Master's thesis

Master's Programme in Computer Science

An Indexed, Heuristic Doubly-Linked List for Versatile Use Cases With Semi-Large Data

Rodion Efremov

September 3, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Rodion Efremov

Työn nimi — Arbetets titel — Title

An Indexed, Heuristic Doubly-Linked List for Versatile Use Cases With Semi-Large Data

Ohjaajat — Handledare — Supervisors

Dr. Jeremias O. Berg, Prof. Veli A. T. Mäkinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	September 3, 2025	60 pages, 55 appendix pages

Tiivistelmä — Referat — Abstract

The list data structures are omnipresent in modern computing, and there are at the very least several different implementations available. Some list data structures rely on arrays, others on the linked structures. Even more, some rely on balanced augmented search trees. All have their own strengths and weaknesses. For example, accessing an element of an array-based list runs in constant time, and prepending an element in linear time in any case. Meanwhile, a linked list requires a worst-case linear time to access an element and a constant time for prepending an element. In this thesis, we will present a compact, indexed, heuristic doubly-linked list data structure that runs all single-element operations in $\Theta(\sqrt{N})$ time under mild assumptions. In what follows, we will discuss the implementation details both in natural and formal manner. Also, we will provide verbose pseudo-code for all the main operations for easier translation to actual programming languages. Beyond that, we will characterize its behaviour as the function of the metric called entropy. Moreover, we will present the benchmarking evidence that supports our bold claim that the indexed list is superior on semi-large data as compared to three other customary more or less famous list data structure implementations. Finally, we summarize this thesis and draw some most relevant conclusions.

ACM Computing Classification System (CCS)
Theory of computation → Design and analysis of algorithms → Data structures design and analysis

Avainsanat — Nyckelord — Keywords

algorithms, data structures, lists, running time analysis, benchmarking

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction

List data structures are omnipresent in modern computing. Virtually any mature piece of software relies on lists in one form or another. In this thesis, we will consider three comparison list implementations, comparing them to the subject matter of this work: indexed linked list (*indexed list* for short). What comes to comparison list implementations, we consider three of them: a basic dynamic table (also known as *array list* or `dynamic array` in other contexts) [Knu97a], a linked structure called *deque* [Knu97b], and a list implementation relying on an augmented AVL-tree [AVL62].

In this thesis, we first proceed through preliminaries where we define notational conventions and asymptotic notation extensions used in this work. Also, we briefly specify the top-level structure of indexed lists. After that, we present a very important concept pertinent to indexed lists: entropy, which is a number within the range $[0, 1]$. Entropy of an indexed list communicates how equidistant the so called "fingers" are. The closer the entropy of an indexed list is to 1, the more evenly fingers are distributed over the actual internal linked list. On the contrary, the closer the entropy of an indexed list is to 0 the more tightly the fingers are distributed over the list. Obviously, we wish to keep entropy high and that is what indexed lists aim to do.

Each finger contains only two data fields: a pointer to an actual internal linked list node and its appearance index in that linked list. The more evenly/equidistantly the fingers are distributed throughout the internal linked list, the faster are the single-element operations on it run, namely in $\mathcal{O}(\sqrt{N})$ time where N is the size of the list.

What comes to the list/deque application programming interface (in short, API), we will provide 11 procedures comprising such an API. They are pushing to both ends, inserting in between, popping from both ends, deleting an element at given index, accessing an element via an index, pushing a collection to front or back, pushing a collection in between, and, finally, deleting a contiguous list range.

Speaking about the other deque data structures, we will compare our indexed list to the following list types: a dynamic table, a doubly-linked list, and an AVL-tree -based, augmented list running all single-element operations in worst-case logarithmic time. The dynamic table is implemented as an array. When we add to it an element and the table is full, we enlarge the capacity of the internal array. Normally, it shifts portions of its content in order to perform a requested operation. The main strength of a dynamic table is that it provides exact constant

time access to elements and an amortized constant time for appending an element after the tail of the dynamic table. However, dynamic tables degrade towards $\Theta(N)$ on prepending elements or inserting elements something in between.

The linked list keeps all the data as a sequence of interlinked nodes. This provides exact constant time access to or modification of both the ends in constant time unlike dynamic table. However, linked lists are rather poor on random access degrading towards $\mathcal{O}(N)$.

What comes to the augmented, AVL-tree -based list, it maintains all the elements in original order in a balanced binary search tree. The tree is sorted by element indices, not by the actual element content. While in theory it would be possible to tailor-made the bulk/collection operations on tree lists to run fast, the Java implementation `TreeList` does not support them at all. In our benchmarking, we will simulate the missing bulk operation via running corresponding single-element operations a requested number of times.

The rest of the thesis is structured as follows. In Chapter 2, we discuss the notational conventions and our own tailored convention for communicating running times of algorithms. Also, we define two metrics for analysing the relationship between the indexed linked list entropy and performance via fitting curves of polynomials of degree two: the average value of a fitting curve on entropy range $[0, 1]$ and its associated standard deviation that happens to be an adaptation of the conventional discreet standard deviation.

In Section 3, we investigate three comparison list data structures. By investigation we imply specification of working mechanics of each list data type under discussion in the section. The first one is the array-based dynamic table such as `java.util.ArrayList` in JDK and `std::vector` in C++ Standard Library. The second one under investigation is the deque data structure (essentially, doubly-linked list) such as `java.util.LinkedList` and `std::deque`. The third and last is less popular, yet it is present in the Apache Commons Collections4 library: an augmented, AVL-tree-based
`org.apache.commons.collections4.list.TreeList`.

Additionally, while not a dynamic sequence/list, we investigate the skip list.

In Section 4, we discuss the inner workings of indexed lists. First, we describe their internal data that an indexed list requires to do its work efficiently. Next, we justify the relationship between the number of so called fingers serving as an indexing data structure and the number of actual elements stored in the indexed list. Next, we discuss what finger configurations lead to poor performance (entropy near 0), and what finger configurations lead to the entropies near 1. After that, we discuss a very important technique: normalization that guarantees most often that on the access operations the fingers will be fixed in such a manner that the entropy of the indexed list won't decrease.

After the above steps, we turn our attention to natural finger configurations which get exhibited when we append elements to the end of the indexed list. Under that condition, we will see that the entropy approaches $1/2$ as the indexed list grows indefinitely. Next, we consider the running time of an indexed list as a function of both its size and entropy. Finally, we generate finger configurations in lexicographical order and we take a look how they distribute in entropy buckets.

In Section 5, we first specify the benchmark battery ran on each of the four list types (array-based, deque, AVL-tree, indexed). After that, we provide some reflection on the benchmark results. Additionally, we justify that – internally – the conventional doubly-linked list works faster than a circular doubly-linked list. Finally, in the section under question, we benchmark the indexed list against aforementioned skip list and the `java.util.TreeSet` from JDK.

Next, the bibliography follows.

After the bibliography, we have the Appendix A. In that section, we present all the 11 most important procedures and their respective supporting helper procedures. Also, as we proceed through each algorithm, we comment it with a proper rationale. We opted to present the implementation pseudo-code due to its non-triviality contrasted to its easiness at conceptional level.

In Appendix B, we discuss a so called "nearest node optimization" for the doubly-linked deques. After that, we present dynamic table expansions and contractions taking place in insert and delete operations on the tail of the dynamic table, respectively. There, we see that geometric expansion theme leads to amortized constant time for the insertion after the tail, and amortized constant time for deleting the tail element via geometric contraction scheme. Also, both arithmetic expansion scheme and arithmetic contraction scheme lead to amortized $\Theta(N)$ running time under relevant operations.

1.1 Acknowledgements

I am greatly indebted to doctor Jeremias Berg for his patience, support and insight. Also, I salute the supervising professor Veli Mäkinen. Also, I appreciate the effort of professor Alexey Stepanov for his proofreading of this thesis. Finally, I would not be here without love and care of my mother. Therefore, I dedicate this thesis to her.

2 Preliminaries

In this chapter, we will discuss all the definitions needed throughout the thesis. We begin with basic definitions and move to discussing asymptotic notation facilities used in this work. Finally, we define the concept of ***entropy*** and some statistical facilities.

2.1 Notational conventions

We set $\mathbb{N} = \{1, 2, \ldots\}$, in other words, $\mathbb{N}$ is the set of all positive non-zero integers. $\mathbb{N}_0 = \{0, 1, \ldots\}$ is the set of all non-negative integers (zero included). $\mathbb{R}$ is the set of all real numbers, $\mathbb{R}_{>0}$ is the set of positive real numbers, $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers, and we start indexing from 0, not 1. By N we denote ***the size of a list***. By the ceil operator $\lceil \cdot \rceil$ we imply the operator that rounds the argument upwards towards closest integer. The operator in question leaves the integer arguments intact. For example, $\lceil 2 \rceil = 2, \lceil e \rceil = 3, \lceil \pi \rceil = 4$. Throughout the thesis, we will adopt the following list notation: each list X is defined as the sequence of elements $X = \langle x_0, x_1, \ldots, x_{N-1} \rangle$. We also define $|X| = N$ and $||X||$ as the capacity of the internal storage array in the case of a dynamic table. (Note that we always have $N \in \{0, 1, \ldots, ||X||\}$.) Also, we will denote contiguous sub-lists using via $X\,[a, b] = \langle x_a, x_{a+1}, \ldots, x_{b-1}, x_b \rangle$ notation. For $a = b$, we write simply x_a. Note here, that $a \in \{0, \ldots, N-1\}, b \in \{a, \ldots, N-1\}$. By

$$\alpha(X) = \frac{|X|}{||X||} \in [0, 1]$$

we denote the *load factor* of the dynamic table X. We slightly defer from the convention of some computer scientific literature and refer to the dynamic table as X and not T for the sake of uniformity throughout this thesis. We say that the list X is *full* when $\alpha(X) = 1$. Also, by log we imply the binary logarithm $\log_2$. By $[a \ldots b]$ where $a, b \in \mathbb{N}_0, a \leq b$, we denote the integer range set $S = \{a, a+1, \ldots, b\}$.

