k$. When $k = 0$, $O_0 = 0$ since there is only one cell in $P_0$. When $k > 0$, there are three possible locations of $x$:

*Case* 1. $x$ is the first cell of $P_k$, and thus the path consists of no more than $I_k$ pointers. By Lemma 4.3, $I_k = k$, therefore this path has no more than $k$ pointers.

*Case* 2. $x$ is in $Q_{k-1}$. Therefore, we traverse up to $O_{k-1}$ pointers to go from $x$ to the last cell in $Q_{k-1}$, then one `jump` pointer to arrive at the last cell in $P_k$, therefore requiring up to $O_{k-1} + 1$ pointers.

*Case* 3. $x$ is in $R_{k-1}$, therefore requiring up to $O_{k-1}$ pointers to get to the last cell of $R_{k-1}$, which is the last cell of $P_k$.

Therefore, when $k > 0$, $O_k = \max(k, O_{k-1} + 1, O_{k-1})$. By the induction hypothesis, $O_{k-1} = k - 1$, thus, $O_k = \max(k, O_{k-1} + 1, O_{k-1}) = \max(k, k - 1 + 1, k - 1) = k$. Hence, $O_k = k$ for all $k \geq 0$. □

Lemma 4.5. *Let $T_k$ be the upper bound for the total number of pointers traversed to get from an arbitrary cell* `src` *to an arbitrary cell* `dst` *where both* `src` *and* `dst` *are in $P_k$ and* `src.length` $\geq$ `dst.length`*. For all $k \geq 0$, $T_k = \max(k, 2k - 1)$, i.e., $T_k = 0$ if $k = 0$, $2k - 1$ otherwise.*

Proof. We proceed by induction on $k$. When $k = 0$, $T_0 = 0 = \max(0, 2(0) - 1)$ because $P_0$ only has one cell. When $k > 0$, the locations of `src` and `dst` belong to one of four cases:

*Case* 1. `src` is the first cell of $P_k$. The path in this case consists of no more than $I_k$ pointers.

*Case* 2. `dst` is the last cell of $P_k$. The path in this case consists of no more than $O_k$ pointers.

*Case* 3. Both `src` and `dst` are in $Q_{k-1}$ or both are in $R_{k-1}$. Since both cells are in the same sub-region, the path consists of no more than $T_{k-1}$ pointers.

*Case* 4. `src` is in $Q_{k-1}$ and `dst` is in $R_{k-1}$. The path consists of the outward phase, having up to $O_{k-1}$ pointers from `src` to the last cell $m$ of $Q_{k-1}$. From $m$, the `next` pointer goes to the first cell $j$ of $R_{k-1}$. Finally, the inward phase consists of up to $I_{k-1}$ pointers to descend from $j$ into `dst`. Overall, the path consists of no more than $O_{k-1} + I_{k-1} + 1$ pointers.

Thus, $T_k = \max(I_k, O_k, T_{k-1}, O_{k-1} + I_{k-1} + 1)$ pointers for all $k > 0$. By Lemma 4.3 and Lemma 4.4, $O_k = I_k = k$ for all $k \geq 0$, thus $T_k = \max(I_k, O_k, T_{k-1}, O_{k-1} + I_{k-1} + 1) = \max(k, T_{k-1}, 2k - 1)$. By the induction hypothesis, $T_{k-1} = \max(k-1, 2(k-1)-1)$, as such, $T_k = \max(k, T_{k-1}, 2k-1) = \max(k, k-1, 2k-3, 2k-1) = \max(k, 2k-1)$. Hence, $T_k = \max(k, 2k-1)$ for all $k \geq 0$. □

Finally, Theorem 4.1 is a consequence of Lemma 4.2 and Lemma 4.5.

## 5 Advanced Myers Lists

In the previous section, we modified where the `jump` pointer of cells in basic Myers lists points to, arriving at improved Myers lists which can be viewed as perfect binary trees with each cell pointing to the roots of its subtrees. The result is that, as shown in Section 4.3, lookups in improved Myers lists can be performed by traversing no more than $2\lceil\lg(n+1)\rceil - 3$ pointers. However, as discussed in Section 1, information theory places a lower bound of $\lceil\lg(n+1)\rceil - 1$ for lookups. We thus ask the question: *can we do better*?

In this section, we present a modification to improved Myers lists so that traversals between any two cells comes arbitrarily close to this theoretical bound—given any $\sigma \geq 1$, the maximum path length is $\left\lfloor\left(1+\frac{1}{\sigma}\right)\lfloor\lg n\rfloor\right\rfloor + \sigma + 9$. This trades a smaller coefficient in front of $\lg n$ for a larger constant at the end. For example, when $\sigma = 4$, the maximum path length is $\left\lfloor\frac{5}{4}\lfloor\lg n\rfloor\right\rfloor + 13$. When $\sigma = 8$, it is $\left\lfloor\frac{9}{8}\lfloor\lg n\rfloor\right\rfloor + 17$. We call this new data structure *advanced Myers lists*.

Like basic and improved Myers lists, advanced Myers lists consist of cells whose type is shown in Figure 1. The only difference between advanced Myers lists and the other variants is where the `jump` pointer of each cell points to.

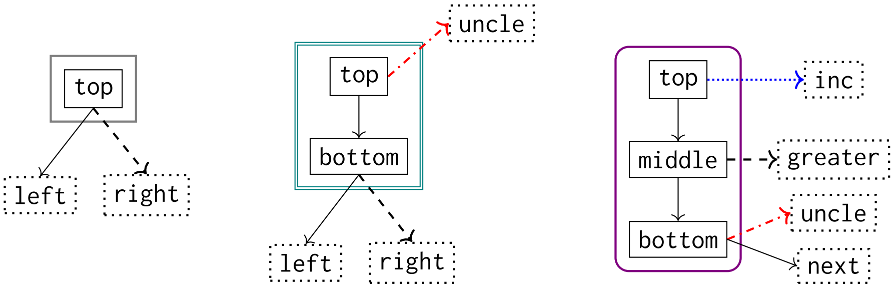
Although advanced Myers lists achieve lookups in fewer pointer traversals, they have limitations. Firstly, advanced Myers lists are significantly more complex and harder to implement. Secondly, the time to determine the shortest path from one node to another is greater, although the traversal is shorter. Thus, advanced Myers lists may not be appropriate for in-memory lists, but they may be useful for applications where the cost of pointer traversals is high relative to other computations, such as when pointer traversals involve network access. As such, in our discussion of advanced Myers lists, we set aside concerns of time complexity in constructing or choosing paths through these lists, and focus solely on the path length between two cells.

### 5.1 Construction of Advanced Myers Lists

In Section 4, we showed that the costs of both the outward and inward phases of a traversal are approximately $\lg n$ each, bringing the total cost to approximately $2\lg n$. Since $\lg n$ is already the information-theoretic limit, further optimization requires balancing the lengths of the phases so that the overall path length is approximately $\lg n$. Conceptually, the approach we take is to add outgoing pointers (in addition to the `next` and `jump` pointers) to the nodes of the tree structure formed by improved Myers lists. These additional pointers allow us to skip a portion of the outward phase, where the length of the skipped portion depends on the length of the inward phase. Essentially, when the inward phase is long, the additional pointers let us keep the outward phase short; when the inward phase is short, the outward phase is allowed to be long. Balancing the lengths of the two phases in this way allows the length of the entire path to be approximately $\lg n$.

However, adding additional pointers to cells violates our requirement of retaining the cell structure of Myers lists. Thus, we conceptually group some cells together and repurpose their `jump` pointers to serve as the additional pointers. We use the term *logical node* to describe such a group of cells, and for clarity, describe individual cells in logical nodes as *physical cells*.

Logical nodes come in three flavours as shown in Figure 9: (1) *normal nodes* consist of a single physical cell with `next` and `jump` pointers pointing to left and right children as before, (2) *skip nodes* consist of two physical cells `top` and `bottom` with an internal pointer `top.next` that points to `bottom`, and three outgoing pointers `uncle`, `left` and `right`, and (3) *leaf nodes* consist of three



(a) Normal Node

(b) Skip Node

(c) Leaf Node. `greater` pointers point to the last cell of a leaf node.

Fig. 9. Node types. Solid arrows: cell `next` pointers. Dashed, dotted and dashdotted arrows: cell `jump` pointers. Normal box □: normal nodes. Double-bordered box ⧈: skip nodes. Box with rounded corners ▢: leaf nodes. Dotted arrows ⋯>: `inc` pointers. Dashdotted arrows -·->: `uncle` pointers. Dashed arrows coming out of leaf nodes ▢- ->: `greater` pointers.

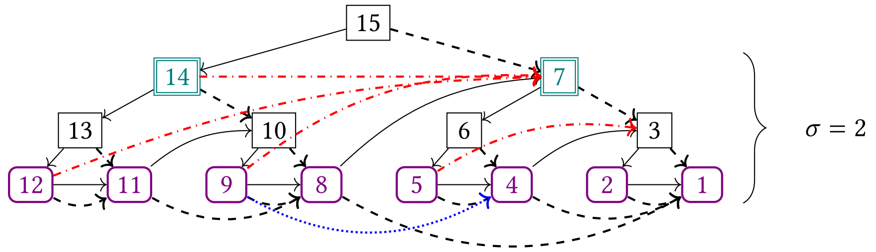


Fig. 10. An example advanced Myers list with 15 logical nodes and 33 physical cells where $\sigma = 2$ with the conventions shown in Figure 9. The internal cell structure of each node is not shown. The `inc` pointer of some leaves point to the same node as its `greater` pointer, such `inc` pointers are not shown. Removing `uncle` and `inc` pointers recovers the structure of the improved Myers list in Figure 8b.

physical cells `top`, `middle` and `bottom` with two internal pointers `top.next` pointing to `middle` and `middle.next` pointing to `bottom`, and four outgoing pointers `inc`, `greater`, `uncle` and `next`.

The construction of the tree formed by an advanced Myers list is the same as that of improved Myers lists, with rules shown in Figure 7. However, the nodes of the tree formed by the list are *logical nodes*, instead of physical cells. The leaf layer of the tree comprises of leaf nodes, and every $\sigma$ layers above the leaf layer are skip layers, comprised of skip nodes. The remaining nodes in the tree are normal nodes.

An example advanced Myers list is shown in Figure 10. Removing `uncle` and `inc` pointers recovers the structure of an improved Myers list of length 15, as shown in Figure 8b. Although the list has 15 logical nodes, since each leaf node has three physical cells and each skip node has two physical cells, this list has a total of 33 physical cells.

In the following subsections, we describe the purpose of skip and leaf nodes and the problems they overcome. We temporarily ignore internal cell pointers and focus on the paths through the logical tree to build intuition behind the tree structure; our analysis of path lengths in Section 5.2 accounts for internal cell pointers.

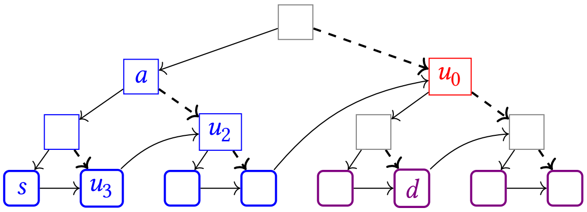


Fig. 11. An example of uncles, target uncles and ancestor subtrees. The uncles of node $s$ are the nodes labelled $u_k$, where each $u_k$ has right-edge depth $k$. $targetUncle(s, d)$ is $u_0$ (red). Nodes in subtree rooted at $a$ form $ancestorSubtree(s, d)$ (blue).

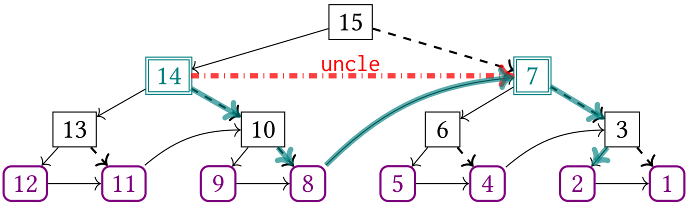


Fig. 12. An example traversal from node 14 to node 2 for the advanced Myers list shown in Figure 10. Some pointers omitted for clarity. Thick arrows show two possible paths for the traversal: (1) without `uncle` pointers, the traversal requires a descent to node 8; (2) with `uncle` pointers, we can jump directly to node 7 from node 14 via its `uncle` pointer.

### 5.1.1 *Double Descent and Skip Layers.*

When traversing from a *src* node to a *dst* node, if *src* is not an ancestor of *dst*, the traversal always passes through the right child of the least common ancestor of *src* and *dst*. We call this the *target uncle* of *src* and *dst* (written as *targetUncle*(*src*, *dst*)). Essentially, the traversal from *src* to *dst* consists of a traversal from *src* to *targetUncle*(*src*, *dst*), then a descent to *dst*. An example of the target uncle is shown in Figure 11, where the target uncle of $s$ and $d$ is $u_0$. As another example, in Figure 12, the target uncle of nodes 14 and 2 is node 7.

In an improved Myers list, the traversal from *src* to *targetUncle*(*src*, *dst*) requires a descent to the leaf layer. For example, in Figure 12, if the list were an improved Myers list, to get from cell 14 to cell 2, we must first descend to cell 8 which is the rightmost leaf of the left subtree, take the `next` pointer to cell 7 (the target uncle), then descend to cell 2. This path descends the height of the tree twice.

The `uncle` pointer of skip nodes and leaf nodes overcomes this hurdle. The `uncle` pointer of a node $n$ points to one of the *uncles* of $n$, where uncles are defined in Definition 5.1. We want to set up `uncle` pointers in a way such that as we descend from *src*, we can always choose a skip node or leaf node $s$ that is a nearby descendant of *src*, and that the `uncle` pointer of $s$ points to the target uncle. Once we have descended to $s$, we then take its `uncle` pointer directly to *targetUncle*(*src*, *dst*), effectively skipping a portion of the descent from *src*.

*Definition 5.1 (Uncle).* A node $u$ is an *uncle* of a node $n$ if $u$ is the right sibling of $n$ or the right sibling of some ancestor of $n$.

The way we set up `uncle` pointers first requires the observation that the length of the inward phase depends on the depth of the target uncle. That is, if the target uncle is $d$ layers away from the root of a tree with height $k$, then the length of the inward phase is no longer than $k - d$. Since the depth of a node depends on the number of elements that have been *cons*-ed to the front, we instead formalize this in terms of *right-edge depth*, as defined in Definition 5.2.

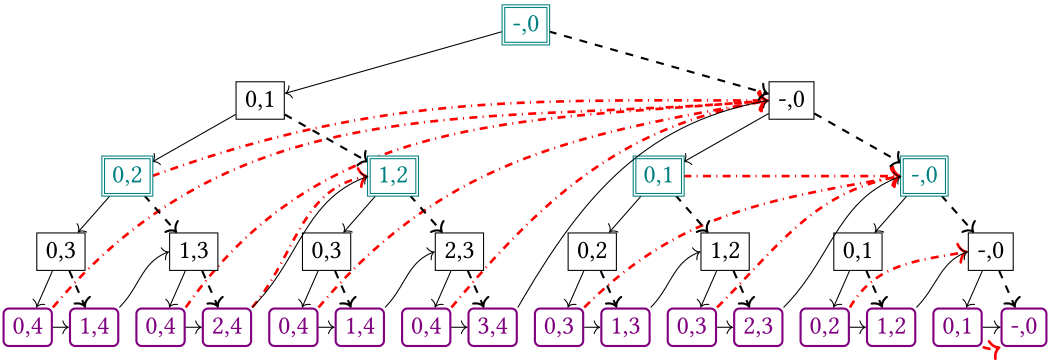


Fig. 13. Example tree with $\sigma = 2$ with each node of right-handed depth $x$ and right-edge depth $y$ is labeled $x, y$. Nodes with undefined right-handed depths are labeled $-, y$. `inc` and `greater` pointers omitted for clarity.

*Definition 5.2 (Right-Edge Depth).* The *right-edge depth* (RED) of a node is the distance to the closest ancestor that is along the rightmost path from the tree root.

The RED of a node does not change as the tree grows, and underestimates its true depth. Therefore, if the target uncle has RED $d$, then the descent from the target uncle is no longer than $k - d$. Then, in order to keep the overall path length within approximately $k$, the outward phase must be bounded by approximately $d$. Thus, we ensure that by descending approximately $d$ layers from *src*, we can get to a skip node or leaf node that points to the target uncle. To do so, we introduce the additional notion of *right-handed depth*, defined in Definition 5.3.

*Definition 5.3 (Right-Handed Depth).* The *right-handed depth* (RHD) of a node is the distance to the closest ancestor that is a left-hand child. For nodes along the rightmost path from the tree root, RHD is undefined.