2.2 Asymptotic notation

Before we proceed further, we need to define some asymptotic notations. To this end, we first need to define limit superior and limit inferior. The limit superior is defined as

$$\limsup_{n\to\infty} x_n \stackrel{\text{def}}{=} \lim_{n\to\infty}\left(\sup_{m\geq n} x_m\right) \stackrel{\text{def}}{=} \inf_{n\geq 0}\left(\sup_{m\geq n} x_m\right) \stackrel{\text{def}}{=} \inf\{\sup\{x_m : m \geq n\} : n \geq 0\},$$

and the limit inferior is defined as

$$\liminf_{n\to\infty} x_n \stackrel{\text{def}}{=} \lim_{n\to\infty}\left(\inf_{m\geq n} x_m\right) \stackrel{\text{def}}{=} \sup_{n\geq 0}\left(\inf_{m\geq n} x_m\right) \stackrel{\text{def}}{=} \sup\{\inf\{x_m : m \geq n\} : n \geq 0\}.$$

Next, we define the limit in infinity. By

$$\lim_{n\to\infty} f(n) = L \in \mathbb{R}$$

we imply that for every real $\epsilon > 0$, there exists a natural number $N \in \mathbb{N}$ such that for all $n > N$ we have $|f(n) - L| < \epsilon$. We read such a limit as "The limit of $f(n)$ as n approaches infinity equals L". Also, another type of a limit considered in this thesis is the limit

$$\lim_{n\to\infty} f(n) = \infty,$$

which implies that for each real $M > 0$, there exists an integer $N \in \mathbb{N}$ such that for each $n > N$ $f(n) > M$.

Limit superior and limit inferior come handy as the "limit in infinity" where such limit does not exist. For example,

$$\lim_{x\to\infty} \sin(x)$$

does not have a limit, yet it has both limit superior

$$\limsup_{x\to\infty} \sin(x) = 1$$

and limit inferior

$$\liminf_{x\to\infty} \sin(x) = -1.$$

Also, if $f(x) = x\sin(x)$, f does not have a limit in infinity, yet

$$\limsup_{x\to\infty} f(x) = \infty$$

and

$$\liminf_{x\to\infty} f(x) = -\infty.$$

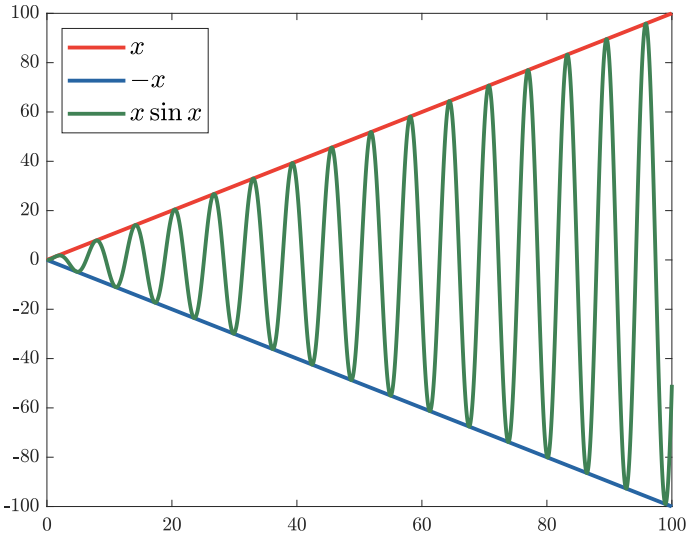


Figure 2.1: While $x\sin(x)$ has no limit in infinity, it has limit superior and limit inferior.

The graphs of the functions $x, -x$ and $x\sin(x)$ are exemplified in Figure 2.1 from which we can observe, in fact, that limit superior or limit inferior don't unconditionally require the function to have a limit in infinity, while we admit that limit superior or limit inferior may diverge.

What comes to asymptotic running time notation, by $f(n) = \Theta(g(n))$ we imply that $f(n)$ grows asymptotically as fast as $g(n)$. For example, $n^2 = \Theta(n^2)$, but $n\log n \neq \Theta(n^2)$ or $n^3 \neq \Theta(n^2)$. By $f(n) = o(g(n))$ we imply that $f(n)$ grows asymptotically slower than $g(n)$. For example, $n = o(n^2)$, $n\log n = o(n^2)$, but $n^2 \neq o(n^2)$ or $n^2\log n \neq o(n^2)$. By $f(n) = \mathcal{O}(g(n))$ we imply that either $f(n) = \Theta(g(n))$ or $f(n) = o(g(n))$. For example, $n^2 = \mathcal{O}(n^2)$ and $n\log\log n = \mathcal{O}(n^2)$. Finally, by $f(n) = \Omega(f(n))$ we imply that $f(n)$ grows asymptotically **at least as slow as** $\Theta(g(n))$. For example, $n^3 = \Omega(n^3)$ and $n^3\log n = \Omega(n^3)$, but $n^2\sqrt{n} \neq \Omega(n^3)$.

Finally, suppose we are given an algorithm $\mathcal{A}$, and suppose that its best-case running time is $\Theta(f_{\min}(n))$ and its worst-case running time is $\Theta(f_{\max}(n))$. In such a case we state that $\mathcal{A}$ runs in $\Omega(f_{\min}(n)) \cap \mathcal{O}(f_{\max}(n))$. We call this notation a ***running time range***. However, wherever possible, we will attempt to analyse best-case, average case and worst-case running time complexities.

In Table 2.1, we will summarize the definitions of the Big-O notation facilities:

Table 2.1: Two alternative definitions of each running time expression.

$f(n) = \mathcal{O}(g(n))$	$\exists c > 0\ \exists n_0\ \forall n > n_0 \colon \lvert f(n)\rvert \leq cg(n)$	$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) = \Omega(g(n))$	$\exists c > 0\ \exists n_0\ \forall n > n_0 \colon f(n) \geq cg(n)$	$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) = \Theta(g(n))$	$\exists c_1 > 0\ \exists c_2 > 0\ \exists n_0\ \forall n > n_0 \colon c_1 g(n) \leq f(n) \leq c_2 g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}_>$
$f(n) = o(g(n))$	$\forall c > 0\ \exists n_0\ \forall n > n_0 \colon \lvert f(n)\rvert < cg(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

We use the following running time conventions. We write $\Theta(g(n))$ when the relevant operation runs **always** in asymptotic $g(n)$ time. As Knuth defines, $\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$ [Knu97c]. Also, we write $\mathcal{O}(g(n))$ when the relevant operation runs in worst-case $\Theta(g(n))$ time, yet may run in $o(g(n))$. Sometimes we need to write something like, for example, $\Theta(f(n)) + \mathcal{O}(g(n)) + \Theta(h(n))$ to denote a situation where an operation requires three suboperations: one running in $\Theta(f(n))$, the second one in $\mathcal{O}(g(n))$, and the third one in $\Theta(h(n))$. Also, given $\Phi \in \{o, \mathcal{O}, \Theta, \Omega\}$, and $f_i(N, M)$ $(i \in \{1, 2, \ldots, m\})$, we write

$$\sum_{i=1}^{m} \Phi(f_i(N, M)) \stackrel{\text{def}}{=} \Phi\left(\sum_{i=1}^{m} f_i(N, M)\right).$$

By "head" of a list, we mean the first node/element in the list; by "tail" we mean the last node/element in the list.

If $g(N) = o(f(N))$, we define occasionally $\Theta(f(N) \pm g(N)) = \Theta(f(N))$. Finally, the *average running time* of an operation $W \colon \mathbb{N} \to \mathbb{N}$ is given by

$$\frac{1}{N} \sum_{i=1}^{N} W(i).$$

2.3 Indexed list

The main subject matter throughout this thesis is a simple list data structure called *indexed list*. For such a list (denote it by L), it consists of two parts: the actual doubly-linked list $L.C$ (Knuth calls doubly-linked lists *deques* [Knu97d]), and the *finger list* $L.F$. The finger list $L.F$ maintains an array $L.F.fingers$ of so called *fingers*. Each finger f consists of a reference to a node $f.node$ in $L.C$ and the appearance index $f.index$ of $f.node$ in $L.C$. All the fingers are kept ascending in $L.F.fingers$ by their $f.index$ values and no two consecutive fingers share the same index values. Also, as a slight technicality, the last finger $\mathfrak{E} = L.F.fingers[L.F.size]$

stores an *end-of-finger-list sentinel* for which $\mathfrak{E}.index = N$ and $\mathfrak{E}.node = \text{NIL}$. Finally, the number of fingers n (the value of $L.F.size$) equals $\lceil\sqrt{N}\rceil$, which is the main invariant of the indexed list.

2.3.1 Entropy

Now, as we have a rough definition of the index list, we need to define *raw entropy* of an indexed list. The raw entropy is defined as

$$\tilde{H}_n^N(f_0, \ldots, f_n) \stackrel{\text{def}}{=} 1 - \frac{1}{N}\sum_{i=0}^{n-1} |\mathfrak{f}_{i+1} - \mathfrak{f}_i - n|,$$

where $\mathfrak{f}_i \stackrel{\text{def}}{=} L.F.fingers[i].index$. In the above expression $E \stackrel{\text{def}}{=} |\mathfrak{f}_{i+1} - \mathfrak{f}_i - n|$, the term $\mathfrak{f}_{i+1} - \mathfrak{f}_i$ is the distance between two consecutive fingers. By subtracting n from that very distance we get the measure of how much the aforementioned distance deviates from n. Note here that in order to maximize the entropy, we need to minimize the E. This happens precisely when $n = \mathfrak{f}_{i+1} - \mathfrak{f}_i$ for each viable i. We can say that n is optimal when it equals to the consecutive finger distances.

We, also, subtract n from $\mathfrak{f}_{i+1} - \mathfrak{f}_i$. We will see that for some states the indexed lists may produce $\tilde{H}_n^N(f_0, \ldots, f_n) < 0$. In order to mitigate this issue, we define *effective entropy*:

$$H_n^N \stackrel{\text{def}}{=} H_n^N(f_0, \ldots, f_n) \stackrel{\text{def}}{=} \max\left\{0, \tilde{H}_n^N(f_0, \ldots, f_n)\right\}.$$

2.4 Useful definitions

In this section, we will briefly discuss definitions needed in upcoming chapters. Later we will fit data as the effective entropy varies in the range $H_n^N \in [0, 1]$. To that end, we will fit the resulting data with a polynomial of second degree $p(H) = aH^2 + bH + c$. We will consider two metrics for $p(H)$. The first is a simple mean value:

$$\mu_f \stackrel{\text{def}}{=} \frac{\int_a^b f(x)\,\mathrm{d}x}{b-a}. \tag{2.1}$$

The second function metric is an extension of standard deviation of discrete data points $x_1, \ldots, x_m$ to integrable functions:

$$\sigma_f \stackrel{\text{def}}{=} \sqrt{\frac{\int_a^b |f(x) - \mu_f|^2\,\mathrm{d}x}{b-a}}. \tag{2.2}$$

Since the domain of $p(H)$ is $[0, 1]$, the mean value of $p(H)$ on its domain is

$$\mathfrak{M}_p \stackrel{\text{def}}{=} \frac{\int_0^1 p(H)\,\mathrm{d}H}{1-0} = \frac{A}{3} + \frac{B}{2} + C. \tag{2.3}$$

In Equation 2.3, we set $a = 0$ and $b = 1$ so that $b - a = 1$. What comes to the standard deviation of $p(H)$ on $[0, 1]$, it is defined as

$$\mathfrak{S}_p \stackrel{\text{def}}{=} \sqrt{\int_0^1 |p(H) - \mathfrak{M}_p|^2\,\mathrm{d}H} = \sqrt{\frac{4A^2}{45} + \frac{AB}{6} + \frac{B^2}{12}}. \tag{2.4}$$

2.5 List operations

What comes to the list abstract data type, it supports the following operations: Search(X, i) for accessing the ith element of list X, Insert(X, i, x) for inserting the element x between the $(i-1)$st and the ith elements of X, and Delete(X, i) for deleting the ith element from X.

Additionally, the *deque* abstract data type supports the following operations: Push-Front(X, x) for putting the element x before the beginning of X, Push-Back(X, x) for putting the element x after the last element of X, Pop-Front(X) for removing and returning the first element of X, and, finally, Pop-Back(X) for removing and returning the last element of X.

Finally, we may supplement the list implementation with bulk operations:
Insert-Collection(X, i, Y) for adding the input collection Y between the $(i-1)$st and the ith elements in X, and Delete-Range(X, b, e) for removing the elements with indices $[e \ldots, b-1]$ from the list X.

While it is worthwhile to note that the bulk operations may be simulated via doing respective single-element operations sufficiently many times, we may obtain better performance by using the dedicated algorithms for those bulk operations. For this reason, we will discuss also Push-Front-Collection$(L, \mathcal{Y})$ (putting the $\mathcal{Y}$ at the front of L) and Push-Back-Collection$(L, \mathcal{Y})$ putting the $\mathcal{Y}$ right after the last element in L.

2.5.1 List operations specifications

Here, we will formally define each reasonable operation. Insert$(X, i, \hat{x})$ modifies the list $X = \langle x_0, x_1, \ldots, x_{N-1}\rangle$ to $X\,[0, i-1] \circ \hat{x} \circ X\,[i, N-1] = \langle x_0, x_1, \ldots, x_{i-1}, \hat{x}, x_i, \ldots x_{N-1}\rangle$. [1] Delete$(X, i)$ modifies the list X to $X\,[0, i-1] \circ X\,[i+1, N-1] = \langle x_0, x_1, \ldots, x_{i-1}, x_{i+1}, x_{N-1}\rangle$. The Push-Front$(X, \hat{x})$ modifies X to $\langle \hat{x}, x_0, x_1, \ldots, x_{N-1}\rangle$. The Push-Back$(X, \hat{x})$ modifies

[1] We use "$\circ$" as the list concatenation operator. For example, $\langle 1, 2, 3\rangle \circ \langle 4, 5\rangle = \langle 1, 2, 3, 4, 5\rangle$.

X to $\langle x_0, x_1, \ldots, x_{N-1}, \hat{x}\rangle$. POP-FRONT($X$) modifies X to $X\,[1, N-1] = \langle x_1, x_2, \ldots, x_{N-1}\rangle$. POP-BACK($X$) modifies X to $X\,[0, N-2] = \langle x_0, x_1, \ldots, x_{N-2}\rangle$.
PUSH-FRONT-COLLECTION($X, \mathcal{Y}$) modifies X to $\mathcal{Y} \circ X = \langle y_0, y_1, \ldots, y_{M-1}, x_0, x_1, \ldots, x_{N-1}\rangle$.
PUSH-BACK-COLLECTION($X, \mathcal{Y}$) modifies X to $X \circ \mathcal{Y} = \langle x_0, x_1, \ldots, x_{N-1}, y_0, y_1, \ldots, y_{M-1}\rangle$.
INSERT-COLLECTION($X, i, \mathcal{Y}$) modifies X to $X\,[0, i-1] \circ \mathcal{Y} \circ X\,[i, N-1] =$
$\langle x_0, x_1, \ldots, x_{i-1}, y_0, y_1, \ldots, y_{M-1}, x_i, \ldots x_{N-1}\rangle$.. Finally, DELETE-RANGE(X, b, e) modifies X to $X\,[0, b-1] \circ X\,[e, N-1] = \langle x_0, x_1, \ldots, x_{b-1}, x_e, x_{e+1}, \ldots x_{N-1}\rangle$. Finally, we call PUSH-FRONT a "*prepending operation*" and PUSH-BACK an "*appending operation*".

3 Common list structures

3.1 Dynamic table

Each dynamic table X is specified by three integers: B is the base address of the first element, c is the size of each datum in X, $N = |X|$ is the number of datums stored currently in X, and $\|X\|$ is the current capacity of X. Basically, the base address of the ith element ($i \in \{0, 1, \ldots, N-1\}$) is $B + ci$ as exemplified in Figure 3.1.

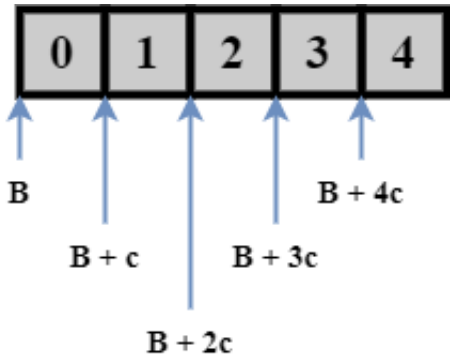


Figure 3.1: A dynamic table of size five. The numbers in the squares indicate slot indices.

3.1.1 Dynamic table operations

In this subsection, we will describe the way dynamic tables handle common list operations. Assuming that the current dynamic table is $X = \langle x_0, x_1, \ldots, x_{N-1} \rangle$, the operation SEARCH$(X, i)$ simply returns $X[i] = x_i$ in constant time, since array elements are randomly accessible via the address $B + ci$.

The operation PUSH-FRONT(X, x) first checks whether the dynamic table X can accommodate another element (in other words, whether $|X| < ||X||$). If not, a new internal array X' larger than X is created, X'`[0]` is set to x and the old contents of X is put to X' starting from index 1 in X'. Finally, the list frees the space of X and assumes X' as the current internal array.