Our trick is then to let the `uncle` pointer of a node of RHD $d$ point to its uncle with RED $d$. In other words, for every skip node or leaf node $s$, if $s$.`uncle` exists:

$$\text{RHD}(s) = \text{RED}(s.\texttt{uncle})$$

This works because the `uncle` pointer of descendants of *src* whose RHD is $d$ will always point to *targetUncle*(*src*, *dst*). This is shown in Lemma 5.4, which we prove in the supplementary material to save space.

Lemma 5.4. *Given logical nodes src and dst such that dst is not a descendant of src, for all descendants s of src that is a skip node or leaf node, if RHD*$(s)$ = *RED*(*targetUncle*(*src*, *dst*)) *then* $s$.`uncle` *points to targetUncle*(*src*, *dst*).

Therefore, to traverse from *src* to *dst*, (1) descend to a skip node or leaf node $s$ with RHD $d$, then (2) take the `uncle` pointer to *targetUncle*(*src*, *dst*), then finally (3) descend to *dst*. As we will later show in Section 5.2, the length of the descent to $s$ is no longer than approximately $d + \sigma$. This means that the length of the entire path is approximately $k + \sigma + 1$, where $k$ is the height of the tree.

An illustration of the right-handed depths, right-edge depths and uncle pointers of each node in a tree where $\sigma = 2$ is shown in Figure 13. The `uncle` pointer of a node of RHD $d$ points to its uncle with RED $d$. In Figure 10 and Figure 12, the `uncle` pointer of node 14 (of RHD 0) points to node 7 (of RED 0).

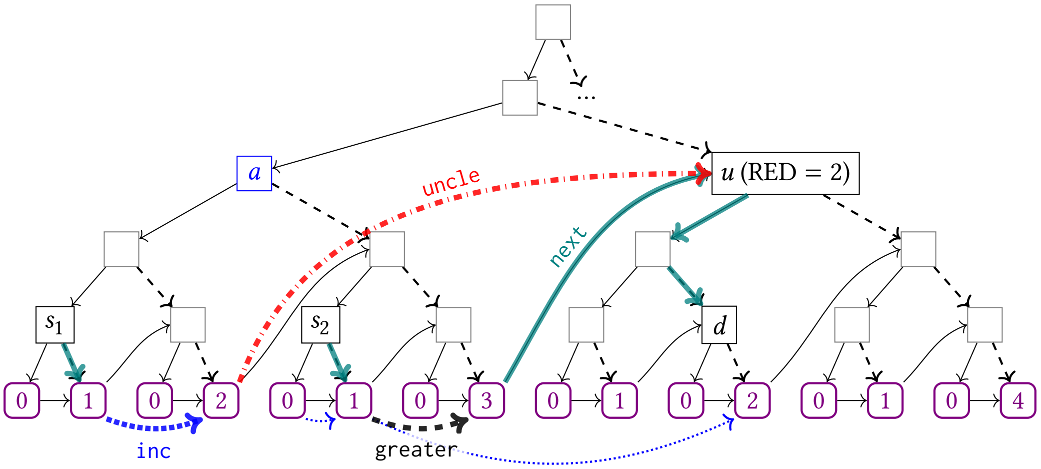


Fig. 14. Two example traversals via lateral traversals across leaves, one from $s_1$ to $d$ (Section 5.1.2), another from $s_2$ to $d$ (Section 5.1.3). Some parts of the tree structure omitted for clarity. RHD of each leaf shown. Node $a$ (blue), is the root of both $ancestorSubtree(s_1, d)$ and $ancestorSubtree(s_2, d)$. Thick arrows show the path of the traversals.

An example traversal via `uncle` pointers is shown in Figure 12. By introducing `uncle` pointers, the traversal from node 14 to node 2 no longer requires a descent to the leaf layer. Since the RED of node 7 is 0, we just descend to a skip node or leaf node whose RHD is 0. Node 14 is itself a skip node with RHD 0, so we take its `uncle` pointer directly to node 7, from which we descend to node 2 like before. (Note that the `uncle` pointer does not always point to a node at the same layer. That is just a coincidence of this example.)

*5.1.2 Falling Out of the Tree.* The method described in Section 5.1.1 does not work if there is no descendant skip node of the desired RHD. This happens when *src* is located too low in the tree or if RED(*targetUncle*(*src*, *dst*)) is too large. This can be seen in Figure 14 where a skip/leaf node of RHD 2 is needed to get from $s_1$ or $s_2$ to $d$, but only leaf nodes of RHD 0 and 1 exist beneath $s_1$ and $s_2$.

The way we address this is to have the tree support lateral traversal across leaves via `inc` pointers. The `inc` pointer of a leaf node $\ell$ points to the closest leaf node $\ell'$ to the right of $\ell$ such that $\mathrm{RHD}(\ell') = \mathrm{RHD}(\ell) + 1$. This is so that from *src*, we can descend to a leaf node $\ell$ where $\mathrm{RHD}(\ell) < \mathrm{RED}(targetUncle(src, dst))$, traverse laterally via `inc` pointers to a leaf $\ell^*$ such that $\mathrm{RHD}(\ell^*) = \mathrm{RED}(targetUncle(src, dst))$, then take its `uncle` pointer to *targetUncle*(*src*, *dst*).

However, traversing laterally using `inc` pointers may take us out of the subtree rooted at *src*. Thus we must ask: *will* $\ell^*$`.uncle` *point to targetUncle*(*src*, *dst*)? The answer is yes, as long as $\ell^*$ is in the *ancestor subtree* of *src* and *dst*, where ancestor subtrees are defined in Definition 5.5.

*Definition 5.5 (Ancestor Subtree).* Let *src* and *dst* be two nodes such that *dst* is not a descendant of *src*. The *ancestor subtree* of *src* and *dst*, denoted *ancestorSubtree*(*src*, *dst*), is the tree rooted at $a$ where $a$ is the ancestor of *src* (or is *src* itself) such that *targetUncle*(*src*, *dst*) is its right sibling.

For example, in Figure 11, $ancestorSubtree(s, d)$ is the tree rooted at $a$, because the right sibling of $a$ is $targetUncle(s, d) = u_0$. Similarly, in Figure 14, $ancestorSubtree(s_1, d)$ and $ancestorSubtree(s_2, d)$ are both rooted at node $a$, since its right sibling is $u$.

The reason this works is that the `uncle` pointer of every leaf in $\mathit{ancestorSubtree}(\mathit{src}, \mathit{dst})$ whose RHD is equal to $\mathrm{RED}(\mathit{targetUncle}(\mathit{src}, \mathit{dst}))$ points to $\mathit{targetUncle}(\mathit{src}, \mathit{dst})$. This is stated as Lemma 5.6, which we prove in the supplementary material to save space.

Lemma 5.6. *Suppose we are given logical nodes src and dst in an advanced Myers list such that dst is not a descendant of src. Let* $d = \mathit{RED}(\mathit{targetUncle}(\mathit{src}, \mathit{dst}))$. *For all skip nodes or leaf nodes s in* $\mathit{ancestorSubtree}(\mathit{src}, \mathit{dst})$, *if* $\mathit{RHD}(s) = d$, *then* `s.uncle` *points to* $\mathit{targetUncle}(\mathit{src}, \mathit{dst})$.

Therefore, if there is no descendant of *src* that has RHD $d$ and is a skip or leaf node, as long as there is a descendant of *src* that is a leaf $\ell$ of RHD less than $d$, we can descend to $\ell$, then move laterally with `inc` pointers until we arrive at $\ell^*$ of RHD $d$. As long as $\ell^*$ is within the ancestor subtree, its `uncle` pointer brings us to $\mathit{targetUncle}(\mathit{src}, \mathit{dst})$. As we show in Section 5.2, the length of the path from *src* to $\ell^*$ is approximately $d$, thus the total path length in this case is approximately $k$.

An example of a traversal via `inc` pointers is shown in Figure 14. To get from $s_1$ to $d$, descend to the leaf with RHD 1 and take the `inc` pointer to the closest leaf with RHD 2. Since this leaf is still within $\mathit{ancestorSubtree}(s_1, d)$, taking its `uncle` pointer brings us to $u$, from which we can descend to $d$ as per usual.

### 5.1.3 *Swinging Out of the Tree.*

The final problem concerns cases where traversing `inc` pointers takes us out of the ancestor subtree. An example of this is shown in Figure 14. In this example, following `inc` pointers from the leaves of $s_2$ would lead us to a leaf of RHD 2 that is a descendant of $d$, from which $d$ is unreachable. Addressing this requires one further observation: because of how the nodes of a tree are ordered, the target uncle is always the next node after the rightmost leaf of the ancestor subtree. In this case, rather than following `inc` pointers to a leaf node of the appropriate RHD, we instead want to reach the rightmost leaf of the ancestor subtree, then take its `next` pointer to the target uncle.

We traverse to the rightmost leaf of the ancestor subtree by using `greater` pointers. The `greater` pointer of a leaf $\ell$ points to the nearest leaf node $\ell'$ to the right of $\ell$ such that $\mathrm{RHD}(\ell') > \mathrm{RHD}(\ell)$ (rather than being greater by exactly 1). Since the rightmost leaf node of any subtree has the highest RHD among the nodes within it, following `greater` pointers will always lead to it[3]. In fact, when traversing laterally with `inc` pointers, if we arrive at a leaf $\ell'$ whose `inc` pointer takes us out of the ancestor subtree, either $\ell'$ itself or $\ell'$`.greater` is the rightmost leaf of the ancestor subtree. This is stated as Lemma 5.7, which we prove in the supplementary material to save space.

Lemma 5.7. *Suppose we have a node n and some leaf* $\ell$ *that is a descendant of n, and* $\mathit{RED}(n) > 0$. *If* $\ell$`.inc` *is not a descendant of n or does not exist, then* $\ell$ *or* $\ell$`.greater` *is the rightmost leaf descendant of n.*

Thus, whenever lateral traversal across leaves is demanded, we traverse laterally via `inc` pointers until we either arrive at a leaf whose `uncle` pointer points to the target uncle, or arrive at a leaf whose `inc` pointer escapes the ancestor subtree. In the latter case, we traverse to the rightmost leaf of the ancestor subtree via the `greater` pointer (if we are not already there), then take the `next` pointer to the target uncle. As we show in Section 5.2, in either case, we can arrive at the target uncle in approximately $d$ pointers. Hence, the overall path length for either case is approximately $k$.

For example, in Figure 14, after descending from $s_2$ to its leaf descendant with RHD 1, instead of taking the `inc` pointer to the leaf descendant of $d$ with RHD 2, we take the `greater` pointer to the rightmost leaf descendant of $a$, from which taking the `next` pointer brings us to $u$. Descending from $u$ brings us to $d$, as per usual.

[3] `greater` pointers correspond to the `jump` pointers of leaves in improved Myers lists, which always traverse to the rightmost leaf of the next enclosing subtree.

*5.1.4 Summary.* We now have all the components necessary to describe the structure of the tree formed by an advanced Myers list:

1. Logical nodes form a binary tree.
2. The leaf layer is comprised of leaf nodes.
3. Every $\sigma$ layers above the leaf layer are skip layers, comprised of skip nodes.
4. For every skip/leaf node $s$, let $d = \text{RHD}(s)$, then `s.uncle` points to the uncle of $s$ whose RED is also $d$.
5. The `inc` pointer of a leaf node $\ell$ points to the closest leaf node $\ell'$ to the right of $\ell$ such that $\text{RHD}(\ell') = \text{RHD}(\ell) + 1$.
6. The `greater` pointer of a leaf $\ell$ points to the nearest leaf node $\ell'$ to the right of $\ell$ such that $\text{RHD}(\ell') > \text{RHD}(\ell)$.

Paths from *src* to *dst* take one of the following forms:

1. *dst* is a descendant of *src*. The path in this case is a descent down the tree.
2. The previous case does not hold, but there exists a descendant skip node or leaf node $s$ whose RHD is equal to the RED of the target uncle (let this be $d$). The path is a descent to $s$, the `uncle` pointer to the target uncle, then a descent to *dst*.
3. The previous case does not hold, but there is a leaf descendant of *src* whose RHD is less than the RED of the target uncle. The path is a descent to the leaf layer, followed by a series `inc` pointers until either (a) we arrive at a leaf within the ancestor subtree that has RHD $d$, in which case we take the `uncle` pointer to the target uncle, or (b) we arrive at a leaf whose `inc` pointer takes us out of the ancestor subtree, in which case we take the `greater` pointer if we are not already at the rightmost leaf of the ancestor subtree, then the `next` pointer to the target uncle. From the target uncle, we descend to *dst*.
4. The previous cases do not hold, so *src* is a leaf whose RHD exceeds $d$. Traversing the `next` pointer brings us to a non-leaf node where one of the previous three cases must apply.

For a logical tree of height $k$, the upper bound of the length of the inward and outward phases of the traversal is $k - d$ and $d + \sigma + O(1)$, respectively, where $d$ is the RED of the target uncle. Thus, the length of the traversal is no greater than $k + \sigma + O(1)$.

## 5.2 Analysis of Lookups in Advanced Myers Lists

As with Section 3.4 and Section 4.3, we analyze the number of pointers traversed when performing lookups in an advanced Myers list. The main result of this subsection is Theorem 5.8. We must warn readers that the analysis is incredibly dry.

Theorem 5.8. *Given an advanced Myers list $L$ of length $n$ that contains arbitrary cells* `src` *and* `dst` *such that* `src.length` $\geq$ `dst.length` *and every $\sigma$ layers of the logical tree formed by $L$ is a skip layer, the number of pointers traversed to get from* `src` *to* `dst` *is no greater than* $(1 + 1/\sigma)\lfloor \lg n \rfloor + \sigma + 9$.

Just like Section 3.4 and Section 4.3, the first step is to relate $\sigma$, the height of the logical tree formed by the list, and the number of physical cells in the list in Lemma 5.9, which we prove in the supplementary material to save space.

Lemma 5.9. *Given an advanced Myers list $L$ of length $n > 1$, for all $\sigma \geq 1$, the smallest advanced Myers list $L'$ that contains $L$ as a suffix and forms a logical tree of height $k$ satisfies $k \leq \lfloor \lg n \rfloor - 1$.*

*5.2.1 Preliminaries for Analysis of Lookups in Advanced Myers Lists.* For the rest of this subsection, assume that we are working with an advanced Myers list $L$ forming a logical tree of height $k$, such that every $\sigma$ layers above the leaf layer is a skip layer. In addition, the traversal of concern is between two cells `src` and `dst` in logical nodes *src* and *dst* respectively, and that `src.length` $\geq$ `dst.length`.

Unlike Section 5.1, our analysis in this section must account for internal cell pointers. Thus, to distinguish paths in the *logical tree* (ignoring internal cell pointers) from the actual path via cell pointers (including internal cell pointers), we use the term *logical path* to denote the former. The length of the logical path is termed *logical distance*.

*5.2.2 On Direct Descents.* The first case is when *src* is a logical tree ancestor of *dst*. The path length is shown in Lemma 5.10, which is the main result of this part of the analysis.

Lemma 5.10. *If dst is a descendant of src, the path from* `src` *to* `dst` *has no more than* $(1+(1/\sigma))k+2$ *pointers.*

To give an upper bound of the path length in this case, we show the upper bound of the path length of the general case (Lemma 5.11), i.e., a descent from any node of RED $d$ to any descendant.

Lemma 5.11. *Suppose we have cells* `s` *and* `s'` *in logical nodes $s$ and $s'$ respectively, where $s'$ is a descendant of $s$. Let $d = RED(s)$. The path from* `s` *to* `s'` *consists of no more than* $(1 + 1/\sigma)k - (1 + 1/\sigma)d + 2$ *pointers.*

Proof. Let $depth(s)$ be the logical distance from the tree root to $s$. The height of $s$ is $k - depth(s)$. Thus, the logical distance from $s$ to any of its descendants is not more than $k - depth(s)$. Since RED underestimates depth, i.e., $d \leq depth(s)$, the logical distance from $s$ to any of its descendants is no more than $k - d$. This logical path passes through up to $(k - d)/\sigma + 1$ skip nodes and potentially one leaf node, which adds up to $(k - d)/\sigma + 2$ internal cell pointers. Thus, in total, the path consists of no more than $k - d + (k - d)/\sigma + 2$ pointers. □

Then, Lemma 5.10 is a simple consequence of Lemma 5.11, because in the worst case, RED$(src) = 0$. Therefore, the path from `src` to `dst` has no more than $(1+1/\sigma)k-(1+1/\sigma)(0)+2 = (1+(1/\sigma))k+2$ pointers.

*5.2.3 On Traversals Through Descendant Skip Nodes.* For the rest of the analysis, assume that *src* is not an ancestor of *dst* in the logical tree. Thus, the path from `src` to `dst` involves traversing from `src` to *targetUncle*(*src*, *dst*) then to `dst`. This part of the analysis gives an upper bound of the length of the path for the scenario described in Section 5.1.1—where there exists a skip or leaf node that points to *targetUncle*(*src*, *dst*) and is a descendant of `src`. The main result of this part of the analysis is Lemma 5.12.