If, however, there is space in X for x, it merely shifts all the elements in X one position towards larger indices, and inserts x at X`[0]`. Clearly, PUSH-FRONT runs in $\Theta(N)$ time in any case.

The operation PUSH-BACK(X, x) first checks that X is not full ($|X| < ||X||$). If it is full, it creates a larger internal array X', copies X to X', and inserts $X'[|X|] \leftarrow x$. Then, X is freed and X' is assumed as the current internal array. This runs clearly in linear time. However, if there is room in X, it just sets $X[N] \leftarrow x$, which runs in $\Theta(1)$ time.

At this point it is reasonable to mention that depending on how we compute the larger capacity $||X'||$ has implications on amortized running time of PUSH-BACK. We will consider two *array expansion schemes*: ***arithmetic expansion scheme*** and ***geometric expansion scheme***. The former is discussed in the Section B.2.1 and the latter in the Section B.2.2. Regardless of the scheme, when we create a new, empty dynamic table $X = \langle\rangle$, we make its capacity $||X|| = m \in \mathbb{N}$. Now, in arithmetic expansion scheme, we also choose a $d \in \mathbb{N}$, and whenever we need to expand X, we allocate X' with capacity $||X|| + d$. In geometric expansion scheme, we choose a $q \in (1, \infty)$, and whenever we need to expand, we allocate X' of capacity $||X'|| = \lfloor q||X|| \rfloor$. (As a slight technicality, we need to choose $q > 1$ such that $\lfloor qm \rfloor > m$. Otherwise, expansion will not produce larger capacity array on first expansion, and so, on none.) We discuss the way the choice of expansion scheme affects the ***amortized*** running time of the PUSH-BACK operation of the dynamic table in the Section B.2.

Next, we will discuss the POP family of operations: POP-FRONT and POP-BACK. Both of them assume that $|X| > 0$. Also, if the load factor $\alpha(X)$ drops below a certain threshold, we need to contract the array in order to not waste space. In POP-FRONT, if we need to contract the array, we create a smaller array X' using one of the two contraction schemes, and we set $X'[0, N-2] = X[1, N-1]$, and assume X' as X. Clearly, this runs in $\Theta(N)$ time. If we do not need yet to contract the array, we shift $X[1, N-1]$ to $X[0, N-2]$ and, finally, set $X[N-1] \leftarrow$ **nil**. Clearly, POP-FRONT runs in $\Theta(N)$ time in any case. What comes to POP-BACK, if we need to contract the array, all we do is create a smaller array X' using a particular contraction scheme, set $X'[0, N-2] = X[0, N-2]$, free the X and assume X' as the current internal array. If, however, we don't need to contract, all we do is setting $X[N-1] \leftarrow$ **nil**.

Just like with expansion, we consider two contraction schemes: arithmetic and geometric. In arithmetic scheme, we choose $d \in \mathbb{N}$ and whenever we need to contract X, we make its new capacity $||X|| - d$. In geometric contraction scheme, we choose $0 < q_t < q_c < 1$ and when the load factor $\alpha(X)$ of X drops below q_t, we allocate a new internal array of capacity $\lfloor q_c||X|| \rfloor$.

Operation INSERT(X, i, x) inserts x between the $(i-1)$st and the ith elements of X. In order to guarantee that both x_{i-1} and x_i are well-defined, we need to restrict i to the set $[1, \ldots, N-1]$. If, however, $i = 0$, we delegate to PUSH-FRONT. If $i = N$, we delegate to PUSH-BACK. Otherwise, we insert normally via INSERT, which works as follows: if there

is no space in X ($|X| = \|X\|, \alpha(X) = 1$), we create another array X' using either of the previously discussed expansion schemes. Next, we copy $X\,[0, i-1]$ to the beginning of X'; set $X'[i] \leftarrow x$, and copying $X\,[i, N-1]$ to $X'[i+1, N]$. Finally, we free X and assume X' as X. If, however, there is space in X, all we do is shift $X\,[i, N-1]$ to $[i+1, N]$ and set $X\,[i] \leftarrow x$. It is easy to see, that if there is no expansions involved in INSERT, the running time is $\Theta(N-i) = \mathcal{O}(N)$. If we, however, need expansion, the running time will become $\Theta(N)$ in any case.

Operation DELETE(X, i) assumes that $|X| > 0$. First, it shifts $X\,[i+1, N-1]$ to $X\,[i, N-2]$ and sets $X\,[N-1] \leftarrow$ **nil**. If the load factor $\alpha(X)$ drops below a particular threshold, we contract X in order to not waste space. If we do not need expansion, DELETE runs in $\Theta(N-i) = \mathcal{O}(N)$ time. Otherwise, it runs in $\Theta(N)$ time in any case due to the data copying.

Next, we will discuss the bulk-operations. We need them since dedicated algorithms for bulk data may run faster than their naïve counterparts that perform single-element operation on each bulk datum. If we would not opt to implementing the custom made procedures for handling bulk data, but – instead – implement them as the series of singly-element operations, we would incur a serious performance penalty. For example, suppose we want to insert the collection $\mathcal{Y} = \langle y_0, y_1, \ldots, y_{M-1} \rangle$ at the very middle of a dynamic table X. Now, for each element in $\mathcal{Y}$ we would need to shift at least $\lfloor |X|/2 \rfloor$ elements in X to the right towards larger indices which would run in $\Theta(NM)$ time instead of $\Theta(N) + \Theta(M)$ as is discussed below.

PUSH-FRONT-COLLECTION$(X, \mathcal{Y})$ puts the collection $\mathcal{Y} = \langle y_0, y_1, \ldots, y_{M-1} \rangle$ in front of X so that X becomes $\langle y_0, y_1, \ldots, y_{M-1}, x_0, x_1, \ldots, x_{N-1} \rangle$. If $||X|| < N + M$ (X has no space for the input collection), we compute the smallest $k \in \mathbb{N}$ such that $N' = \lfloor q^k \|X\| \rfloor \geq N + M, (q > 1)$. Next, we create another array X' with capacity N', copy Y to $X'[0, M-1]$, and, finally, copy X to $X'[M, M+N-1]$. However, if there is space for $\mathcal{Y}$ in X, all we do is shift $X\,[0, N-1]$ to $X\,[M, N+M-1]$ and copy $\mathcal{Y}$ to $X\,[0, M-1]$. Regardless whether we needed expansion or not, the running time of PUSH-FRONT-COLLECTION is $\Theta(N+M)$.

PUSH-BACK-COLLECTION$(X, \mathcal{Y})$ puts the input collection $\mathcal{Y}$ after the end of X. Just like in PUSH-FRONT-COLLECTION, if there is no room for $\mathcal{Y}$ in X, we compute the smallest $k \in \mathbb{N}$ such that $N' = \lfloor q^k \|X\| \rfloor \geq N + M, (q > 1)$. Next, we create another array X' with capacity N', copy X to $X'[0, N-1]$, and, finally, copy $\mathcal{Y}$ to $X'[N, N+M-1]$. If, however, there is room for $\mathcal{Y}$ in X, we just copy $\mathcal{Y}$ to $X\,[N, N+M-1]$. If we needed expansion, the running time of PUSH-BACK-COLLECTION is $\Theta(N+M)$ in any case. Otherwise, it runs in $\Theta(M)$ time in any case.

INSERT-COLLECTION$(X, i, \mathcal{Y})$ inserts $\mathcal{Y}$ between the $(i-1)$st and the ith elements of X. Just like with INSERT, in order to have x_{i-1} and x_i well defined, we must restrict $i \in [1, \ldots, N-1]$.

If $i = 0$, we delegate to PUSH-FRONT-COLLECTION; if $i = N$, we delegate to PUSH-BACK-COLLECTION. Otherwise, we proceed as follows. If there is room in X for $\mathcal{Y}$, we shift $X\,[i, N-1]$ to $X\,[i+M, N+M-1]$ and copy $\mathcal{Y}$ to $X\,[i, i+M-1]$. If there is no space for $\mathcal{Y}$ in X ($||X|| < N+M$), we compute the smallest $k \in \mathbb{N}$ such that $N' = \lfloor q^k \|X\| \rfloor \geq N+M, (q > 1)$. Next, we create another array X' with capacity N', copy $X\,[0, i-1]$ to $X'\,[0, i-1]$, copy $\mathcal{Y}$ to $X'\,[i, i+M-1]$ and copy $X\,[i, N-1]$ to $X'\,[i+M, N+M-1]$. If we needed expansion, INSERT-COLLECTION runs in $\Theta(N+M)$ time. Otherwise, it runs in $\Theta(N-i+M) = \mathcal{O}(N) + \Theta(M)$.

The last operation left to discuss is DELETE-RANGE(X, b, e). For dynamic tables, it works as follows. Let $m = e - b$ be the number of elements to delete. Assume first that contraction is not needed. Then all it does is: (1) shift $X\,[e, N-1]$ to $X\,[b, b+m-1] = X\,[b, e-1]$ and (2) sets all the slots in $X\,[N-m, N-1]$ to **nil**. The step (1) above clearly requires $N-e$ element copy operations, and the step (2) requires m settings to **nil**. Together, they run in $\Theta(N-e) + \Theta(m) = \Theta(N-e) + \Theta(e-b) = \Theta(N-b) = \mathcal{O}(N)$.

If, however, we need to contract, we create a smaller array X', copy $X\,[0, b-1]$ to $X'\,[0, b-1]$, and copy $X\,[e, N-1]$ to $X'\,[b, N-m-1]$. This runs, clearly, in $\Theta(b) + \Theta(N-e) = \Theta(N-(e-b)) = \Theta(N-m) = \mathcal{O}(N)$ time.

3.1.2 Array expansion schemes

When the internal array of a dynamic table is full, we need to expand its size. We have two expansion schemes:

Arithmetic expansion scheme Choose constant integers $m \in \mathbb{N}$ and $d \in \mathbb{N}$, where m is the initial array capacity and d is the expansion length. If, upon appending a new element (via PUSH-BACK operation), we find out that the internal array is fully filled, we expand its capacity by d array components and append the input element normally. In Section B.2.1, we prove that appending an element via arithmetic expansion scheme runs in amortized $\Theta(N)$.

Geometric expansion scheme Choose constants $q > 1$ and $m \in \mathbb{N}$. Let m denote the initial capacity of the internal array, and q be the so called expansion factor. (We require here that $\lfloor qm \rfloor > m$, or, otherwise, the first expansion won't yield a larger array, and, thus, never.) If, upon appending an element, there is no room in the internal array, we create another array X' of size $\lfloor q\|X\| \rfloor$, copy the contents of the internal array X to X', deallocate the current internal array X, and set X' as the current internal array in X, and append the element normally. In Section B.2.2, we prove that appending an

element via geometric expansion scheme runs in amortized $\Theta(1)$ time.

3.1.3 Array contraction schemes

What comes to the contraction schemes upon removing the last elements, again we have at least two of them:

Arithmetic contraction scheme Suppose $d \in \mathbb{N}$ is the contraction length and $m \in \mathbb{N}$ is the minimum allowed capacity. When there is d unoccupied array components, make the array d array components shorter. Prevent contractions when the array capacity reaches m array components. In Section B.2.3 we prove that under arithmetic contraction scheme, popping the tail runs in amortized $\Theta(N)$ time.

Geometric contraction scheme Choose $0 < q_t < q_c < 1$, $m > 0$. When the load factor $\alpha(X)$ of X drops below q_t, contract the array to $\lfloor q_c||X|| \rfloor$ array components. Don't contract when $||X||$ has already reached m. In Section B.2.4, we prove that popping the tail of a dynamic table via geometric contraction scheme runs in amortized $\Theta(1)$ time.

3.2 Linked list

Usually, linked lists come in two types: singly-linked lists and doubly-linked lists. Despite the fact that singly-linked lists require less memory, they are not very efficient on some operations. (See Table 3.1.)

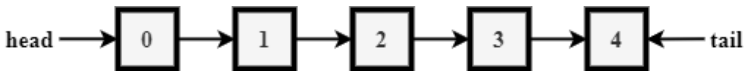


Figure 3.2: A singly-linked list of five elements. The numbers in the boxes are the indices of each storage slot. Each node has only a forward link.

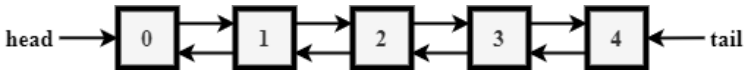


Figure 3.3: A doubly-linked list of five elements. The numbers in the boxes are the indices of each storage slot. Each node has both forward and backward links.

Table 3.1: Comparison of singly-linked and doubly-linked lists' running times.

Operation / List	Singly-linked list	Doubly-linked list
PUSH-FRONT	$\Theta(1)$	$\Theta(1)$
PUSH-BACK	$\Theta(1)$	$\Theta(1)$
INSERT(i)	$\Theta(i) = \mathcal{O}(N)$	$\Theta(\min(i, N-i)) = \mathcal{O}(N)$
GET(i)	$\Theta(i)$	$\Theta(\min(i, N-i)) = \mathcal{O}(N)$
POP-HEAD	$\Theta(1)$	$\Theta(1)$
POP-TAIL	$\Theta(N)$	$\Theta(1)$
DELETE(i)	$\Theta(i) = \mathcal{O}(N)$	$\Theta(\min(i, N-i)) = \mathcal{O}(N)$
PUSH-FRONT-COLLECTION	$\Theta(M)$	$\Theta(M)$
PUSH-BACK-COLLECTION	$\Theta(M)$	$\Theta(M)$
INSERT-COLLECTION	$\Theta(i+M) = \mathcal{O}(N) + \Theta(M)$	$\Theta(\min(i, N-i)+M) = \mathcal{O}(N) + \Theta(M)$
DELETE-RANGE(b, e)	$\Theta(b+M) = \mathcal{O}(N) + \Theta(M)$	$\Theta(\min(b, N-e)+M) = \mathcal{O}(N) + \Theta(M)$

In Table 3.1, we see that, for example, GET operation runs in $\Theta(i)$ for a singly-linked list. However, when we consider doubly-linked lists, and we wish to access the ith element of a list of size N, we can iterate towards the desired node/element i starting from the head node and proceeding towards larger indices, or, start proceeding $N - i - 1$ moves from the tail node and proceeding towards smaller indices; whichever is closer. For example, consider the following scenario: we have a list of size 10000, and we wish to access, say, 9998th element. Clearly, in a doubly-linked list, it makes no sense to start iterating forward from the head node; instead, we can do a couple of moves from the tail node along the nodes' *prev* links. (Let us call this optimization a "nearest node optimization".) Section B.1.1 shows that SEARCH in doubly-linked lists runs on average twice as fast as in singly-linked lists.

Also, in Table 3.1, it is worthwhile to discuss the running time of the DELETE-RANGE operation for both the lists. Namely, the term b in $\Theta(b+M)$ stems from the fact that we must traverse the singly-linked list b times in order to access the very first element to remove from the requested range. What comes to the doubly-linked list, the term $\min(b, N-e)$ comes from the fact that we can access either of the two: (1) the very first element in the removed range via b moves forward from the head, or (2) $N - e$ moves from the tail towards the very last element of the range to remove; whichever is closer.

3.2.1 Linked list operations

In this section, will review the main operations of a doubly-linked list. In PUSH-FRONT(X, y), we insert the input element y before the current head node. Since we keep a head node

reference around, pushing y to the front of the list runs in constant time. In PUSH-BACK(X, y), we insert the input element y after the current tail node. Just like in PUSH-FRONT, we can guarantee that pushing to the back of the list runs in constant time.

The POP family of operations assume that $|X| > 0$. POP-FRONT removes and returns the head element. Since the list keeps the reference to the head node, and that head node maintains a next link, running this operation takes constant time. POP-BACK removes and returns the tail element. Since we keep the reference to the last node in the list and the previous links, we can access the second last element in constant time, and so, we can update the tail node n to $n.prev$ in constant time as well. Clearly, POP-BACK runs in $\Theta(1)$ time.

SEARCH(X, i) merely accesses the ith element via nearest neighbour optimization and returns its satellite datum.

In INSERT(X, i, y), we insert the input element y between the $(i-1)$st and the ith elements. If $i = 0$, we delegate to PUSH-FRONT. If $i = N$, we delegate to PUSH-BACK. Otherwise, we access the $(i-1)$st element x via nearest neighbour optimization, and insert y right after the x and before $x.next$. Clearly, INSERT runs in $\Theta(\min(i, N-i)) = \mathcal{O}(N)$ time.

DELETE(X, i) removes the ith element from the list X. It merely accesses the ith node n and unlinks it from the link chain in constant time.

Just like in Section 3.1.1, it makes sense to implement custom operations for bulk operations. If not, consider what would happen if we, say, insert a collection $\mathcal{Y} = \langle y_0, y_1, \ldots, y_{M-1} \rangle$ in the *middle* of a linked list: we would need to access the nodes somewhere near the middle of the list M times. This would take roughly $\Theta(NM)$ instead of $\Theta(M) + \mathcal{O}(N)$.

PUSH-FRONT-COLLECTION iterates over all the elements in the input collection $\mathcal{Y}$ in backward order and it prepends each such element to the head of this linked list. This takes $\Theta(M)$ time. PUSH-BACK-COLLECTION iterates over all the elements in the input collection and appends each such element to the tail of the linked list. This takes $\Theta(M)$ time.

INSERT-COLLECTION$(X, i, \mathcal{Y})$ links the $\mathcal{Y}$ between the $(i-1)$st and the ith elements in X. If $i = 0$, we delegate to PUSH-FRONT-COLLECTION. If $i = N$, we delegate to PUSH-BACK-COLLECTION. If we need to delegate to PUSH-FRONT-COLLECTION or PUSH-BACK-COLLECTION, the running time is $\Theta(M)$. If not, INSERT runs in $\Theta(\min(i, N-i) + M) = \mathcal{O}(N) + \Theta(M)$.

DELETE-RANGE(X, b, e) deletes the range $X\,[b, e-1]$ from X. It runs in $\Theta(\min(b, N-e) + M) = \mathcal{O}(N) + \Theta(M)$ time.