Lemma 5.12. *Let $d = RED(targetUncle(src, dst))$. If there exists a logical descendant of src that is a skip node or leaf node with RHD d whose* `uncle` *pointer is reachable from* `src`*, then the path from* `src` *to* `dst` *has no more than* $(1 + 1/\sigma)k + \sigma + 6$ *pointers.*

First, we prove Lemma 5.13, which shows showing the existence of descendants of nodes with specific right-handed depths as long as the node is sufficiently high up in the tree.

Lemma 5.13. *Given logical nodes $s$ and $s'$, if (1) $s'$ is a logical descendant of $s$ and (2) the logical distance from $s$ to $s'$ is $d$ where $d > 0$, then for all $0 \leq i < d$, there exists node $s_i$ such that (1) $s_i$ is a descendant of $s$, (2) the logical distance from $s$ to $s_i$ is also $d$ and (3) $s_i$ has right-handed depth $i$.*

Proof. From $s$, take $d - i$ `left` pointers to a node $x$. $d - i$ is nonzero, guaranteeing that $x$ has right-handed depth 0. From $x$, take $i$ `right` pointers to $s_i$ with right-handed depth $i$. □

Using Lemma 5.13, we prove Lemma 5.14, which gives an upper bound for the number of pointers needed to traverse to a descendant skip or leaf node of the appropriate RHD.

Lemma 5.14. *Given a cell* `s` *in logical node $s$, if there exists a skip or leaf node $s'$ where (1) let $d = RHD(s')$, (2) $s'$ is a descendant of $s$, (3) the cell containing the* `uncle` *pointer in $s'$ is reachable from* `s`*, then there exists a skip or leaf node $s^*$ such that (1) $RHD(s^*) = d$, (2) $s^*$ is a descendant of $s$, (3) the cell containing the* `uncle` *pointer in $s^*$ is reachable from* `s` *and (4) the traversal from* `s` *to the cell containing the* `uncle` *pointer in $s^*$ has no more than $(1 + 1/\sigma)d + \sigma + 3$ pointers.*

Proof. We proceed by case analysis.

*Case* 1. The logical distance from $s$ to $s'$ is no greater than $d + \sigma$. Then, let $s^* = s'$. Let the cell containing the `uncle` pointer in $s'$ be `s'`. The path from `s` to `s'` passes through up to $d/\sigma + 2$ skip/leaf nodes. If $s'$ is a leaf node then we must add one more pointer to get to `s'`. Therefore, the total number of pointers traversed to get from `s` to `s'` is no more than $d + \sigma + d/\sigma + 3$.

*Case* 2. The logical distance from $s$ to $s'$ is greater than $d + \sigma$. This means that the logical height of $s$ is at least $d + \sigma + 1$. This must also mean that there exists $0 \leq b < \sigma$ where $b$ layers below $s$ is a skip or leaf layer, and any $a\sigma + b$ layers below $s$ is a skip or leaf layer as long as $a \geq 0$ and $a\sigma + b \leq d + \sigma + 1$. Now, choose some $a' \geq 0$ satisfying $d < a'\sigma + b \leq d + \sigma$. By Lemma 5.13, there exists a node $s^*$ that has right-handed depth $d$ that is $a'\sigma + b$ layers below $s$ (therefore a skip or leaf node) and is a descendant of $s$. The logical distance from $s$ to $s^*$ is thus no greater than $d + \sigma$. With similar arguments to Case 1, the traversal from `s` to the cell containing the `uncle` pointer in $s^*$ has no more than $(1 + 1/\sigma)d + \sigma + 3$ pointers.

□

Finally, to prove Lemma 5.12, by Lemma 5.14, we can traverse no more than $(1 + 1/\sigma)d + \sigma + 3$ pointers from `src` to arrive at the cell in some skip or leaf node $s$ with RHD $d$ containing an `uncle` pointer. Then, we take the `uncle` pointer, which by Lemma 5.4 brings us to *targetUncle*(*src*, *dst*). Lastly, by Lemma 5.11 we traverse $(1 + 1/\sigma)k - (1 + 1/\sigma)d + 2$ pointers to arrive at `dst`. The total path length is no greater than $(1 + 1/\sigma)k + \sigma + 6$.

*5.2.4 On Lateral Traversals via Leaf Nodes.* For the rest of the analysis, further assume that there is no descendant of *src* that (1) is a skip/leaf node, (2) has RHD equal to RED(*targetUncle*(*src*, *dst*)) and (3) whose `uncle` pointer is reachable from `src`. In this case, as described in Section 5.1.2 and Section 5.1.3, we could descend to a leaf node whose RHD is less than the RED of *targetUncle*(*src*, *dst*), traverse laterally to the right leaf node, and go to *targetUncle*(*src*, *dst*) to descend from there as per usual. The main result in this part of the analysis is Lemma 5.15.

Lemma 5.15. *Let $d = RED(targetUncle(src, dst))$. If there exists a leaf node that (1) is a descendant of src, (2) whose* `inc` *pointer is reachable from* `src` *and (3) has RHD less than d, then the path from* `src` *to* `dst` *has no more than $(1 + 1/\sigma)k + 7$ pointers.*

We first prove Lemma 5.16, which gives an upper bound of the height of *src*.

Lemma 5.16. *Given a logical node s, if there does not exist a descendant skip node or leaf node of right-handed depth d whose* `uncle` *pointer is reachable from s, then, the logical distance from s to any leaf node that is a descendant of s is no more than d.*

Proof. If the logical distance from $s$ to any leaf node that is a descendant of $s$ is at least $d + 1$, then by Lemma 5.13, there must be a descendant leaf node of RHD $d$. The cell containing the `uncle` pointer in a leaf node is the last cell of the leaf, thus always reachable. □

Then to prove Lemma 5.15, we first consider the cases of the relationship between *src* and the leaf layer to determine the starting leaf $\ell$.

*Case* 1. *src* is a leaf, and `src` is the first cell of *src*. Let $\ell = src$. The path from `src` to the first cell of $\ell$ has 0 pointers.

*Case* 2. *src* is not a leaf. Thus, the height of *src*, $h$, must be greater than 0. This also means that $h \leq d$, otherwise, by Lemma 5.13, there must be a leaf descendant of *src* whose RHD is equal to $d$, contradicting the conditions of this case.
From *src*, take one `left` pointer, then $h-1$ `right` pointers down to leaf $\ell$. Thus, $\text{RHD}(\ell) = h - 1$, and the logical distance from *src* to $\ell$ is $h = \text{RHD}(\ell) + 1$. This logical path passes through up to $(\text{RHD}(\ell) + 1)/\sigma$ skip nodes, thus no more than $(\text{RHD}(\ell) + 1)/\sigma$ additional internal cell pointers are traversed. In total, the path from `src` to the first cell of $\ell$ has no more than $(1 + 1/\sigma)\text{RHD}(\ell) + 1/\sigma + 1$ pointers.

In either case, the path from `src` to the first cell of $\ell$ has no more than $(1 + 1/\sigma)\text{RHD}(\ell) + 1/\sigma + 1$ pointers.

Clearly, $\ell$ is within the ancestor subtree. Thus, one of two cases must happen when traversing from $\ell$ to *targetUncle*(*src*, *dst*):

*Case* 1. Traversing $d-\text{RHD}(\ell)$ `inc` pointers brings us to a leaf node $\ell^*$ of right-handed depth $d$ that is still within *ancestorSubtree*(*src*, *dst*). From $\ell^*$ we traverse two internal pointers to reach the last cell of $\ell^*$, then by Lemma 5.6, the `uncle` pointer takes us to *targetUncle*(*src*, *dst*). The path from $\ell$ to *targetUncle*(*src*, *dst*) has $d - \text{RHD}(\ell) + 3$ pointers.

*Case* 2. Traversing some $0 \leq x < d - \text{RHD}(\ell)$ `inc` pointers brings us to a leaf node $\ell_r$ such that $\ell_r$`.inc` is either not within *ancestorSubtree*(*src*, *dst*) or does not exist. The root of *ancestorSubtree*(*src*, *dst*) is always a left child and thus never has RED 0, therefore, by Lemma 5.7, $\ell_r$ or $\ell_r$`.greater` is the rightmost leaf of *ancestorSubtree*(*src*, *dst*). Thus, from $\ell_r$, take one pointer to the middle cell of $\ell$, take one pointer (either `greater` or the internal `next` pointer) to the last cell of the rightmost leaf of *ancestorSubtree*(*src*, *dst*), and take the `next` pointer to *targetUncle*(*src*, *dst*). The path from $\ell$ to *targetUncle*(*src*, *dst*) has $x + 3 < d - \text{RHD}(\ell) + 3$ pointers.

In either case, the path from $\ell$ to *targetUncle*(*src*, *dst*) has no more than $d - \text{RHD}(\ell) + 3$ pointers.

By Lemma 5.11 we can descend from *targetUncle*(*src*, *dst*) to any cell in *dst* in no more than $(1 + 1/\sigma)k - (1 + 1/\sigma)d + 2$ pointers. Thus, overall, the path has no more than $(1 + 1/\sigma)\text{RHD}(\ell) + 1/\sigma + 1 + d - \text{RHD}(\ell) + 3 + (1 + 1/\sigma)k - (1 + 1/\sigma)d + 2 \leq (1 + 1/\sigma)k + 7$ pointers.

*5.2.5 The Length of All Possible Paths.* With the preceding results, we can prove Lemma 5.17, which gives an upper bound for path length of the traversal from `src` to `dst`.

Lemma 5.17. *The path from* `src` *to* `dst` *has no more than* $(1 + 1/\sigma)k + \sigma + 10$ *pointers.*

Proof. Logical paths from *src* to *dst* belong to one of six cases:

*Case* 1. *dst* is a descendant of *src*. By Lemma 5.10, the path from `src` to `dst` has no more than $(1 + (1/\sigma))k + 2$ pointers.

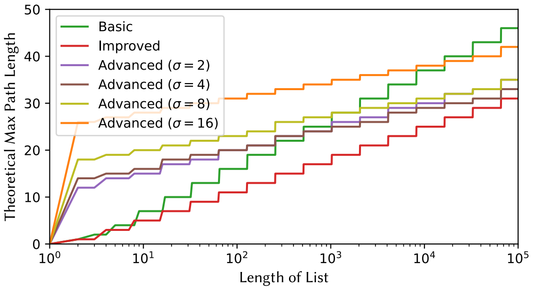
*Case* 2. Case 1 does not apply, but there exists a skip/leaf node $s$ that (1) is a logical descendant of *src*, (2) $\text{RHD}(s) = \text{RED}(targetUncle(src, dst))$, and (3) whose `uncle` pointer is reachable from `src`. By Lemma 5.12, the path from `src` to `dst` has no more than $(1+(1/\sigma))k+\sigma+6$ pointers.

*Case* 3. Cases 1 and 2 do not apply, but there exists a leaf node with (1) RHD less than RED(*targetUncle*(*src*, *dst*)), (2) is a logical descendant of *src*, and (3) its `inc` pointer is reachable from `src`. By Lemma 5.15, the path from `src` to `dst` has no more than $(1 + (1/\sigma))k + 7$ pointers.

*Case* 4. Cases 1, 2 and 3 do not apply, but *src* is a leaf node whose RHD exceeds RED(*targetUncle*(*src*, *dst*)). Taking up to three cell `next` pointers reaches a non-leaf node,

| Variant | Upper Bound |
|---|---|
| Basic | $\begin{cases}\lceil \lg n \rceil & \text{if } 1 \leq n \leq 4 \\ 3\lceil \lg n \rceil - 5 & \text{otherwise}\end{cases}$ |
| Improved | $\begin{cases}\lceil \lg(n+1) \rceil - 1 & \text{if } n = 1 \\ 2\lceil \lg(n+1) \rceil - 3 & \text{otherwise}\end{cases}$ |
| Advanced | $\left\lfloor (1+\frac{1}{\sigma}) \lfloor \lg n \rfloor \right\rfloor + \sigma + 9$ |

(a) Summary of theoretical worst-case path length given list of length $n$.



(b) Length of list vs theoretical worst-case path length. $x$-axis in log scale.

Fig. 15. Comparison of theoretical worst-case path lengths for each variant of Myers list.

and thus either cases 1, 2 or 3 must apply. This gives us a total of no more than $(1 + (1/\sigma))k + \sigma + 9$ pointers.

*Case* 5. Cases 1, 2, 3 and 4 do not apply, but *src* is a leaf node whose right-handed depth is not 0. This means that `src` is the second or third cell in *src*. Similar to Case 4, we take up to two pointers to reach a non-leaf node, and thus either cases 1, 2 or 3 must apply. This gives us a total of no more than $(1 + (1/\sigma))k + \sigma + 8$ pointers.

*Case* 6. Cases 1, 2, 3, 4 and 5 do not apply. This means that `src` is the second or third cell of *src* and that *src* is a leaf node of right-handed depth 0. Let the logical next node of *src* be *s*. *s* is necessarily the right sibling of *src* and is thus a leaf node of nonzero RHD.

If the first cell of *s* satisfies cases 1, 2 or 3, then similar to Case 4 and Case 5 we take up to two pointers from `src` to reach the first cell of *s*, and so the overall path has no more than $(1 + (1/\sigma))k + \sigma + 8$.

Otherwise, the first cell of *s* must satisfy case 4, and thus *s*`.next` satisfies cases 1, 2 or 3. If `src` is the second cell of *src*, taking the `greater` pointer brings us to the last cell of *s*, then the `next` pointer brings us to the first cell of *s*`.next` for a total of $(1 + (1/\sigma))k + \sigma + 8$ pointers. Otherwise, `src` is the last cell of *src*, and taking four pointers brings us to the first cell of *s*`.next`, and the entire path has no more than $(1 + (1/\sigma))k + \sigma + 10$ pointers.

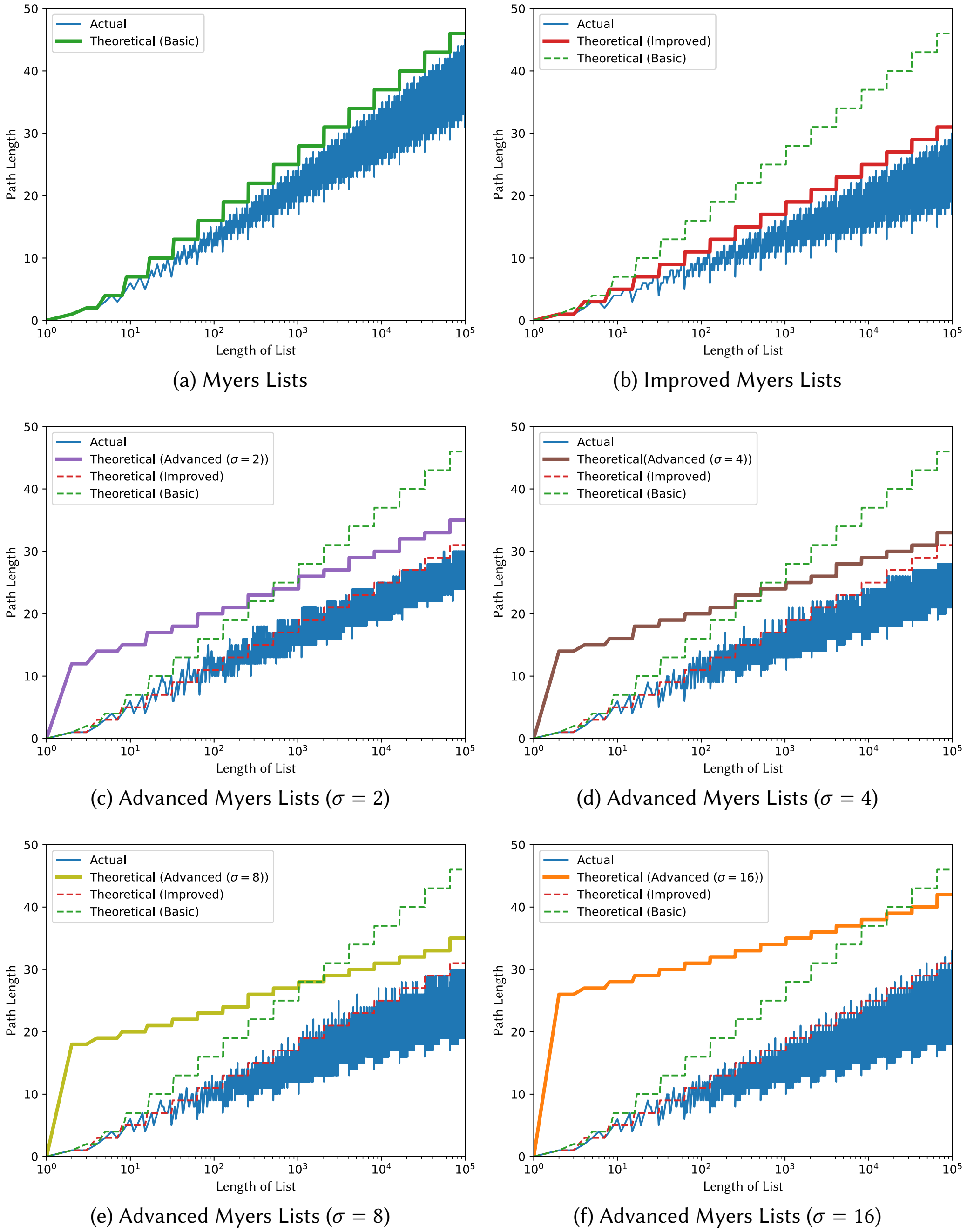
Therefore, the path length is no more than the maximum lengths of these cases, which is $(1 + (1/\sigma))k + \sigma + 10$. □

Finally, Theorem 5.8 is a simple consequence of Lemma 5.9 and Lemma 5.17.