What comes to iterator procedures, they may be easily programmed to run in constant time since adding/removing at the iterator's location is simply a matter of constant amount of link

manipulations.

3.3 Tree list

What comes to the list backed by a balanced, augmented binary tree data structure, none of the single-element operations run any worse than in logarithmic time due to balance. In this thesis, we assume that the underlying balanced binary tree implementation is an AVL-tree [AVL62]. In industry, Apache Commons Collection project implements a tree list[1]. Next, let us briefly discuss the common operations for tree lists.

PUSH-FRONT(X, x) sets x as the leftmost node in the tree. Since its height is logarithmic in N, this operation runs in $\Theta(\log N)$. PUSH-BACK(X, x) sets x as the rightmost node from the tree list. Since its height is also logarithmic in N, the running time of PUSH-BACK is $\Theta(\log N)$.

The POP family of operations assume that $|X| > 0$. POP-FRONT removes the leftmost node in the tree. Just like above, this requires $\Theta(\log N)$ time. POP-BACK removes the rightmost node in the tree. It – also – runs in $\Theta(\log N)$ time.

SEARCH starts the search from the root node and descends down the tree towards the node with the given index. Since the height of the tree is logarithmic in N and we may hit the target node before the bottom, the running time of this operation is $\mathcal{O}(\log N)$.

INSERT inserts the new element to the bottom of the tree which – clearly – runs in $\Theta(\log N)$ time. Also, we need to fix the tree balance invariant which takes $\mathcal{O}(\log N)$ time. Since $\Theta(\log N) + \mathcal{O}(\log N) = \Theta(\log N)$, inserting takes overall $\Theta(\log N)$ time.

DELETE(X, i) deletes the ith element from the list X. To this end, we need to traverse the tree down from the root node to the ith element x. If x is at the bottom of the tree and – thus – $x.left = x.right =$ **nil**, we do $\Theta(\log N)$ worth work. In such a case, also, we need to fix the tree balance starting from x and upwards up to the root node, which is also $\Theta(\log N)$. If x is not at the bottom of the tree, and thus, $x.left, x.right \neq$ **nil**, we need to access the successor of x which is guaranteed to be located at the bottom of the tree. Clearly, this case takes also $\Theta(\log N)$. Finally, if only one of the links ($x.left$, $x.right$) is set, it implies that x is already very close to the bottom: in other words, this case runs also in $\Theta(\log N)$. At this point, we conclude that DELETE runs always in $\Theta(\log N)$.

Speaking about PUSH-FRONT-COLLECTION$(X, \mathcal{Y})$, what comes to `TreeList` from the Apache

[1] https://github.com/apache/commons-collections/blob/86d1f5d0ddd0e3ab3da8d640fee5a997970c7a81/src/main/java/org/apache/commons/collections4/list/TreeList.java

Commons Collections, it does not provide the bulk operation for prepending a collection to it. Instead, it merely iterates over the elements of $\mathcal{Y}$ in reverse direction and adds each the element to the head of the tree list. Since

$$\log_2(N!) = N \log_2 N + \frac{1}{2} \log_2 N - N + 1 + \mathcal{O}(1/N)$$

by Stirling's approximation [HS24], the running time of Push-Front-Collection is

$$\begin{aligned} \Theta\left(\sum_{i=1}^{M} \log(N+i) \right) &= \Theta\left(\log \prod_{i=1}^{M} (N+i) \right) \\ &= \Theta\left(\log \frac{(N+M)!}{N!} \right) \\ &= \Theta\left(\log((N+M)!) - \log N! \right) \\ &= \Theta\left((N+M)\log(N+M) - N \log N \right). \end{aligned} \tag{3.1}$$

(Note that this operation isn't included in the public application programming interface of `TreeList`, and, thus, needs to be simulated.)

Push-Back-Collection$(X, \mathcal{Y})$ appends $\mathcal{Y}$ to the tail of the tree list. This algorithm is rather involved, and its authors claim it to run in $\Theta(M + \log(N+M))$ time. Once again, what comes to Push-Front-Collection, it could have been implemented symmetrically to run as fast as the operation under current discussion.

Insert-Collection$(X, i, \mathcal{Y})$ merely inserts the elements from $\mathcal{Y}$ to the tree list one by one. Since each new iterated element in $\mathcal{Y}$ is inserted at the bottom of the tree M times, the running time is the same as in 3.1.

Delete-Range(X, b, e), once again, simply keeps removing the bth element from the list for $e - b = M$ times. (Here, we need condition $M \leq N$.) What comes to its running time complexity, it is may be calculated in an analogous way as in Equation 3.1, yielding

$$\sum_{i=1}^{M} \log(N-i) = N \log N - (N-M)\log(N-M)$$

Also, what comes to iteration over a tree list, adding at the iterator or removing from an iterator's current location runs in $\Theta(\log N)$ time. However, the actual cost of iterating over a tree list (without removal/addition) is still $\Theta(N)$ regardless the fact that obtaining the next element in iteration order from the current one runs in $\mathcal{O}(\log N)$. This is due to the fact that each node is visited at most three times: downwards from the parent node, upwards from the

left child, and upwards from the right child. Upon each visit, we spend a constant time effort. Finally, tree list iterators support bidirectional iteration within the same time bounds.

Virtually, the only weakness of the TreeList is the fact that each node in it contains three references, two 32-bit integer `int` values and two boolean flags. The indexed list requires only three references per node and the number of fingers is $o(N)$.

3.4 Skip list

One data structure of interest is *skip list* [Pug90]. In this section we will discuss it superficially. In Section 3.4.1 we will discuss the structure of skip lists, in Section 3.4.2 we will discuss its three most fundamental operations, and – finally – in Section 3.4.3 we will discuss some uncategorised issues of skip lists.

3.4.1 Structure

A skip list is defined as a hierarchy of singly-linked lists layered on top of each other. The lowest list contains the actual data in sorted order without duplicates. We say that the aforementioned list is at the level 0. Also, we will call the list at the level 0 as *the logical container*. Other layers in the hierarchy are numbered $k = 1, 2, \ldots$ starting from the lowest level that is not the level 0. The list on level k contains some subset of the elements on level $k - 1$.

The best attainable data structure invariant for the skip list is that – at the level k – every 2^kth node appears in that very level. So, for level $k = 0$ every $2^k = 1$st element in the logical container appears. In other words, the logical container does not omit elements; it stores every element ever inserted successfully to it. On the level $k > 0$, only every 2^kth node from the logical container appears. Also, the above specification may be paraphrased as that at level $k \geq 0$ every node points to the right ***over*** $2^k - 1$ nodes in the actual logical container.

Despite the name, there are three differences between conventional lists and skip lists. First of all, the logical container of a skip list is sorted into an ascending order by some given total order relation unlike conventional lists that do not care about the order of data elements. Secondly, duplicates – unlike in conventional lists – are not allowed. It makes little sense to keep duplicates since skip lists implement the set abstract data type. Finally, skip lists are randomized and require a fixed parameter $p \in (0, 1)$ for the coin tossing whenever we need to decide whether another node at a higher level is needed.

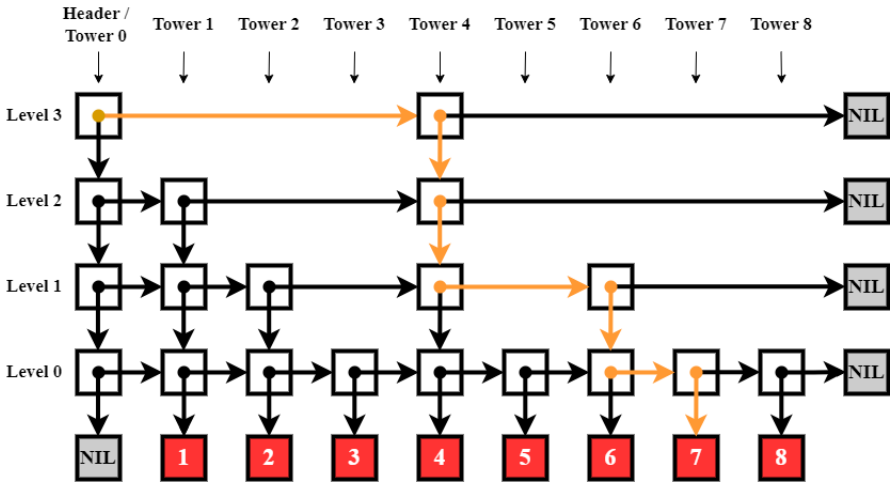


Figure 3.4: A skip list with satellite data consisting of integers $1, 2, \ldots, 8$. The orange links denote the transition path in order to access the satellite datum with value of 7. While it is only one link traversal shorter than the traversal over the corresponding logical container (8 link traversals), as the size of the skip list grows, the difference becomes more substantial.

In Figure 3.4, we see a more or less optimal skip list. The idea is that instead of iterating over the logical container, we can "skip" – thus the name – over substrings of the logical container via following links in the upper level nodes to the right as far as possible.

In Figure 3.5, we can see the so called *towers*. Basically, a tower is also a linked structure that facilitates faster element access. Each node called an *index object* in a tower retains three data fields. First, the field `node` points to the node in the logical container belonging to the same tower. Secondly, `down` points to the next index object one level lower. If the index object i is at the level 1, i.`node` = **nil**. For third, the field `right` of the index object i points to the right to the index object i' that appears most closely on the same level as the i. If the index object i residing at the level k is the rightmost tower object at the level k, i.`right` = **nil**. Finally, we note that the *height* of a tower equals the number of index objects on top of the node in the logical container. As a special case, the height of the tower without any index objects is 0 and we call such a tower *empty*. What comes to the logical container, it is implemented simply as a singly-linked list. There are only two fields in any node: the satellite datum `datum` and the forward link `next`.

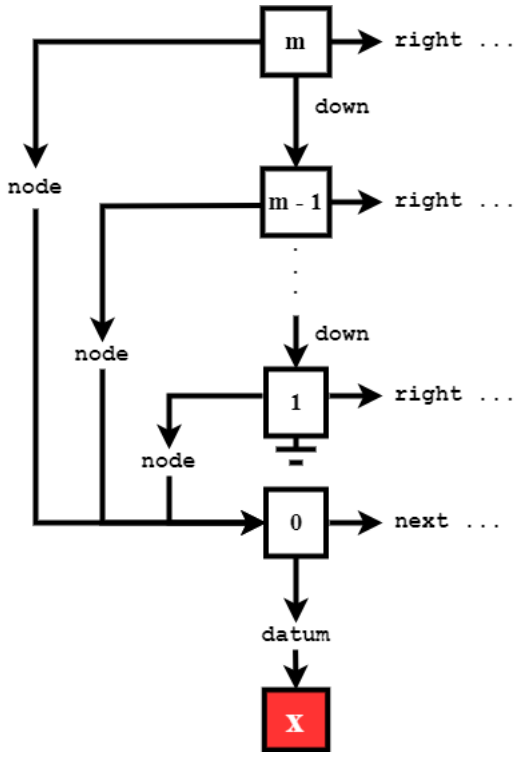


Figure 3.5: The depiction of the links in a tower. The box with the integer box 0 is *the node object* holding the pointer to the actual satellite datum. The boxes with integers $1, \ldots, m-1, m$ denote the *index objects* on top of the node object. The *height* of the tower equals m.

Again, what comes to the levels, at the level k, we would ideally have that forward links are between each 2^kth nodes. While it is not the usual case, skip lists attain fast the state in which all three fundamental operations (INSERT, CONTAINS and DELETE) run in $\mathcal{O}(\log N)$ time [**WikiSkipList**] with high probability.

3.4.2 Operations

Next, we will discuss the three aforementioned operations of the skip list: INSERT, CONTAINS and DELETE. We start from the simplest one: CONTAINS. In order to answer the question whether the input element x is in the skip list, we point to the header index object. Then, we traverse the topmost level to the right until we meet the first index object i for which $x \leq i.\texttt{node.datum}$. If we have $x = i.\texttt{node.datum}$, we have a match and we return **true**. Otherwise, we go back one step at the same, topmost level and traverse towards $i.\texttt{down}$. From there, we repeat the same operation until we reach the index object at the bottommost level of the index hierarchy (level 1). Finally, we decent to level 0 and scan the logical container to the right until we reach the first node n such that $x \leq n.\texttt{datum}$. If $n.\texttt{datum} = x$, we have a match, and, so, return **true** and **false** otherwise.

What comes to INSERT, we do essentially the same as in CONTAINS, yet if we discover already the input element x in the skip list, we fail and return **false**. Otherwise, during the search, we will end up with the node object n that should be immediately before the node with element x. Then, we simply create a new node object $\hat{n}$ and set $\hat{n}.\texttt{next} \leftarrow n.\texttt{next}$, $\hat{n}.\texttt{datum} \leftarrow x$ and $n.\texttt{next} \leftarrow \hat{n}$, which is sufficient to link in the new datum x. If insertion was successful, we might need to add some index objects on top of the tower belonging to $\hat{n}$. To this end, we keep throwing a random coin $c \in [0, 1]$. If $c \geq p$, we terminate insertion and return **true**. Otherwise, for such a coin, we add a new index object on top of the tower belonging to $\hat{n}$. When we are done with adding the index objects on top of the corresponding tower, we denote the height of the tower of $\hat{n}$ as m. Next, suppose that the height of the header tower is m_h. If $m_h < m$, we add $m - m_h$ index objects on top of the header tower. For each added tower node i at the tower 0 and the level k, we set $i.\texttt{right}$ to the corresponding tower object over node $\hat{n}$ at level k. Then, we return **true**.

Finally, we will discuss the DELETE operation. Just like CONTAINS internally, it attempts to find the presence of the input element x. If none found, it fails and returns **false**. Otherwise, unlinks the node containing x, and removes all the index objects from the tower belonging to the deleted datum, if any. Finally, for each index node i pointing via `right` directly to the one removed index objects, sets $i.\texttt{right} \leftarrow i.\texttt{right}.\texttt{right}$. After omitting the removed tower, if header node has at this point $i_h.\texttt{right} = \textbf{nil}$, unlink the i_h from the header tower and set $i_h \leftarrow i_h.\texttt{down}$ if $i_h.\texttt{down}$ is not **nil**; halt otherwise. Repeat the previous sentence as long as $i_h.\texttt{right} = \textbf{nil}$.

3.4.3 Miscellany

What comes to the other work on skip lists, Munro et al. presented a deterministic version of skip lists [MPS92]. They achieved that result by relaxing the skip list invariant, according to which, at the level k ($k = 0, 1, \ldots$) we should add `next` links at every 2^k element.

At this point, we need to state that in Pugh's implementation of skip lists, the towers are implemented as arrays, and the version of skip lists with linked towers relies on [Git19] which are based on work of [**cslm-article-harris**] and [Mic02]. The Pugh's arrangement has a weakness: a tower need to be expanded via copying the current tower in order to add an index object on top of it.

While it is very unlikely due to the randomization, in principle, a skip list may degrade in a sense that all levels are as dense as the logical container as is exemplified in Figure 3.6.

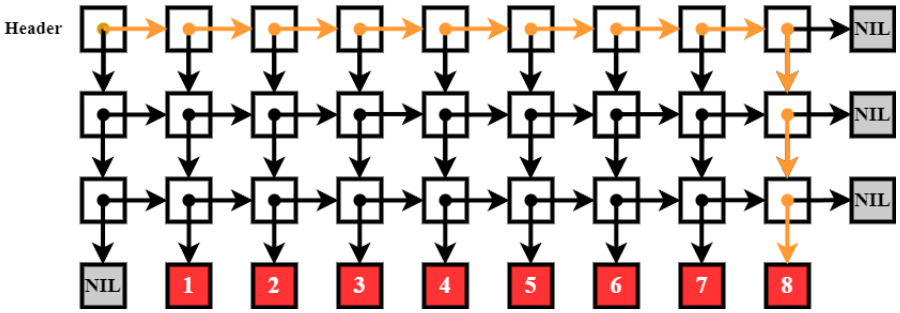


Figure 3.6: The worst-case skip list over eight datum elements. All the levels are as dense as the logical container.

We can see from the worst-case skip list that its operations would run in $\mathcal{O}(N) + \Theta(\log N)$ and the total space complexity will grow up until $\Theta(N \log N)$. This stems from the fact that while any tower height is possible in theory with very low probabilities, the expected height is logarithmic in the size of the skip list. We, however, do not justify this claim formally in this thesis.

On the other hand, access and modification within $\mathcal{O}(\log N)$ is guaranteed on near-optimal skip lists as can become evident from Figure 3.7.

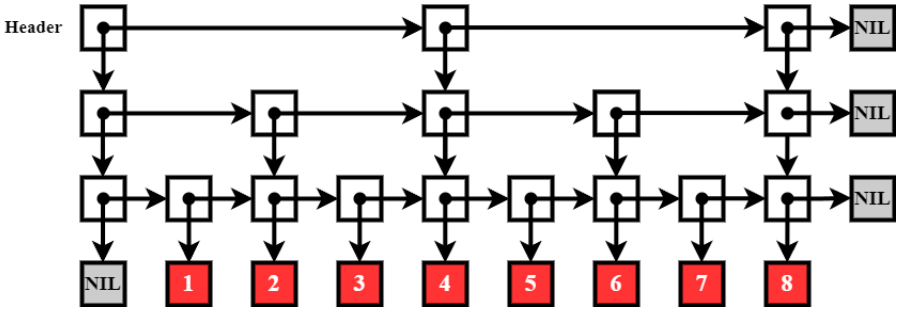


Figure 3.7: An optimal skip list over eight datum elements. At the level 1, the list proceeds over every second element in the logical container, and over every three elements at the topmost 2nd level. The height is logarithmic in N.

4 Indexed list internals

In this section, we will describe the structure of the indexed list. After that, we shall review reasoning behind the data structure.