## 6 Empirical Evaluation

In this section, we compare the theoretical and actual path lengths for lookups in the variants of Myers lists presented in this paper. The variants considered are basic Myers lists (Section 3), improved Myers lists (Section 4) and advanced Myers lists (Section 5) with $\sigma = 2, 4, 8, 16$. We evaluate lists of length up to 100,000—the time taken to perform our evaluation of significantly larger lists (e.g., $10^5 < n \leq 10^6$) is prohibitively long. Figure 15 summarizes the theoretical results.

To see whether our theoretical analysis is tight, and as a check on our worst case analysis, we also directly measured the worst-case path length of traversals in each Myers list variant. The results are plotted in Figure 16. In each plot, the $x$-axis is the length of the list, and the $y$-axis represents path lengths. In blue, we show the actual worst-case path length for lookups starting from the *head* of the list—the worst-case path length of lookups starting from an *arbitrary* cell is



(a) Myers Lists

(b) Improved Myers Lists

(c) Advanced Myers Lists ($\sigma = 2$)

(d) Advanced Myers Lists ($\sigma = 4$)

(e) Advanced Myers Lists ($\sigma = 8$)

(f) Advanced Myers Lists ($\sigma = 16$)

Fig. 16. Comparison of path lengths of Myers list variations. $x$-axes in log-scale. Blue: actual worst case path length starting from the head of the list. Theoretical worst case path lengths also plotted using the color conventions from Figure 15. Dashed lines are theoretical path lengths of other variants for comparison.

obtained by taking the maximum of the worst case path lengths of all the shorter lists. We also overlay the theoretical bounds plotted in Figure 15b for comparison. We measured path lengths instead of runtime because our focus is on validating the theoretical bounds obtained, and also because as noted in Section 5, the time taken to compute the shortest path for advanced Myers lists is expensive, and our implementations are not optimized for it.

## 6.1 Discussion

*6.1.1 Tightness of Theoretical Upper Bounds.* The theoretical bounds for basic and improved Myers lists are tight, closely matching their actual path lengths. This is expected, as their analyses are straightforward and do not involve significant overheads. The theoretical bounds for advanced Myers lists are less tight, particularly for smaller list lengths. This is due to the constant $\sigma + 9$, which dominates the theoretical cost when the height of the tree formed by the advanced Myers list is small compared to it. As the size of the list increases, the bound slowly tightens, and the actual path length curves and asymptotically tends towards the theoretical bound (this curvature is easier to see by using the improved Myers lists as a reference line).

*6.1.2 Path Lengths and Tree Structure.* Recall that for all variants, when the list forms a perfect binary tree, the path length from the head of the list to any other cell is minimized, relative to lists of similar sizes, and this is manifest in the plots as downward spikes.

The worst path lengths occur for basic and improved Myers lists when the list head is the leftmost leaf of the tree formed by it. For advanced Myers lists, the worst path lengths occur in edge cases, such as when starting at the last cell of a leaf whose RHD is large (relative to the height of the tree) and greater than the RED of the target uncle. In general, edge cases incur an additional but constant cost. But, when the list is small and there are few or no skip layers, the edge cases can cause the traversal to exhibit double descent (Section 5.1.1) as traversing out of the edge case brings us to the root of a large subtree, and since there are few or no descendant skip layers, we must descend to the leaf layer to reach the target uncle. As the number of skip layers in the list increases, the edge cases stop degenerating into double descents. For this reason, the path lengths in edge cases are more pronounced for smaller lists, but become less so as the size of the list increases and more skip layers are present.

*6.1.3 Comparison of Variants.* Among all variants, the advanced Myers list with $\sigma = 4$ demonstrates the best actual path lengths among all lists when the list length is close to 100,000. This is because $\sigma = 4$ strikes an optimal balance between the constant overhead of descending to the next skip layer and the $1/\sigma$ overhead of descents through skip layers. For advanced Myers lists with $\sigma = 8$ and $\sigma = 16$, the actual path length is closer to that of improved Myers lists, as lists of size 100,000 have only one or no skip layers. In these cases, the overhead of descents through skip layers is minimal, but the problem of double descent—similar to that in improved Myers lists—still persists. Smaller advanced Myers lists with $\sigma = 2$ and $\sigma = 4$ have longer path lengths than improved Myers lists, as the path cost savings from using skip layers are small relative to the overall traversal cost. However, as the list size increases, the path cost savings from skip layers outweigh the overhead of descents through skip layers, leading to better path lengths for these variants. This suggests that the use of advanced Myers lists is advantageous (compared to improved Myers lists) only when the lists involved are large. Remarkably, despite basic Myers lists outperforming advanced Myers lists on small sizes in Figure 15, when we measure actual path lengths, all the Myers lists variants outperform basic Myers lists (see Figure 16).

## 7 Related Work

*Random-Access Lists.* Myers lists [23] support logarithmic-time random access by augmenting each cell of a cons list with an additional pointer. Okasaki [24] later introduced a purely functional random-access list structured as a list of perfect binary trees, preserving $O(1)$ time and space for *cons*, *car* and *cdr* while reducing the lookup complexity (starting from the first element of the list) to $2\lg(n+1)+O(1)$. As shown by Okasaki [24], Okasaki lists with tree nodes pointing to left children and right siblings (instead of right children) are isomorphic to Myers lists with redundant pointers removed. Improved Myers list (Section 4) matches the lookup performance of Okasaki lists but can do so starting from arbitrary positions. Advanced Myers lists (Section 5) further outperform both improved Myers lists and Okasaki lists in this regard.

*Skip Lists.* Another list-like data structure supporting $O(\lg n)$ random access is the skip list [27], a probabilistic alternative to balanced binary trees. A skip list is built in layers. The bottom layer is an ordinary cons list. Each higher layer allows some elements to skip across multiple nodes, where an element in layer $i$ appears in layer $i+1$ with some fixed probability $p$. On average, the cost of looking up the $m^{\text{th}}$ element of a skip list with $n$ elements is $\log_{1/p} n + \frac{1-p}{p}\log_{1/p} m + \Theta(1)$ [25, 27], but due to their probabilistic nature, the worst case is in linear time. Deterministic skip lists [22] address this with an upper bound of $2\lg n + \Theta(1)$ for lookups. However, though a single perfectly balanced skip list takes $O(n)$ space, it degenerates when multiple skip lists share the same tail, since the headers of a skip list each have logarithmically many outgoing pointers. On the other hand, since Myers list cells are of constant size with two outgoing pointers, multiple Myers lists that share the same tail only take up space linear in the total number of cells.

*Purely Functional Data Structures with Fast Lookups.* Just like Myers lists, other purely functional data structures supporting fast lookups have also been designed. The one-sided flexible array [15] is implemented as a tree backed by a base array, with different base array implementations offering space/time tradeoffs. Optimizing the array for lookup performance gives $\Theta(\sqrt{\lg n})$ lookup complexity. However, in general, the worst case complexity for *cons* is linear in the number of nodes, unlike Myers lists where *cons* is in $O(1)$. VLists [4] are implemented as a linked list of memory blocks of exponentially increasing size. On average, lookups are done in constant time for a single VList, however, in the degenerate case where a VList is constructed by *cons* on tails, lookups are done in $O(n)$ time like cons lists, while Myers lists achieve the same in $O(\lg n)$. Kaplan and Tarjan [17] observed the algorithmic notion of *recursive slowdown*, introducing a persistent deque that can perform lookups in worst-case $O(\lg d)$ time, where $d$ is the distance to the closest end of the deque. These also support $O(1)$ *cons*, *car* and *cdr*, but go further by also supporting $O(1)$ *concatenation*. However, unlike Myers lists, they do not support traversals from arbitrary positions nor can they be adapted to support range queries.

*Finger Search Trees.* Unsurprisingly, trees are commonly used for their lookup performance—our variants of Myers lists also leverage trees for this purpose. A prominent tree structure that supports fast localized lookups is the finger search tree, first introduced by Guibas et al. [12] as a variant of B-trees [6], which supports access to elements near a "finger" (a reference point in the data structure) in $O(\log d)$ time, where $d$ is the distance from the finger to the target element. Alternative implementations of finger search trees have also been developed [9, 12, 16, 18, 19, 28, 29], and other implementations supporting finger searches from an arbitrary number of fingers have also been proposed [7, 8, 10, 13, 14, 21]. Similar to Myers lists, a traversal in a finger search tree generally involves a traversal to what we call an "uncle", and the extra pointers of treaps [3] serve a similar purpose as `uncle` pointers in advanced Myers lists. Finger search trees are typically optimized only for *search* and not for list operations like *cons* and *cdr*, which are typically in $O(\lg d)$.

## 8 Conclusion

We have explored Myers lists, a purely functional data structure that extends traditional cons lists by adding a `jump` pointer to each cell, enabling efficient random access while preserving the advantages of cons lists, such as tail sharing. Unlike cons lists, Myers lists support lookups in a logarithmic number of pointer traversals, making them suitable for applications requiring efficient access to elements. Notably, the structural characteristics of lookups in Myers lists make them well-suited for range queries.

We revealed that Myers lists possess a recursive structure that is isomorphic to binary trees, allowing us to analyze their path lengths using recurrence equations. This analysis uncovered opportunities for optimization, leading to the development of an improved Myers list that reduces the lookup complexity from $3\lceil\lg n\rceil-5$ to $2\lceil\lg(n+1)\rceil-3$. Improved Myers lists match the lookup path lengths of Okasaki lists but additionally supports lookups starting from arbitrary cells. Building on this, we introduced an advanced variation of Myers lists that further reduces the lookup complexity to $(1 + 1/\sigma)\lfloor\lg n\rfloor + \sigma - 9$, showing that it is possible to get arbitrarily close to the optimal bound by trading a factor in front of $\lg n$ for a larger constant. We achieved these improvements with the same cell structure as basic Myers lists, the only difference being where the `jump` pointer of each cell points to.

## Acknowledgments

This work is supported by the Singapore Ministry of Education under Project Number T1 251RES2422.

## A Artifact

This paper is accompanied by an artifact online at:

- Zenodo: https://zenodo.org/records/15628635 (doi:10.5281/zenodo.15628635), which also contains a virtual machine for reproducing the results in this paper [26].
- GitHub: https://github.com/plilab/paper-limits-myers-lists-artifact/releases/tag/v1.0.0, tag `v1.0.0`, commit `0aad997`.
- Software Heritage: https://archive.softwareheritage.org/swh:1:dir:8ce24c2b5a81f1a93a5c888e1d0ff905bef3f168, SWHID `swh:1:dir:8ce24c2b5a81f1a93a5c888e1d0ff905bef3f168`.

The complete artifact is also embedded in this paper as both QR codes (in Appendix A.1) and copyable text (in Appendix A.2). For each of these, copyable versions of the encoded artifact is also embedded as “alternate text” of bullets placed between guillemets (e.g., `filename` ↦ «•»). Copying the bullet copies the file contents. The guillemets are delimiters to make selecting the text in the bullet easier but are not part of the file contents.

The expected file sizes are:

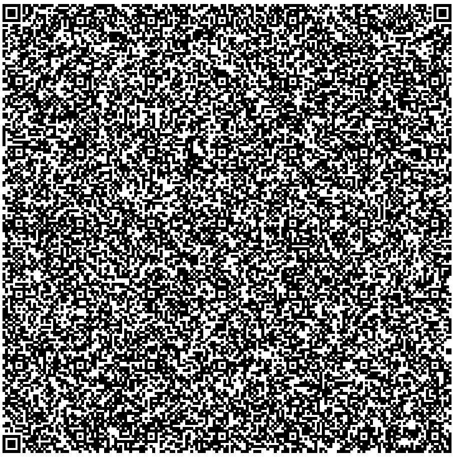
| File | Size |
|---|---|
| `source.tar.gz` | 19067 bytes |
| `source.tar.gz.base45` | 28601 bytes |
| `source.tar.gz.base64` | 25519 bytes |

## A.1 QR-Encoded Artifact File: `source.tar.gz.base45` ↦ «•»

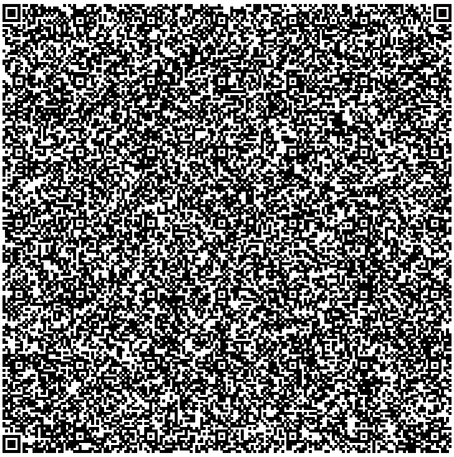
The complete artifact is a binary file, which cannot be copied as alternate text, so it is Base45 encoded[4] and then stored in a sequence of QR codes in case it is not copyable from the bullet. To reconstruct the Base45 encoded artifact, merely concatenate the scans of the QR codes. Each QR code (except the last) contains 4296 bytes. Some Base45 decoders append a spurious newline at the end that causes an "unexpected end of file" error from `gzip`. That newline should be removed with a command like `head --bytes=-1`.

*QR Encoded Artifact:*

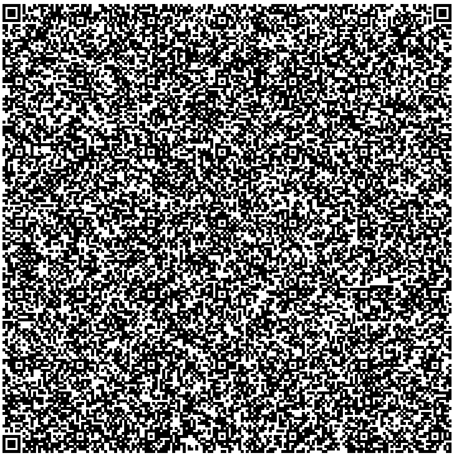
`000` ↦ «•»



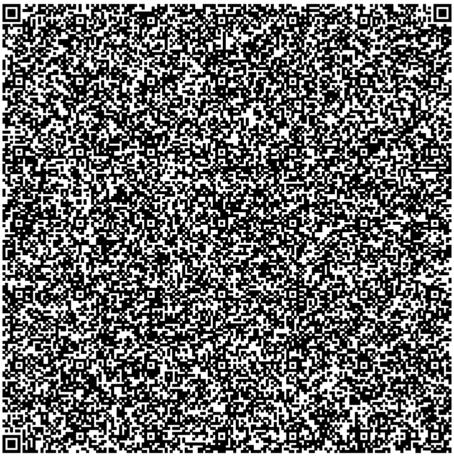
`001` ↦ «•»



`002` ↦ «•»

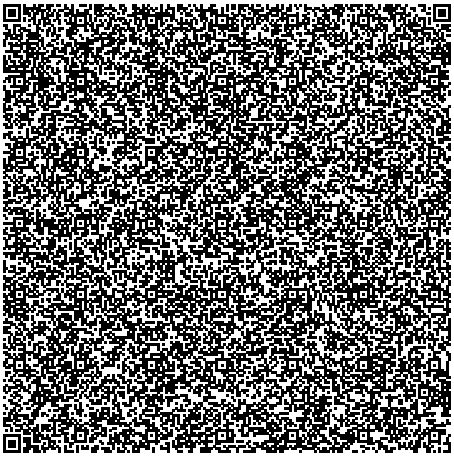


`003` ↦ «•»

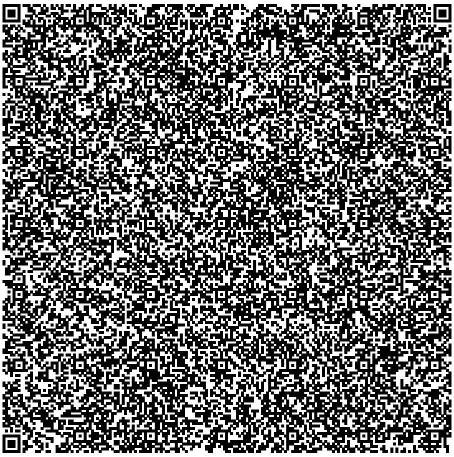


[4] https://www.rfc-editor.org/rfc/rfc9285

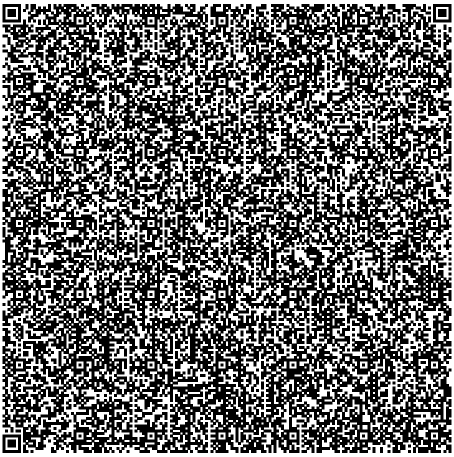
004 ↦ «•»



005 ↦ «•»



006 ↦ «•»



## A.2 Base64-Encoded Artifact File: `source.tar.gz.base64` ↦ «•»

The complete artifact is a binary file, which cannot be copied as alternate text, so it is Base64 encoded[5] and then stored in small-font text in case it is not copyable from the bullet. Newlines should be omitted from the Base64-encoded file. Be careful to not include page numbers, headers or footers when copying.

*Base64 Encoded Artifact:*

H4sIAAAAAAAC/+w8aXPbSK7zNfwVeIzzQia6KMt2RllnozjKRLW+SrY3b8p2yS2xJTHhoWFTtpXrtz+gm6cOx3F2PLu1YY0jEg2g0Wg0gL5GBNNwwKu//JlPDZ+tWk3+1hZ/5bu1UatvrW9sNSR8q7618Qts/HIPz1RELMQqf/nvfITq/93OTnv/qP3n9f/m6v6vW1a92P9WbR1/oPaz///0Z69zDLvOgPuCa9pOMJmFzmgcgTEwoV6rb8BhGIxC5nmOP4Jd5o+mbM
Sh4/vBJYucwEdYH17C/smRph3y0HOEIKgjYMxD3p8BEvsRt0swDDmHYAiDMQtHvARRAMyfwYSHAgmCfsQcnyphMEApNMSMxshGBMPoioUckW1gQgQDhyE/sIPB1ON+pKQYOi4XYERjDvpRTKGbshKbM1dzfKCypAiunGgcTCMIuYhCZ0A8SuD4A3dqkwxJset4TlwDkUvVCA2ZTgW2gOQsgRfYzpB+uWzWZNp3HTEuge0Q6/40QqAgoNRxidpRDUIQ3HU15OCg3LKt
mXQSh0SfkEKjWEWCIFfjwCu2xBHacBr6WCWXNHaAKpM1vueDiCCEPgxcN7iipg0C33aoRaKpacdYxPrBJZdtUT3vBxGKqkSgDphkvRoXiTFzXejzWGFYL6qX5ZoTUvU4rPzIYS5MglDWN9/MCtb/tg1HB2+O37W6begcwWH34J+d1+3XoLeO8FsvwbvO8duDk2NAjG5r//h3OHgDrf3f4R+d/dclaP/fYbd9dAQHXa2zd7jbaSOss7+ze/K6s/8bvEK6/QM07w4aOT
I9PgCqMGbVaR8Rs712d+ctfrZedXY7x7+XtDed433i+eagCy04bHWPOzsnu60uHJ50Dw+O2lj9a2S739l/08Va2nvt/eMK1oowaP8TP+DobWt3l6rSWicofZfkg52Dw9+7nd/eHsPbg93XbQS+aqNkrVe7bVUVNmpnt9XZK8Hr1l7rt7akOkAuXY3QlHTw7m2bQFRfC//bOe4c7FMzdg72j7v4WcJWdo9T0nedo3YJWt3OESnkTfdgr6SROpHiQDJBuv224kKqhkKP
IAp9nxy1U4bwut3aRV5HRExNTJAr2i8/n//c+L/HPnByon9B/LfqVn1rPv5vNeo/4/99POi3m+DNMMSUMW5FmvaQfGa7iV7c5iF4LIqwTEbzEsyCKXioMGCXgWNjVAnRkdsYRYdY7A+40I66O7ANZxqACAfVaeS4Fc9Nv2U9C4CeHwSTRSiJswidMCdchGJUW8J33R35i9D6Uqg1B+1je8YeCz9g0CxgY5qC35qW6awJawa229QeBAPmuUMHQ6d8CyYRlA/WoXwFT1
vI4cGDcoeYqNcgp3YFcR3/w+TDCMrROOTMVsAJG3ygnGsQhLwnpSrCJ5PrnssVh7WnmjZwOfOb2oPQg/Iw37MxhBrxpDLwnPTtOn4LNE2LMCESeYMA7YH3wXZCKE8oW5q6mAA9eMkH4wD07tSXKVuqK1GpVPS0+B3DRAOL0UwoPYBJGKCJCC4zmUHgTVBsXoFjyjKuHESI0AmBRaXrgPlXKJBXpZqThXIqhnYG8qvsIAugToayVLXL/VE0hlrZko4FXiQSV6UlRNcR
/C/p6Zs867fhWf8+npbkKT44E6h/kzchS9xy/W5VNL6jisbdqnj2HVU8u1sV1uZ31GFtZpVcoek9Tw2xJdPVxEZzptdVTEBOCUIHXZ1PuWzCW8fRhLMVHBApK4yTMouWYGXt6fBQwKr2YDLD6YO/Dt6H3kBcViYzaaTL4PUV8Fz/31zcuLn42c3F1mahfOIG2Khl4qYl9ZUlq0RegtD4FsKzbyHkBcc5RRDOsFx6uKTv0LGJy4IFqO75du/H3ZhzZMeBnB/J2UvMJZ
5XjQM0W48Nxo6P4TGc+guzLc+jmdQwjOdteYJmVoMYTKB8CBv4oDiRM2SD6KWLQcQlgmYs0pn+pCIuR2c6VPQfdfQPb/lgPvAqHTqaHEVNOdkrDmRZUPbQgZcjmuiXx1zOJq1a5pDk6KZA8DRyPL6agfIF8ZC3ejje6W+Oj0eK/Fcwku5GMoLyH9MgYlCvwUMF9jm30T7ApSWLEFQxTcOD8MN3qfB17OeElng8lHCyTJFpeUF4dHz1Z4WAx8KQzT7j5NyFss/hMRrs
k7PTs8/V6nMQ1dPnn8/Oz6vV0XN4eY0JWfWsf2Y/PesTAA3OmzzHaOz4Ufxj6FDT4RoMWQusrT28Nkugn/m6+RheoKQVm0VMezDypzQYgSDxu5a1SFrpD7TpM6jmMGqPlAv0tbU3p7+eA/1Y1jkKhPLIeuYkUrBUpu/ommPKeMA4RoMR8NUCz/GnETdlJtQE+reSmnU1MytZoFpSDoaJuUsoJYBV/OPX2Tema9FYAeVrDjMtVjAyPJWHLas3Vu+t2/dANRCIQY9Y9U
J+CZdCAYgjAeb6a2Wj01FdthpZLiN7kDJw7MFRyNGTXcJjVF045WtrjxE4YBGOqOOD1wdNhTdg/uMIRlyuT1kbMkMUAY5ATHtF4N9BnI05cWhKUSKkEk0YSmRfpboyslUiag9IDIdCwNqaIfgfmIVaNfM5LandWZ61NSfnZ4gXGvcNMmhL7equva749IJhT/FZptgFG75Bq9Lpfo8G6zdo8DYV30192QC8q+LkHA5HiWSyTGu5MV5MTv9MjX2r0rtra949/YjeetKX
JcqTX6s0uOAV71eXt6v+B2wwreRH7TDV60pbzKLIPdvjNyq+m/YIt5nFw4rMKbNPqQmtWHxXFVNWLpM9WYdAaQbRNJzPKu0gKquJ6Xxiu44JC+a0L4gcG15er/j8SntgO8MhlKeQgWlyWkTSii364RYQEzEnuOoZWnSqlTeoV2wExq+J/Llpt0R/Qf8Sivyba9BcGbVqCfrPhfX/rPX/brv1eq9d8ey/YP/f2qo15tf/NzZ/7v/fy/PwIRzsMM+FDi2LZDvqwRD2yI
vALnoRQTMqOOLRdKJptFUEAl2k3A4Ys0sOF8GEeRcYNVCXrsvtCkBnKItpbQW5nfSnfjRVGwiY/NP+uUZu6/CwBTg5+Ns4iiaiWa26bIqz9wmz0YlE1a/sUnjVpywcjJ1LXp1KJtXJhL2QO/RNTQN8xBSDE7PtMptEZQwrgXCiIJwB4jUlA3zJIU4imE5w8siLsFh28BpAjZF74/7iig62zGizwTjezBY8WeKRkxq1gs3gauy4vGIq+Ygdsnei7CsngPwW2BzkOcDp
T8ShUaltVSxZyi+ZC2uGROL+pZlnqOTNtgU0DXXOL3k4i8a09kSHJ6jHCCXkg8iVC1g+XKB01ADaobgAMQ6mrk0/VxpNwRSt62A7LqpjBFTXTo7a3WqFaq0q0ap9x68q+oqca8Pb4CpZ1MeWq3bTvKhQisrUtN9w0jfm7iTuvHysmkfOr9lmLGUaIAq4I+7zkDQXL85pbV/aZzRG20qN9FCuF8KFxyJaJHCdfsFiCwarzRssilMwt5zFxOuQ5YyvtlwA5ops38OWdp
Vr4VSQ4i+yJl6ooxm0QcL9kvae9t1IKzlVqPZq/wL/n64O/xXnvxrWvP+3rPXGT/9/Hw/mfkEYgZgJTS5Se5QAxsAnmvams9uGbSqusHB0eWqda519BOjpHgk8BYn0FHRKBnV5vmQVAtqYrmmazYe0hioiuUVr+GZT2rQzBAv+tg0+/dNQMHpCdGOhDwPuuIYbjOpIoDxhXLAOT4qFUIaNuJo+X1KLD9vbYC1UUMszrS9hWi/IXv9xrqgUS3Jez4tbYBwT5vAR3SoI
Qqm7IWL8hykF8/o2A5xH1TLpgLsYswyLGFWFidLEzUOAwL9fC3Wu5jB0gyA0FB+oguSkYIm6Mo6a9uodWsQnyVqnvtCbue4vkcxpN5klhVbPo9ULaPRlwheFl9t2StGlPuoJp9y+UxGjsYjxrIjxbBHD2iyiWJuI80XT3h10j8juX707JWs/x1ZTjoLBmvtGZ98EhrP4kvrEEVIC/eqpLqGjuJ+ReFihfX/DrIiJ60SGXICXha7A0tNz+U4z9BADH21UZsZGCAiNKc
0Ujv2G03K0DniBnbeeEdCDPQq0/C9Ofz2n/rIKpUhKTTmtnVPH65ZeJFZyVdgEm2Qbpzhzxm4vgTi16uclmkgbUidkC+qzMGrMc7PAjazqzvxjXunahSsyVqMK7e1xQy/plfeB4xuOaS4WSk3/F8//ctus9x7/Gxtbtfn4v1Hb+hn/7+NZCPoUnUoyKmgxJEts0Trkbht6rYkbabncQRs6oxIw2mvEkoqY9qU5oSNi1xWchvSuxYC5ONCQsZ4BXdbnrqHvqoVDnFPR
bDMrn8XlhySfQsJCOqfoM4/P5SUTHNxR7CfvLcW4FjhHwL+P+HeFf++VAKW5Py3ntGUIKKRHaYOSFMlcEgeyNAM7AFllsg8CP3L8qZpPUiX5SIBuz0wok3qWefRrbAc2A1uB5B6bGKjMEvHJnOVV6oxzwPcZMNP2day8olOPq4hjDrJGOcxS+mHlP+rnhRA2L3l9LhbdKMV1WnitALMUMFOAjyngI07adw72Dk/owPh8zqKvnZ25ZAX4MwL/7Cykj69nZxG/jj6hmF
/QuLCM/4H2pX4bJVhfRkTmkxAGOBMMrxzBv6zpxexHX6sVuPvbVgnqeX5xSphxXb+BazFXkq2ROZvKB8/OhiEbfLK+fKp/MSEpU1LjZ6hQszes11rCunEj68ZtWa8vYf3sRtbPbst6awlrmdat5m1t3pJ5fYOYfyEj2qWT/wsmFLF+cxRy7s93NRWE3F7RV1Q6mYYTl6/QOCH0ceT4K9RG5YHrXPJVbZcIIfNHXDVgt/WqvbukAcd03IpHDrpzMF4x4QzM+aYUUDre
JAwuuW2uaFkBuWVf0kERGwzsCuGMPIYC1NdMc0Wrv03cWEr87HbEz5YSK319m9raVOSoTIxnFA+NJFzIuLattwbRlLk6XSVyg3BbdaI75SoCphQfUwrVKaeJOzxPKJW15eGu4/Mrx47G2+uVmqnd4P2TmjI/HceyAmvVueep6Kv6WFYsopnLt/VyWZdMV1RPF4VWOfZ8898vyLJ+gyyJSc4Jgn6dEpNr1/EMdKKWOgtmSuCMgEatBBs1U0FcPqJo4AaDbX2KkSHESd
SQEhOZ3LBLjumOMdTjc3mfklZ8oeN5WHW/H1z3HH8w5gI7lfYLEThhdgKr/fvMNeL8nw4G/vvc/8Xe2Wz8vP97z/0/d+XhPtd/1+f6f7228XP/714e44mmveUu3TGV6ye0HjJ3p0PTTgSGtv4MAt+d0RaFvAJzIe8LO1jCXHmshF8zuctEU8omJFt6IycaT/uVQeBV3zPyyJj8RNVs26rad4N+1WMi4mFmhvKKzRN02qcvX768YqG8Z6KXGw39HIwn8C6GNBoYZ+U1
VZpVAXZlJPcw6Y6qHVwJucniBfbU5eDYWOAMHXTmNMsEQ+2DIa2LVCHtSdomYJ2S1Q4KmL0lO2x00cYN0JF7nNEOD9Ul4Gvi/3Hyh9FrV15cYhOEN43h1M/md+UX6jB1ZS8jrxC/jIWZYgtVHfOZO/vIe8y3e7aDkzo2g6+F6v8uUYQjr0FhYEJIjBgDwDClYMQunnei2gpiKrHiuhY5mlCo0fGXc0PRUZM9YpqUJDqQXUMK8BhtdapPWiGV4f4ztMMQrY+HIeLEmY
g8AjwEXRU9Emf+o//RwZCflSjo0f1qf9QbTz2iM5/HZPuBz2OmBx+A+MGRPFtqKrlVUxNVzqsqueMkdS8Pycqu7yUn+Y3cll0z5kWncCqRvMOiFE0NNuZ1KK+E56oqAQtJs6lO6DOvEuOiS3wo3ehhc8ku5B2DmJ7WIBJVKTmUkSZp3Nd5AbKSoiAZPGOewv6erz6FZkpIZX2Do14iZQacE1BqM29C2zePI9lTMhH8pvkvtLNpLJrvXJObRvHbxCGi/bfH/9iv3/f6
70ZjfXNzfv13ffPn/d97iv/qAqYcufFhEFpQZLRSSeE+CiZll19yF+zQucQ4OQhsTmFSQ9Jdpx+ycBaHWEHgONruxP5yW8bPSvJJsZvH14fV8rE89UA4F7SsLE9MZMEbC0UhJMeBOP6WPCu7POqJGYp8Teynca6CHucRBqKeRE3EpRtVeWFl4atc3qsgJ5HjSoK5Q1HKNxEz8md0bJLO1qs4hNEgfkGHHr89ASNWhzxLVSmyM2nHb+rL//uI9hl0urOgE7V6KxWJ1S
3pOQ5ERmFHkcm3eTJ5jXoJGd2LUGTybZ5M3rNeQkb3NxSZfJsnkxexl5DJGbskk2/zZPJ+7hKyekpWX0ZWX0FmpWTWMjJrOZk8BotEQ+a4cu/WOJQZSEUkmcjU/+AHV77EnDOLJiYouizIWYfImYchUgsx8c24nYmY6mZslr0NYFiwtO/kkxmcits6c10diJFM4NQqOBjYVkrpdRNOF/Zk5+1e6fg8YSjpiOHpPJ606uff5ifN+BZ40m5vgScN9RZ40jJvgVfPt9dZ
1lTnXCUwUqdYPV2AMtJ8ewjGkcRTWzS9AJOuMQvhcekxCHO5+Qjs6MyA4lTzLla0XKR5YxVCOcy8a0z93tC5ll4FDJdEeMywevxHykRCxe+p0RpjuwSRS+7OVXoZ29BsIkjxUwl2diEtSbWbmRtNXL1iaujpnat4dVblv5JFTN3j1xF8FVP8CGdNPU1Y1Qm79ObbhR73t1GIF/FNBhwuLiOLVFuPYODUohfgBAcjUTqWv9rBoKl/2m3v/3b89uhLTKsv2BE1SZ64T7
jSB/GcUB8xt0c+p2fzIcMJCNRpKyxhfvSPzqGiFc5Hvpy19F4Ja/q4WVz6/yWhsEs8mZ4l30smQFndRs6U8hJlViUnfm42CYBsLnBJMwALyuXEJHJPkd5wSuCZC0ziKYUy+g4yM+iorOGRRd44FMyF+iA/jYvP4H6VPn0hAKR2VyXFNR+JKvZ285GNOgPXXHQdqgVyjueMfOoGo1MpGKDq1hoqxDQXGBiLXiAdfDLUzJPQru0KfGWWpnwWxp28GXrbcUfIdx13yRXU

[5] https://www.rfc-editor.org/rfc/rfc4648

FeMuvlyTmLH6vNmQ37bp/6qFphwj/xx3yurGC0Pm/seVXC5DHp3K/E1UpdXxCpqbvMP3jlZ5RzcZrfF12kf/3963drdxIwt+56/ocG9GTUViRMqvSKHvZpKMk3OSOMfJ7N1zFA3dbDbFtik2wyb12HHmt2898CigHyT1sp2QJ7HIbqBQAAqFqkKhaijW7XjDlWtuUasuPMDaZbSLa5dvWK6xbqngDdYs1dvukcW1Wg6eruYa+DTm9Q389M23//dbaALN1FSegyNNsu
ztclYxPOPlaDRJ7Ajxb2homvWjuQGuyyFojuWVjXQb732LJwk3P2BXoKpFgri2KqpizVEg+jglmbOvnyBwclTGL+9fwEBQOTHC5fRyji1XiwG34Xg0ZjeXTWhgUjwMoZepy/2YvHmJBSn2833JLOzktQbfo4Kb8z2qth9H6OiezbdscDM2OE9mQKOmAf5ZiX1HYv/1y3/+9Gswzi6BLU2vA4zKQwxRgaR5CXBe7pE3VvTpAtZ6IjpFvwuQdbkPkuu+H76nKr0AUFzx
BV7XS2NgKffAKsngAAO+FwzR4EBhknCCQM+5Atbyom1Czo0i3B8ASLjfqYIEgDpotuCmiQa9WSAeiY6XKcKANlWxSnhd3roEoUx5YrBr1B7vW/itEsjOyv0PG9KAurfZUmjNeVvK59BJ/ANjmuTqi95pAmecFHekkRnezyYEo/Gg21C4G7wAuhz/n/T/Ba+H2eK1MYzBj7UtWMJ4pV8N0HNN7EccAiud2uagAdyQziMO+sBn1chhhFELu7HhVjVPfl+mc4rjvfigNy
mD6C13KOBoS8vL6VfJHoGXi2GUuTBMQ4L3vqfZsAKoYSwarn5QDdryIh0fEL7wnNvGxOI1i6Xh8Tvi5D3fUk1DKJf+nXD2jXg6WQDQSxNKlqnnlmaDE8IXz1EUycHXYXBKTEVzEQtYAdOw8PAFmPsi6+Ma4RZ54swYh3ZHedH22iZaFMvcrud1LWPCKHZn65narlnPVdaysvXMZrIPyUj2F1nMf/L1a2hUrl8dtU6vX0Wod71+bdtsirNWMrt+8XBynfXLB8V3tn7F
QfNgmU6GjOvB+1vIZaDWWbkUEErDwh+CKXgr95dXX+998wugBphNOBwmDAoBoCWBseFBl4Mi5U0B2L7THDzYv0mTWPgcM4+s3fR97d2ljS2n8STpO5NEj/bVVJU0SIWpUJAMz6qUVAY8SaKRBxgflQOmwusBdmHW4Ik11oM7wrnEJRwb4PxoHx6Vt8DvMfXMGoDPSLOZ+8DV49oGdNWVjXjAV2KNNdcAjpFlNGTy0SgDi4UYxupNRu8QmuUTc6rbXSp9ZVbuJlxJby
nrbieST1ZvJs3JWadZtqEgaCSjH0tPabj8Ad7r29+vPo/iHaQA17KXnvLVJdZifHXV5x07GudkEsB9PZ/He8GQvILJp6y9uMz6rp4Z5CXKuf5837YBNQlB2AP5bIB/xhjQu4UtuGigC3TRsMOYGy7rY38fHYDPC+6DaXb9fsiuBCenK8QHx1L2I4WE7LM44cFkWZ+I0M6r5uZe2RBX2Lt3ipPpL8S7W1UvkfW1KuAoPmC+IetrVb9VfMKHFms7hA4rbi0Q6sHdmyFM
DG/c7vZ5qztPMHka3xu5S3vEX9d0jskuhPRT4PrL3Eo3ekby9QW3IjADhMWY14bfvK4WlBwhqQCVopzR7mQy4Kl4gisQJlOt3fPwVw3wiK7OroKJN4cMSPxRgEglLtF+uuDUHu03y/MZnjBinAU11iu2V89iWWpY98qUG+/vxOq+sc1d75wr7SPlVZGz67lLR0TCZI8WpnYa5Jjt0V9hUYIP2/FBKUSkPwmSg+MjTK6MwRRpC2H+EXNULGstlpbnsLCHxmiq38ca6+
DjbTSu1M9sUMgq2ABJKtKCdYQqMI+g513q9M1riUcBDesIhIfk4gx+tLsoy/PBBryWo3XQ0i+gqHjRKrZrpslvFqdTt0pTW9qonXS3TfO8pEniHdRm6C/W1QMxpOQHZCYgQYTVboWDfIKblULIYUBiNFrHJe3XDsitW7fj0ioZGYZz06Gh2mtjV00Vq4dhs5ZEn49rF5UJc7bRYUxjs9MeX0K6c8PuTSSkm1p4/xL+kH89CUldPr1D+agW4gcrHY0/BunItz5vpaO1
pSNh0kGHAk9KGkvsW1uRaSsybUWmW4lM5mBsc5Ep3A0w+KK99EpjsYbYRPtpvdgEz2fLRWJPiQbJ4jLBveYyC+JkMsnLhaJ7FF4wT4qBPI/rRZdfXn0NkgtfoA8o7QCgUCEUSJkgXyHBffMLim/DJF+kUxaH6oH/hSSu20sadyRIbChGeJjl5XdzbIFFiVsm+vthEGdmNkWLO3l/4r1KcgE9wHOOPDiR0XOeucFznunYOTNK8j3dpwMClToC5bvkahwtgQoxCcFuq9
TMz9LREa4zkJAWq/02Lyr8Nj2uHZadKBD/EY2pK80lMsyi4kzC46Lkb0H1yqQUXtn0dYEJQBdGrNh3zDuBkVzywPN7nJEodVyGijfjMYpm/gaCLpVSFmoFs+M1ukWBckoLqvCxFiZjOx6qLzhNM2im10PHWJoJcxueFmwwiSgbVpJjBkcVy4fGadMGZ9TKoryVMaY/d1vhKahv5UU7HifxW0UK2EQZcCwaZ9NpEi+SYbNsduSmXkmJQ0WJWhQvuBbHFURYQcalB2Vr
k7KpdoekPGyHi/dDvGX+vJXiDZ5PbuDNi0eNNxSFOFesFoUwSwtrgRtdW8caPf8Ce5WMtOAEb9HCvz9L+6l3l51XytaSVHez9mMWG4pmEW/tVGamrXCngH//zlHV8Bos+t85JK0NG95i2UTPWN8wS037bhlrHWF7eWTXX1F+a2oBgeC5vay+XVJiSRVSFa8wajkLS9u01Oqq8Dx6wEVGYsbaV8exdE/eH19jYWEdsZi21yf/3ItostYiEtmrV56Y+8tnYpaPJV7tlL
Duwik5OLzJwiEDmE1tve4SIkfmns01vdliYvOHt6LcU6jtsvrrLqtCIvPbLjCPyN/TUrOorLtPmcG48V5VPN7dLqy/8H5ls9vfdkl5NsoHWkwY+dusAsweG+ofZ/NsORNLopj/OVVZPdj7QGc44azQ2FTepOiQIQfOaO4FVaBfObl8myqmZEXYv2P/pXTUkS8dCVa+cKKDnLaOG4TkMFvUoPhCpxA2d7vO8Euu8jyzjQ4GexkvlvNEd8G/8H1snhaR9i+jWcT0VaNq
7L6OJvESo9SLsxF9WMYHZRqhMv/vY/eVRE0hIXmswgoJd8WU5tE0XVxzsmKNQLlN7Ni+q9LwRZGifuK/rJDB/GKF/eOUrJmnf9FQ3yvjfxMXvYcA4PXxv7sHncPHhfyPT7b5Px7ko3bRxfUs8aNd90CmwEQA9G4nwrOt3YBSgZBHmDpGYaao4sOKwjtuaQoyAqVAJoiugXkB9zhPMOL3axVAA5jO7lqfFhUNKM8ThglKZ8gaEWF6hRLFFAM7HTHO+88N7l+hdxxHcI
aG4+Hc1iC3NreGeqMEPvsOdmr1Ko7mTh31WJ1/Hqlh0OA27uHPyZxcQ/BEKca0LBi8XMdvzi3u5PB1hDVIsjjis1rdD9X8jsYZq64s7NdSl7Uq69FfeAeDVXylnqecjUWNp7k1i/GmF1mAvFp2CRX8lc2lw6vS5vB5RXPslCYa5LH+6eWv3x6hk2OiI/MM1Z6fYbVrmGuUYaBEik6V8XKepxdJoKdqlGJgKlDLgZhT5RA4i+ZA4ih0mm655+ErZoGa9+bBvdpYDgCe
KcviGmA3Jsqfsun+bE3CtCLvOoRTeIOYSnpYExRMfh2ogv21HKD90wAKbei0BlbHgNLFcP7MDKElm0ZAff9eERo+Q6nJGy6TbcbeuQkVTqTtHUm8FjtuIPd36O3xnPkAB8eih4r0Kb9MSfwsvgqrtIuGxMF6tt4aBzXGGofb3MrVKC4y7gqHII8Vn21pTmxCj9uI+WcZhyo371uV9ZSzgfGdFel4qH8xOthM1JM+PkHg/BSg0ysYTG/XT+FSEUOrkFN54nSjgIbC3lf
U3SqyEm07ojEHcMmCT39HpO4w7BmoYd0ULgyybiMGh06vvony8GEx0LGDOgXHzFePHcd4XHOl90TJchAH6WQWgq/AO1oJOxAQY7QVxt4XjFAvvJx5tjYH2f0C8oAvigrK6mBx3aWeCLvV69OVof0Pk4rxRefM51FPZwToHWEM/ISgHreDdO1Fc4xINhwqNuCvdFnC6w+/ZDR47pb9y3+ANkwC94a+619aIgcSgR+RU8QXrWkA/caEYoxQNtmET3hmts0RDw9PUivUm
3K5s6ahVE0xFkoJlTYXK3k1B6/ZhoNreM43a5kxd1fUCE9DD4HIQdzwKlR5mWIr9RJqqZ2RiwtkESZKIjR8cYXpgr3vi1MPtmJFOyzsWr5wqzf+Nt+dbTB+ejGj16m7i9aWUnApBcBI3C4aZSO+FKyxwAvTFQdqwZkhcb4F/7193QMVPbBj/K4D2Ca5x9YRdsQC5ox57V7muRJ797h9fff/Dt98cFcZOXzYwBjx6TN8U4m1dhJ0K9U827ClcCfVU+jbh/Qr7q6tMdU
PgRYzmJ9nb0llniYZiiVgkK+ffN8p/FJTgXi0pkkPBdbOSIqRfJVOH86SLcRPIC5BTEWbzPoZVIIPvgLaqlo3d2VV+a8jdaUfp8I7S4kfG1npbsvOnbDUBUjmPCp1npaTojk3Le7A+QRpea6elhBDty/dFgpM1SdBlNhOf/IzzawkdTu6VH9nmCiTBDpr9cka0HkVMbsCctLAnPGhDcvY102nF5FLBT4t9NBz8+4oemMavcJEhTHp71UG56KpL/1LImfCqw9snCU5X
tKL1I5KyrmCBggAnUCQMQw1ES08qvaoVpl4li+UcdC1Q4t1dV6W4Q7OVE8wrG3H0LrSq+0YDrQL/isH6d6DxhLU/PiOIFgb0f7ks2w6pNDVWJ+gkQdyRKYQPFyxC8tGuKObgLxuPBVFOBqb9RSp53xMaPyQxZU90Ks+M+Jui4XjomwfpmYFyyaXqMr1Gm4mD7PazNrQ8ZpVDtN2CKg7oYVsDB+jEAiRPt1xD+NSHJov4bF4xosthFc0QGV3rf7nq1c/ff/Ti6MgPw
eun8wDmI5Ph7RizAIBwOlOiT8zoiUP4+jiI7twm16GdPuCWuYufrmv8DcAuQjqF8gpDhoO8jzMfBOTFQYfEVlCqZGOxnAQxMdm5kZFnUsPsnV2//bVq5evjoLzNM+RzoFS59cYnk7TGw1LQ8zFFda/0m1qOsKEZdGa64gOo3gxzLKUDW34/Cy9UEdV97aKBlH8ltrE48kK9d5fQfOytzULyAah0ssH7R39ec3CAaIwa+YT2rTcSwXztt42zawjbZFFoeSdJRq880AP
ESyTlH8xSNRXNFcKW78TsFWvDEaq1+q5IsS5TxQ8jOrEVJBCNKeQq+WhGWNKeBJY+yLdCIRaGEBRXQFDQU7WzBXIeLIcJkNLA+4FClc4sTNEtqDipmbJwr/ZUXWvo5qEHGJ0yisbJlW5NeOm68HFDsogdWaa45Y0ZtF6J/tV2GqUmk7w9rGs3OuV30b2G9GMMT4u9QwxaOPFqJYgw9jhdkNisqEnvykSNRH46fosj2L/LDGiP2Uh55dp3s8Rm5AHV5vdEOmWEg5sUg
nA7HoanadxMJtnZy1FkuI9rhQjU5jciJp2cebwMH2Z4ygjLZqy36jM1foBIqG/Iy670jCIUWVDcj/oc5g6mzc0xNgOfQ6Ma+QEemZotFR8UF5TlsQHQCR/X45GydzY/brPHDpdKuOkTVnKfE1Xi4ZDHQFw4JJ4pJMTSLSIbAoLqSbGgA0VWdoTfffMjMee16AgdtIe9viPuKlkIk+CkHDy6fAUgNKdJUvLkSTsEC2vccuFyR0ph1kA50KgqquxqkTIBVODiANBjC7F
CakX+aY8lWYHMxuYmjq1AznC4JsVddReZeoQpflYByi5nTAB9H5rfpr/1gxOj/EhdJFe5Ytrikk7jPIx7AHOS5RK5hGyckWdnR7/5l+elvPCdBa3akrw1+nqIm8sH8OeNYcpLeXgLPj3bzq9bEWXZW+hbCB4QMtWhILzaPp2mGJc2x9eGaCWyfLE2vMD/7RAwPrDVNfcAIYuQYV8wNypQZ5LQRixobYlfqPN0/wmuYzYKjxBdoanZlv3mo/O/4dj996tE1C9/8/jJ4
edruf/03188HTr//MQn5P/LSMEPOq6IQIedXWMAIryMLIukCCDTlKQz0F+RZeffWBHdJ8a9/jofJCeLelQfK8BO8plCvI7uwudz9IJ783By6+j80nwqH1wEGTzIInmAG/eJt5hT88byjeDVQJSIlK0TgKQ18Nkhpci4DdIMJPJdTDH45V9DIgd0DuCtbZ7Azb1P0nAIbQW6BiNCsur777hgObZGbk5mMjT0k6korsB7wQ1N02UISlWXpswDNQt7RpFdXrBv4FRWkcl
fIiM2Doi6SfSqoq/50u0b/BPrUjzM5Xcgh1feDDGgDtOCg0H4BSRWtUOvsOg5CCNRIPsIiF9esSFg3icToYIJMT0jNfZknStQokW+kpA69/+vkyh3YR1fZiIIdJNNhopnPLk9yUNSIjrjPzESKVLpyPUJq9bSEEmYH6gMimQc9gkuUiUYijP76jTP7jTkY24OuPEzjyoNPLr3CUc7I+eTu2rQvI0DPofTHHo3TI5ClAj0kYLEofoeR+HggSbmC2VsIbS6SSdJrR2fq
C3NIpuNZoRVY+m2a33il/riojHL2QTNuFwNDxUh3wk1sOEako8qOpqZExdjnVd2YefMzLmK4MNF+bsd8NgcB28jttEFa/3ggwKzC9TtLEpW5Gp9jruH7zWQ/BDEo1gCMZ4JuYMAobZpswBlcjgwqBRoWQEl+MsLyyU5CqKF8A5cKJ1UH7gNGyrkGW1UWM5nyOpE0SJiK5cP6prYIT6DaF0K3TkNBnquP8JM+0rvcBtvoRC3eapAK1jZS3Sjg86DBWs58kMTSqqLSQI
aosNPsBygxkZqeHVjl2fAG2ED0L4/+gIw4JYLCwcHJ1KOBz2sA6Ot0ZcSKUN+JU1RW9W1SfBktp2hEIfWqsSXFUvfGB1wKuhFydtLeCiWhH4+ls9LkVkKHhnysbzQMTyPh4cax+Sfo0TCWbZnOR6i2mUUYILrA6WMv4wY4/ohk1P722fq1PeqSw0iqbZcmGQcEvMo0vCHIGEM5WWRVeRbVCcx88ZNf2enjmNERhuR+N9nvGBJUzCrmiOqtHXwnjwHrjRiMjxRbTg/8
KoSr8av0naLtZuENnR17jXx9lyuvhv/tPnLWe6PB/AyspGfbMFBcFhsFskGenX+BkU6ga7tUhToQ4UKhmfur5t5O/UKw6FRAGGtuEuI2PSRyZsRSxy1x6NQM5H3+d5li2sSMkyvUnUB5vEXhDl+fKc4olRyu6Ilhhw9CY87OtwGYqVo0nbwO4j7PKeuT4GoYXkeqki/Vgj1JcVA2BrN9RBnfOEzFXlSAl3BQcJPL8rjGBBL9GXyrJL9L6g4WP9hk/W8H4XKCCUM0FE
ARwsF6RFscp+zBKtkvZ551ZIqCFlDm7x57bd4VTRC/VP2ZwznoZF4TzloKzF4/4QL1KUzMAaQHG1oYE813GKLZENYdoGcxiD8W7LmiqBUVeSscBAn1ONuMZzZ/z+9jd++mVPDWHDnM8KEOEu3khAjXeK6ZtZWGaE9gLSYexLlMEVlpxrt7bd54VmqwaSeLAaRQkp3B1kePw5m0HFEgQlLrWw9RTtE2beHJmHu38vtIY9bu+KDUIuil7dYlEton0fIL9QVdXEt4OXU5
B5pwlQfh7Nr1EiJC8PHvE2dUyC3Hf7576qGjxek5KpXYJMm/AFD6C7UTLfg07lvKTycYT58hB0W8mg3+B1DhhLEBpSd20CSbAFZIKXm6YJORIxDmo1pjkILiSth+zq45zaIWW701D0ooq1H83znk9f8udzW3C/YsXE7ZJtZRPp6Tuo7VwDca4zcH8pDLUrQV1QZ1u2yw5bieXz4N+sDsCiP2ajCH0z1+8PjpVC1Av2O8c2FtN+99hkBL0I/mBCjYuyAaLqYdfX6IV9
Z1YkVta7Yjk1N+ST4ZFMfIYHgE1/R/0HetcRsYIumy3P2GW4HXyFeyTQk09MSR4DH1J+ES414VjRSTer52X8NazqAG+LhkB6DiXpTTBuWH7Gqk/bbKQlxQUuEhVWvjSoykII3o5QFM+zHBMARxcJ6YURzmaqUhC2C0NAmq+7pRFZ1Ow9dWPDVxZNikNiOzQIitZ6THQreh5KHVwfARNcHBEFVg+w5QvsOi/055bHv9A34ij4Gkg3RV6M1tRJcpUurvnm5UqMFFSJkE
mFuCZOoWvtaPkovsDp2yNR5TKbv803wawAWyJKCSHR+rgYL/PgbZLMgrMMZUtFMSp3mu2GJQ/qA09gsWWyILX8JbqS7nhySwgvuCHlGfcV3slry7L7AZfj7T/jnRnFS+AaZIHFvA0gUqL9mvnhpbZ8ZpTJAY3WQsIaJpMFHtxzJ/YNydNuTYar6Bw1IrQKXY5TaPkSMxew2RcGEWYNT/RVLQWYNkmE+1xdvQkR2IuMkUV/PjT6JHQBdJgoGMrqruRbvHeZoWwwRjkE
v8c8HfoIVjVg9jR0GoVGvkfgO9BfphK0TS+h9pIVGWC9ezoRIGnCw0xFmqXz0wxdUKKcLeQ5ht/O0wGQgWg2NO3tMwotoxSra0VIUdkEZHxsX0ppwNjoRGRAngxLxeaoJmkB4yhXxwlzlCjEFKr27XpiMqwgcMVbrUvjGjWUvUzfiiWlVJvWNS/0bADa29FipUg4VlDQii1g2J1FARL1NdVoOvrEMtyyBvQOU7/HhNJyrZkK2pV9pFayjNjyCfc0g+RF0g1IIb4cZx
N7xqBbcRgIF3EZyOYbuceoYwdbM6CAc0y6ZaAKK/1TFN5Dol+6Y6iorWrnVqOmW/glpevAOnZ4bVsLlvDpxoHVXgU30qJduQqrZFglk7iqS0fogIbC9M0/TVbPBVVpXsHHXAs66QIKoIMuzQNh6CZ4tHUNncoVdntMj5eoKGRLGJx5whQgppckRWPZtmenqqfEWAE4szYykJxn6EGJIzlD2/88wrvo+pxKLgFugEaieqNXjbzUpvs9pTWZE0tn4NdlEnWL4DwdDicJ
rwJ5KqHuN6WLlM7PCkLbTWRYKZh+ombYXxBsuK9cFfzanmaAnI2FKVG6Mzb14qu7EpgM1EnmIDlLp1M+E7Wk4LMG00TFtCpORwO/nJh4DDGd5f6Dgg+wOcW0wM7StFGz6Y0MUcpkly5ytQcaghJV6e6ptkfxWp4YbkkkrVdRRwGY6RQhZJpk3Eq9b2vUXDm97Nsn1Fg5FsqxT1/Ipq8d+qq3A76Z3bAXjA0jEE6BY45LAto0AbLu2mN3Z3Odqu2mKGSB2plr2EsFEg
ptigIGNdwzB2bGBikuEazTjFzF9bSk8KB9EJ1bc43Nu4BHdnVz70D1NiWdxV3XKM5OuLvf3W2tW1cbFIrXX12ySoMiNQXefVNBVGnLUdK1ewat+7yv4gFgYDa+9c6N8J1rjCVzGV3nsGvO0ekFpGRzcKsEWVry8yELww03tgJ55dUY7I2hJLzIbVwAd5UonYFOxDjsCmKKl+X4ypwytAU/8Xkz9Tka8Ylyggfe5NNDp7z4YOFcOpXNCpihyi/SUrC/ZfsHOuCc61sg
GYZJiYaOX77WtHLRg2rndzUWyukd2xGp4SmqSt62ELlPymBg9iF9Mk7dop29BCNSNe8YJwXzZljF/QOBjRj3l4M3gAfsGahYWJWMikVmeg/aHHBJ6w/4FuZD3cS5zHTcJj7L5/e4Ocxm82w2T0HtbgfWE50t/QvdowvtDDxlbY7vAYg+t8Mpef9fHLvVwqny9RdH/6xhkLM4Jf7CC48Y+CEK5tzIPPhSt0FjTlwtaofzCjB8mxTZx39cPfw/Ia9nHlextvGV635VVL
NpKdA1xQu6YXhxyPSPPgGfXNxLdh3mdDH3pUCZPW+87DLQ2roBot0BNgCkpswBZSR7NYwkCxPrw5tkHqvbQ3EYtQgQieeL/Tidx8uUBETFhYWIz+JVGWJ5nsV9C5rwU5NK1+M0JuKegLYKf2LYoXdbZpAtFtl539iWteitjs+a1hHln2z6E+5k37389ZefX/4qd2t7UQn6OkvilKKB4carBGvcZcjzXd1ua0u/cdky2TZmkwhGWOJIToqypRj7lBGcXu22TOLUdY3
3vn2Fo4yoq9VM5v36S9P0ifAlywcM9j8xb7QShv9tY+lad48NCb6jrbRB6WYEUfsSQTtJUjWNarn9IWyJGLHV02oGtZCO2Lg9Jg4C6t2aJBcNxieTsnwdO3w8O2gWXV/v0crHF2l3KDHYhAre2z40aredtfv7UFJZw9tZ61YI8Afch0UTnrIhY812Wh9xBHWjX0XdiZt1ezY+2l6KA0eYtMKnRLGtEvbmAoyqt7Rj9IdCbkO7Urs2QG7DRsvzL7kinx3vW2RwbGv8k
33rOypevsfQuYoZP2IMSPVTqlVB/Blil+m7NKgaqkcF6E4atWv+NhFAeS+KICsJMC/pSC5b0f8Rz/EHh5pW6Tl8qM1esR92QjpAw+X0ALZcwaSIjlxIVxWXw2HyrAL+4aaL5KYR9oeny8HiGLFIuRuOjMlunvxILKGMqW6J6/h7tdKstQWA1UMWUpFZxQ/EZOkV5biJrKfxw2PhZiNW78p5R+Ge/AX/bhw3uvyDysUGH7ssRDunbeWcSFHFBIFCI2UtQdcwqhhKq++
HAYQrxYzP0cLVjYTfldkqLHOuSho5cqbDG26l3hTYx9lsteCG72Wkpia257LrriravGolaJXCFOuJCM21JJeUyLS0VU0lutWC3MMigBIqnREEm4Jt6OHXzFKb3fXC5lzjIF3R8WzwQK8XNJ20t5TnifRsLBqeAq8BWMH4gNbLti3usVCk/OAiwVtZ8LQL2QAoRaWELDjuOWfapWCYjawemk0PJPfDVYWW7mpwS+Drp/IkmJ7oAom/E4c+9KRdlEkb6aIkirTTU9331
S7Ju+AeqcUxke9re+rPZO3LMce0ZBb58mpsHWZiKNhqRdQjXHrnvquo9jI6FgXuXVmG9e7sE307XQ2ke1oSBiak45wZBpVtJKFagRmIFUGJxen2n21JPuXBRxL4UYES6XUuTuqnBfXaiPnLQzhO6534NL3Le7DYUn586888QltqAyiCp9KTNCO1Q5NCGkznyb/9oGPvqhC9xBEA94tiLWqmgG/if9TWbSTzZygaKjXGOkN3aP8eyTrDcUDekt5V1E2xe/hnKeKF3zC
TXAvQ35jhyr3elLokV/5YXGRwDfxr6ok7GAzyl6HiXj+V/rS0U1WySbuWfJS38fkoMVy5tZFS7loyWmUrhqle4Gg7RWrxsinpdvR2nDuyKGLm3I4yZ26dpV2UjRdHBb/uuUGDLzW/6tuUAsc7U48wcpY3Z1KSPfpKSYCxQZBzRS4gtJf0HHs1XffrOE2xuN2P75jpVKDaNV3vnDW3I1dyW65ntZ2Krv3VfS+nM5KlhJDCdVfvI6ugi/dqStaOVVoR+FQ/TWtGwc13i
o+RPe0quDV63uoGYLYwE3tJD61jmonPGzHgXr4vrzVqiZXYXvnbmtrtFfwX6snQHck13ZoW4lIpWfbZuiscHWrRcP6vK2m1hLHN+W+VgyvX+0At4kN6XsMbTpCrwXc785UViTHiKTTvLpJ5Y5KM7yq3c4kmusJ87rJqVTy2k04Zx8L57hqP2LrlEhhC5mdVpdVnr7C38QvzQEH9QBrc140LwOL5UhfaOhTDw6LX42A7lTKYTjxj24CR7xHfzSCbB/U3o+mJnpQ9vhv
o5B1Ck2NPfd3ISuUKKN/N6T2TKQmyE62rF/aH85wCqNor/CoNAlXMfGvm5Brc7Noasi6eKGV+qvDszqLsSJWK9pYJ+rMwxEutINCReBWeizurLgvxN1C9cIBh/fZ3Ara3uQHhtUkJmLDrogMy6Rw1cf/h8xftKdp4xZRY4uRXn3UZLDXygCvyGI5erN2bgM8g1ARR9uPPduyoWYDJ/z8EPbYT5xucogA+eSoFwwLgeSLkVd/+22qqxxhHFaresC2ELVDXkIo9Sxa+K
q160R1rQrnasCY4UuHfGBaM7dBeWRYz3iu3K66yuyg3rScGe5jJIsdoC5J7IuOsyMtuiVEUBf7NMDE2hi+NYf/MForoKY6tei0AoCXl2CxBhJiNFwsVDdAsuyItrqiFdjW+ulQONVgKbSxaQuk527dcBKjO6V7wSfiJx6slk6HLCTWxnUG6JIiMEenW6VsyK6SvU4+mLuMpyXzsCQEvxc0+Vsz+JfuaujErq2aphMdqDb4rbl3mQ4X4167s8eMGr9RaN5eOr1I89OA
5pIbsoFn1dg3T2TRptb8ufRxPR7/xiC4PQoUaCiH/3WA/uGQEg5TSw+AeUpjJY/gDOfGxRM6butaO3RWls5OlFxhWnSVWwix4Ujlyfnv8NMAbWEKPFpmTcZ0lE4myXAPoxNO4mySzWFo/9fX9Pmt2eSlCH9tzA0V/j2uJO+C4C+NGs+1TaPM7ufGd6+KzZxjAOZxNEt6g+xqzwQu/uno0xx4Hh8+Aftr7mHRU3c5y4DLxlISOkMO8q9NR2DWerMY9BkoS0lxxdL8Dq
U24z7hLSRd1yuGGQWLD+SlGakFtYpWz+e3HMZf7mwYjxXiGw9n7ZiaYILHVg/1hRYisIKxhNVxgQEnkJahuvcCXgJzi9A6c6gnsoCbmbyKp4XLUO683nAKf7jzKTyWCIv51GOPgl/NoIfCbXdBKdxMUEOpzKeFmkLK0RHh+ezwX05HHFGqZrD0AOGI8NZxoHeOg71Acm4aq6mTJqcaruL8RDqAIabQm+6hGJhdosNVb4qXSIrDvyH4YTrvYWoPRaO6IR9wioALE2QD
elZMh43ReZM5MYfRH/K8DChRw/1PC7ZTPyurmBD3uIIxOtyRlCa6N+AF45Xp43zVDVYq17kbHilgqSkvzT0ALE6nEKD7OqxcAT/FkTomxnej1AQCrp+agIAitz1ueCkHSlv6/HNUtAKjNVH2rqarj9EcFtMVbGP530n8/+5Dx//vHDx9/LgQ/7/T2cb/f4iPH5TeD0nvB6R3w9G7odM1LLJkIizydZZ1OZ+kbu6Pewk6t3PXcebWCCe3MtgdD4nx8JQIMM4XDFqNGe
MQkYkthnFaJ3ydcas1MMYGBoy1xaMMpXGfLyyOC6alcX+MGX24wD4UkBsWlcIzICqE0YjfvTM/JrwXmuOGiZVwrIlcnUqz26uZi7GZi4kSfs2EjI1repkje1A6hBMWq/6wrmzV7d2yqbLZQqJbIyrGJCgNhuFY6CiHrRukT+dcKsjzJnw5QHhjvV0nqj4LKZySSdaSrsU76Fv8EMEX8L7LRZaSbxwR5nK6SCccmSCbqVQQ6flsnpFf7Jh8HDQjOGq8oyOJfTTOfH4I
G9jnnT7sLPg/nv7B56D9xRf49clT+HKe4wFft9394tPgs2778eGn+Opx++Dgx0sq/6j97GnxaxB88azd6UJhp71ubXtfQCu6vc4TaK/TfvykpL1nxa9B5+AAymB7TggJfTizU8kBShzrndWuDUjjAv25R0862a/l6SX5fsvZDRVQF3L0gXU1o1mX1dyS2dwHu2HpVrOYi9z6v9wVl1EtxM5VBTXSzuYBxegx/QOYNNz8LOiOQc7F0RADL/OURgvOvpIvklljTrqtP
kr0PWzZ0zXnfbTA1pHnc7a6+iw3Xm60Tp6ilB1e0+4vWe8jp7J9r7oFL+KddRQTiA3vZ5SuooGha11rXUz8JYIm3RxYyd0LjBhvZ5xSg6MMga/atHslsSU6d80qExpx1xxrcUPbLCZEkioo3K/OW2OGQxvBMgfg8Dt4UWZBru4qAM8MsfD8x17nU9SNiEADYdjeZVXfe7hzl/g3D82GPQsMp/pVAXSjbBYTbIC6qNlB/D2WN6nFw0JrrDjsQUNHcdKD5YeLdOoizUO
54MPnMMEEZ9j506w7Lkd0q4ZUhnReccfg3iPe13LGAmJvaAvKOmASUiEDK+Sv+nPH3d+xcrzRetvJhg69xZKz4NsrXAWrHVroSho6utT7p5WKmzeVNzkzQ3kTJFVR7M82FW3rlC3cIVCpv4Ru0IV91WFwKCNi/R9+EPtvCeHqD5dTbW7TPdm/lDr2P8OH9r+B6/K7H8HW/vfQ3zQpx924XSK+Q4PgxEskHk4OQunn2HItn1MYNRqNO7SSvjxJZq4YT4JI2+vCPvu76
f0TG2kYot6I9y03sjHUkumvZYfKRcuLdcUOvlmR/TStOProVtj1ubGrDsJgFDj6ONqmDozSVHDjFdpmPHmGuYto5beh4K5hqpJARuEPnmRvy/thzS9CrVP6JM1Wl8Y7wVac94pV5051JIpVK9AV6iU/RKtkp0Kw3LFt/VBKpf96kG2GmanTsN0xnun5WqURql0F0+FdhlvtcitFrm9ULO9UPO+9cfOnemP28+fzP8Hif5h9f/Hh08Pff3/ydb/58H0/6+C1zjpr+kY
TvmEp+5uspYNwMjtH5HCr8J036uOX2yPdjRPzL8vTbtczZaqLMtUE9voSv24oIcJncvkZXkXpHQLioDJlkC6IhVhqyc/rJ78sWjHxdPVFdpwpR58U933g1d8Q0/jLQxZtW5bqsb+uXXYDTXXSrXVXBpyAp9ITbNUz+zfRNHcuEPeiq9c7DP6mio2V9ofxaDdfgXpVqFdodCKsqsV2q02u9VmP0T9b0JXlh9S/3v09PBpQf/rPt7qfw+m/22k+ClJ8n2pdRhH+25UMx
WsG+ueKHDD0xaK+/WqUNjHfTIGGSrKgzvRi3aMTnSHAmNj3WycVQqkQFANldrVYvSvrtbh+hsqcTZoupwGk2vwPahXH7U2xW1Tn26oKN1UP3p4NUghorBaT695QK3mo9ACsAFmaSi/tugOUzrF9lg4LioGjfubWKltrB+tsU4tFHOwipf9SdUbOyjOzkSjsJhI5WatkrWqjT+0HaHm+NBV6fFwq+O8Px2nIP9Ps2z2sPJ/92m3Wzz/2cr/Dyb/T4PkfLa49mV/iuxM
cUnPgkEyjcfn0fxtkF0kc53Dqqga3I9WsI5acCGzURmW4Nfsr9YKYik31+/b0gRfBKF0h1oA6fCqGsB6cnW/WrC+WH1HrF8lPNeCvV8T8E0GXUk6aIDdRPIqnwAB7C9o81zT3OmJA/1SeeBgpZlzu/mXbf59GrsHMGsW9v9ZlM4f/P7Ho8L9j6dPt/v/h2n/C3Fl79Im8X7tgOEB6PytDY2BYZjuBX1K5NsvM+DJfnIC4Asq7LiD3LmVrtLVQ6BbZnCscf5IVzt+4I
WIe+nKR281vaVF0zPbpFVeNLdyUEk5gS5RiKJRx02lY1bI1kfl9lZVx/cDLaCt21pZ/0zuKGG56wmh3trYQvsxe54Ihs3m1Js5oqR1Tii+XbZxj4bY+zFkS3vz/Ti21Niu79VovbUh39qGPIIJDpVpGPa0VRpkbtIuy+JbfXLrMLP9bKT/Lxfp5I41/7Xiv3Yed3z9v3O41f8fSv/faL/KHT1vll0G4QBzApttX+3WipN7alwHvkwp9hnW2aX69DWc3uAI9gdKtqHS
1b+G7X0/ePMa8Fos55gQkyUFTCcI2+xZMg8GyeISo7WntN3i9dF4ssxhJw9U91Dseh68UVkDMcgkgaCELDllrm4HMgNmat4tZyaFdbi/LwWh8I07NqXikCmjxUwl/Ahx8EtAmNDi8iQVYz0QrUXKzVR1QRb9rLzoRAVvxedQ8A2omirPVdYfXMseYGjAu+3OG+hOqmIxOpIt9QzbsxgXInuFVChtBZ+rkrv0F/qZumEQdYfU5jtLooUJcskIhe5BgtOZyY6WWnCPnh
Szr1xVabOVEuoVfZ1qBV4D71MK1yG2eNXSS8F05eQ0mNoVF+V5FvfPkn42g868teh7VjmFCpRCwdBZiWxSAO0ogR+orSxIW8E8tLRyiYR6wVuen1+y8yRQN66dxt9SDR8zlSQcBOZ7xfAtBkFJixjSq+f6DUKQmHvI6R5swnO+wsRilglynrFofsa5xiJFB/QYphIf8pNduWh2mT169FRRWDHTXZ+1GqqCeo7mgym9orb24X9O8jHC5rwTKevlpoBKcwIFin7/1Cfq
sXbpiMr6ndd03OETq/ptC3mdl7VXjYBKUqdCCFQPxAk9P5VD4Sd0CyRftZB6vbIxRQpFzQUe7+n3NRCxthe/gLD25yA8ARR5dmpmop76jGoFIElbcsk2oqJ7wUHLgZkvzwswtYrGqIyyCeZLH8E6/wytNpFDHBdnJdVHkyyiScS/qKHg29A2CMzh87b3tqbfmA5ing6WxEPKkaUfIruh81R1BPWvIOzAFmKHM4Lx0FnduFRKScMwt9cVyTDtEGblS5z6z/hHK4gwNc
VgM5byy3g5Gk3QxKC5Sn6JmmvE+4uezVKJwqT8wmqoiiKVKuLhr4AffHlDOTPoCz65cuaZ2k/8BgVw2qhs72mZEY4RfA9fgTSVnbdpqtgm3UKrtDtrTNT71meYNmXT9KRgY3b24sjMFNAEvVH7sdsDGn0uB0IMldvQIeXlzDUOZvM+Z4FSY8O7VUvHJjQ2CM5PGXFOyne8JV3xlv+OtyEK+Mu2tLyf43sBkN5mmXfyQVAwNzTltxSAKAeMQpDqw5688OERXj60Ak6j
KJ2Q2agZR5iELngNoPoM5XWQ8R7adNo6R+IccXPYzqBVaHpQuq1bDOgbwAAGaVGh/X6zlUM5+OTJ4SJHrqFy8zm5IUvE1VFtCZ2Nmxtp57NJCgMz7cfjaB7s7GPiSJGB9CQnmeUEQAgM8lPzunMc5N1TtaHJIh1cFd6zrlsN/j+kqlo09yC0Co+6xUeHLZOSWs57IQPP99OLaJIOg+nyfAArHvQnSoKE50rBzqf5TtMk4tS3J+JI2+uQNGBUy4dsD4asVTpTIEDkdi
YcidpMR3lj3ozn+rbA4jLrr6YGLVHUTfSemej6WdwrzuENhjrOzs8jPdQ40kq3nQ4xgHsqtVvVDuq3QT6L4sRkuNcnsNOFlaJMJl1fJW82tU7eDDB3mKqIWgjr48F3yWSWzHNyB4QJuEjm5BGIqMIKp0uKOVBmkNvVSKsLj1wWfbUtmB3LIFKW/lQ7fXNVxUMErzGJXq9cdlzIFHylp0wzlybzMcmEmiF9x04DJ4J/m62mVWcUDkR/NRj46UD99lmVaZ6cYttXKCNe
5dy4aBcf00/R5ggLCpzUO5ZbahAyu7dIUOoP9ck7TJv27rRplqpaWM1jwCN0d9BQi0i4tEHGaTn4oLdQKTpn+GhQhuGu2jKqhiwEUfS6VRikPRqis+C6ZFRo7yT4nOK6GjhlikbQ+AXnxCRabtI3F65mHVcO6zCwm5id9V+MHnxtfMT2X1Bksvl1e3Z9H22s8P867D4t2H87T7fxfx7kM5pn57hIxuj5lc0x+XI62Qsm2Vl3j6MBN9QLKDSbZItJOgAywW/oyDObLB
qNUXoGajSqH/CznS8H+DYPW43oqp2DOHmVg3CZhE0A2rQPKXlg2PyBNYSMUyjb99fq/a9ImckiBRDBj9DIz4grV4LCjcYwGQWX2Ry0CYxcHU5bR9rK2Am+7MHWBv88OjKKNm+g1MsQOwkVWDBSLw6BPzkvQXN5rJoZJCWtTCkVTaGBAwm0WwK06+DevT1UVr0A8qFE1wGsKorydBgoEelg+dwrH50Ph1EwPUJ9WOPGZGyOF02qcwcNsmiNVTGkVVdb8CqH/7+AZq5y
YO2H8D/Q1nWH/nmE/zzDfzpP0KnvdK/i/wbKH2jhxRSTZ0kIEleHU8UoZK/yNhtSQe2lB9eH+oGgj1RN93XXfdl1Xna8tzQq3ZYo8KikwCNZ4FlJgWeyQOdJSYnOEy7S+Prljz//89dvv8HwOFShieg3QVD4e5SncXOPH3bVw+85NdZQP0dg++iosd/F118NL9D9ZRiE//Xbb3l6do7KYfe/WsXijyqLPyor/qyy+LOy4p0nleU7T6jCHw1kAMhAQk0tnHtUD8gJD8
SpzoHaXESDI0rDy7zDVu0Wq3aLVecwaF7FTklNOaAugNlyPpskBRiP6mA88mAM5tllAX9cFtUgnnkgskl6UcSi86QOBkyHB4TWFvJV5OST5AzJc5LFveYSSHVOblzwFl+exVHYIk4d5bMkXoQH7YPH/Opqkp7L9UkPr/EhOiQ+RjsjbRTRRQJbRzhqoqq1yLUQkl+cgbQ3i4aY0xo0T0xAPBhkV/pnc4FODc2tP8L2s/1sP9vP9rP9bD/bz/az/Ww/28/2s/1sP9vP
9rP9bD/bz/az/Ww/28/2s/1sP9uP/Px/nD4zqAC4AQA=

## References

[1] Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. 2011. Flow-sensitive type recovery in linear-log time. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 483–498. doi:10.1145/2048066.2048105

[2] Lloyd Allison. 2003. Longest biased interval and longest non-negative sum interval. *Bioinformatics* 19, 10 (07 2003), 1294–1295. arXiv:https://academic.oup.com/bioinformatics/article-pdf/19/10/1294/48903824/bioinformatics_19_10_1294.pdf doi:10.1093/bioinformatics/btg135

[3] C. R. Aragon and R. G. Seidel. 1989. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (SFCS '89)*. IEEE Computer Society, USA, 540–545. doi:10.1109/SFCS.1989.63531

[4] Phil Bagwell. 2003. Fast Functional Lists. In *Implementation of Functional Languages (Lecture Notes in Computer Science, Vol. 2670)*. Springer, Berlin, Heidelberg, 34–50. doi:10.1007/3-540-44854-3_3

[5] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-Based CFG Reconstruction from Unstructured Programs. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David Schmidt (Eds.). Springer, Berlin, Heidelberg, 54–69. doi:10.1007/978-3-642-18275-4_6

[6] R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. doi:10.1145/1734663.1734671

[7] Gerth Stølting Brodal. 1998. Finger search trees with constant insertion time. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '98)*. Society for Industrial and Applied Mathematics, USA, 540–549.

[8] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios Tsakalidis, and Kostas Tsichlas. 2002. Optimal finger search trees in the pointer machine. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02)*. Association for Computing Machinery, New York, NY, USA, 583–591. doi:10.1145/509907.509991

[9] Gerth Stølting Brodal, Christos Makris, and Kostas Tsichlas. 2006. Purely Functional Worst Case Constant Time Catenable Sorted Lists. In *Algorithms – ESA 2006 (Lecture Notes in Computer Science, Vol. 4168)*, Yossi Azar and Thomas Erlebach (Eds.). Springer, Berlin, Heidelberg, 172–183. doi:10.1007/11841036_18

[10] Rudolf Fleischer. 1993. A Simple Balanced Search Tree with O(1) Worst-Case Update Time. In *Proceedings of the 4th International Symposium on Algorithms and Computation (ISAAC '93)*. Springer-Verlag, Berlin, Heidelberg, 138–146.

[11] Takeshi Fukuda, Yasuhiko Morimoto, Shinich Morishita, and Takeshi Tokuyama. 1996. Interval finding and its application to data mining. In *Algorithms and Computation*, Tetsuo Asano, Yoshihide Igarashi, Hiroshi Nagamochi, Satoru Miyano, and Subhash Suri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–64.

[12] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. 1977. A new representation for linear lists. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/800105.803395

[13] Dov Harel. 1980. *Fast updates of balanced search trees with a guaranteed time bound per update.* Technical Report. University of California Irvine.

[14] Dov Harel and George Lueker. 1979. *A data structure with movable fingers and deletions.* Technical Report. University of California Irvine.

[15] Ralf Hinze. 2002. Bootstrapping one-sided flexible arrays. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. ACM, New York, NY, USA, 2–13. doi:10.1145/581478.581480

[16] Scott Huddleston and Kurt Mehlhorn. 1982. A new data structure for representing sorted lists. *Acta Inf.* 17, 2 (June 1982), 157–184. doi:10.1007/BF00288968

[17] Haim Kaplan and Robert E. Tarjan. 1995. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing* (Las Vegas, Nevada, USA) *(STOC '95)*. Association for Computing Machinery, New York, NY, USA, 93–102. doi:10.1145/225058.225090

[18] Haim Kaplan and Robert E. Tarjan. 1996. Purely functional representations of catenable sorted lists. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 202–211. doi:10.1145/237814.237865

[19] S. Rao Kosaraju. 1981. Localized search in sorted lists. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisconsin, USA) *(STOC '81)*. Association for Computing Machinery, New York, NY, USA, 62–69. doi:10.1145/800076.802458

[20] Danny Krizanc, Pat Morin, and Michiel Smid. 2003. Range Mode and Range Median Queries on Lists and Trees. In *Algorithms and Computation (Lecture Notes in Computer Science, Vol. 2906)*, Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono (Eds.). Springer, Berlin, Heidelberg, 517–526. doi:10.1007/978-3-540-24587-2_53

[21] C. Levcopoulos and Mark H. Overmars. 1988. A balanced search tree with O(1) worst case update time. *Acta Inf.* 26, 3 (Nov. 1988), 269–277. doi:10.1007/BF00299635

[22] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. 1992. Deterministic skip lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms* (Orlando, Florida, USA) *(SODA '92)*. Society for Industrial and Applied Mathematics, USA, 367–375.

[23] Eugene W. Myers. 1983. An applicative random-access stack. *Inform. Process. Lett.* 17, 5 (Dec. 1983), 241–248. doi:10.1016/0020-0190(83)90106-0

[24] Chris Okasaki. 1995. Purely functional random-access lists. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. ACM, New York, NY, USA, 86–95. doi:10.1145/224164.224187

[25] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. 1992. Average search and update costs in skip lists. *BIT Numerical Mathematics* 32, 2 (01 Jun 1992), 316–332. doi:10.1007/BF01994884

[26] Edward Peters, Yong Qi Foo, and Michael D. Adams. 2025. *Pushing the Information-Theoretic Limits of Random Access Lists (Artifact)*. doi:10.5281/zenodo.15628635

[27] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (June 1990), 668–676. doi:10.1145/78973.78977

[28] Robert E. Tarjan and Christopher van Wyk. 1988. An O (n log log n)-time algorithm for triangulating a simple polygon. *SIAM J. Comput.* 17, 1 (Feb. 1988), 143–178. doi:10.1137/0217010

[29] Athanasios K. Tsakalidis. 1985. AVL-trees for localized search. *Information and Control* 67, 1 (1985), 173–194. doi:10.1016/S0019-9958(85)80034-6

Received 2025-02-27; accepted 2025-06-27