4.1 The data structure

Each indexed list L in question consists of two parts: a doubly-linked list $L.C$, and a ***finger list*** $L.F$. The actual doubly-linked list $L.C$ has three members: $L.C.head$ pointing to the head node of the list, $L.C.tail$ pointing to the tail node of the list, and $L.C.size$ caching the number of nodes in $L.C$, which is, effectively, the size of the list. Each node u in $L.C$ contains also three members: the actual value $u.datum$, the pointer to the previous node $u.prev$, and the pointer to the next node $u.next$. We note that $L.C$ is not circular.

What comes to the finger list $L.F$, it consists of an array $L.F.fingers$ of n fingers $f_0, f_1, \ldots, f_{n-1}$, where each finger f_i contains two member fields: the pointer $f_i.node$ to a node u in $L.C$, and the **appearance index** $f_i.index$ of u in $L.C$. Also, by $|L.F| = n$ we store the number of fingers in the finger list $L.F.fingers$, and by $||L.F||$ we denote the capacity of $L.F.fingers$. Note that we have here an invariant $|L.F| + 1 \leq ||L.F||$. The fingers in $L.F.fingers$ are kept sorted by finger indices, which allows faster access via binary search over the sequential scan. The finger indexing starts from 0, so the fingers are $f_0, f_1, \ldots, f_{n-1}$.

We rely on an additional arrangement: there is an end-of-finger-list sentinel finger $L.F.\mathfrak{E}$ in $L.F.fingers$. It appears right after f_{n-1} in $L.F.fingers$ and has $L.F.\mathfrak{E}.node =$ **nil** and $L.F.\mathfrak{E}.index = L.C.size$. We use it in order to allow the binary search over the finger list to process those nodes that appear in the list **after** the node pointed by f_{n-1}.

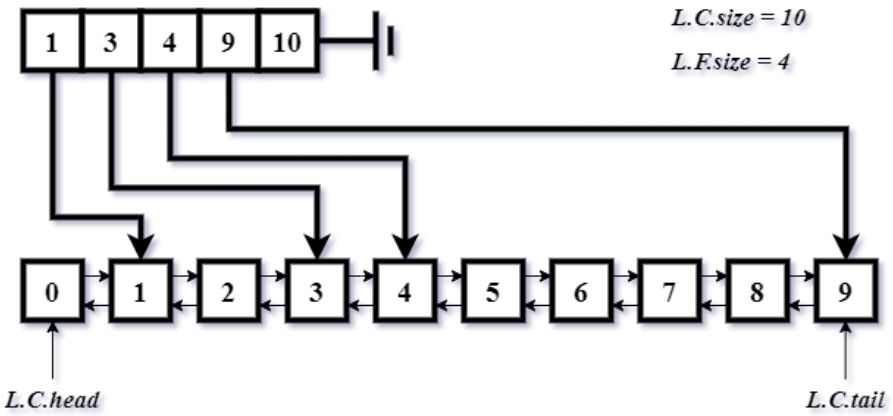


Figure 4.1: The indexed list with parameters $N = 10, n = 4$. Note that the index of the fifth finger equals to N and its link is unused. This is the end-of-finger-list sentinel finger $L.F.\mathfrak{E}$; $L.F.\mathfrak{E}.index = L.C.size$ holds always.

Accessing an element at index i in L runs in two steps:

- search $L.F.fingers$ for the finger f closest to node with index i,
- traverse a portion of $L.C$ starting from $f.node$ until the element with the index i is reached.

Assuming that the fingers are distributed as evenly as possible, each finger covers N/n nodes. Suppose the number of fingers is n, and the size of the list is N. Then, the work for performing a single-element operation is

$$W_N(n) = \frac{N}{2n} + n.$$

Above, we divide by 2 since we can choose the closest finger, and, so, we need to traverse at most $\lfloor n/2 \rfloor$ nodes in $L.C$, where n also is the distance between two fingers that surround the target node. Also, we add n in order to account for manipulating the finger list. Next, we fix N, and we wish to find such an n that minimizes $W_N(n)$. Therefore, we must have

$$\frac{\mathrm{d}}{\mathrm{d}n} W_N(n) = \frac{\mathrm{d}}{\mathrm{d}n}\left(\frac{N}{2n} + n\right) = -\frac{N}{2n^2} + 1.$$

Next, we need to find such an n_0 that

$$\frac{\mathrm{d}}{\mathrm{d}n} W_N(n_0) = 0.$$

This happens when $n_0 = \sqrt{N/2}$. Since we work with integers, we set $n = \lceil \sqrt{N/2} \rceil$. Actually, in our implementation of the indexed list, we set $n = \lceil \sqrt{N} \rceil$, since dividing by $\sqrt{2}$ in every operation introduces additional computational overhead without any improvement in asymptotic sense.

4.1.1 Entropy of indexed list

Perhaps the most important concept in analysing the asymptotic behaviour of the indexed list is that of ***entropy***. The **raw entropy** of the indexed list with fingers $f_0, f_1, \ldots, f_{n-1}$ is defined as

$$\tilde{H}^N(f_0, \ldots, f_n) = \frac{1}{N} \sum_{i=0}^{n-1} |f_{i+1}.index - f_i.index - n|,$$

where $f_n = L.F.\mathfrak{E}$. Suppose $f_i.index = t + si, s \in [1 \ldots n]$, $i \in [0 \ldots n-1]$ and $t \in [0 \ldots (N - si - 1)]$ (this implies that $t + si < N$ for all $i \in [0 \ldots n-1]$). Basically, t is the node index of the leftmost finger ($f_0.index$), s is the distance between two consecutive fingers f_{j-1} and f_j, namely, $s = f_j.index - f_{j-1}.index$, and i is the index of the finger f_i in question. Also, we have that $f_n.index = N$. At this point, it is essential to note that the term $\delta = |f_n.index - f_{n-1}.index - n|$ won't necessarily equal to all other values $\Delta = |f_i.index - f_{i-1}.index - n|, i \in [0 \ldots (n-1)]$. In order to simplify calculation of the entropy as a function of s, we must strengthen our assumptions and set the triplet s, t, n such that $t + sn = N$. In such a case, the raw entropy will equal

$$\begin{aligned}
\tilde{H}^N(f_0, \ldots, f_n) &= 1 - \frac{1}{N} \sum_{i=0}^{n-1} \Big| f_{i+1}.index - f_i.index - n \Big| \\
&= 1 - \frac{1}{N} \sum_{i=0}^{n-1} \Big| t + s(i+1) - (t + si) - n \Big| \\
&= 1 - \frac{1}{N} \sum_{i=0}^{n-1} \Big(n - s \Big) \\
&= 1 - \frac{1}{N} n \Big(n - s \Big).
\end{aligned}$$

It is evident that $\tilde{H}^N(f_0, \ldots, f_n) = 1$ when $s = n$. Now, let us investigate what happens when we pack all the fingers into a contiguous block, that is, when $s = 1$:

$$\tilde{H}^N(f_0, \ldots, f_n) = 1 - \frac{1}{N} n(n-1) \approx \frac{1}{n}.$$

Clearly,

$$\lim_{n \to \infty} \frac{1}{n} = 0,$$

and so, we can deduce that – at least – $\tilde{H}^N(f_0, \ldots, f_n) \in (0, 1]$.

Actually, there exist finger configurations that lead to $\tilde{H}^N(f_0, \ldots, f_n) < 0$. Consider the following scenario. Let $n > 1$. Set $n_l > 0$ and $n_r > 0$ such that $n_l + n_r = n$. Pack n_l fingers at the very beginning of the indexed list, and n_r fingers at the very end. Now we have:

$$f_i.index = \begin{cases} i & \text{if } 0 \leq i < n_l, \\ N - n + i & \text{if } n_l \leq i < n. \end{cases}$$

Now, let us calculate the entropy of the index list with the finger configuration stated above. We use the short cut $f_i \stackrel{\text{def}}{=} f_i.index$ non-recursively.

$$\begin{aligned}
\tilde{H}^N(f_0,\ldots,f_n) &= 1 - \frac{1}{N}\sum_{i=0}^{n-1}\left|f_{i+1} - f_i - n\right| \\
&= 1 - \frac{1}{N}\sum_{i=0}^{n_l-2}\left|f_{i+1} - f_i - n\right| \\
&\quad - \frac{1}{N}\left|f_{n_l} - f_{n_l-1} - n\right| \\
&\quad - \frac{1}{N}\sum_{i=n_l}^{n-1}\left|f_{i+1} - f_i - n\right| \\
&= 1 - \frac{1}{N}\sum_{i=0}^{n_l-2}(n-1) \\
&\quad - \frac{1}{N}\left|\overbrace{N-n+n_l}^{f_{n_l}} - \overbrace{(n_l-1)}^{f_{n_l-1}} - n\right| \\
&\quad - \frac{1}{N}\sum_{i=n_l}^{n-1}\left|(N-n+i+1) - (N-n+i) - n\right| \\
&= 1 - \frac{1}{N}(n_l-1)(n-1) \\
&\quad - \frac{1}{N}\left|N-2n+1\right| \\
&\quad - \frac{1}{N}(n-n_l)(n-1) \\
&= 1 - \frac{1}{N}(n-1)(n-1) - \frac{1}{N}(N-2n+1) \\
&\approx 1 - \frac{n^2-2n+1}{n^2} - \frac{n^2-2n+1}{n^2} \\
&= -1 + 4/n - 2/n^2.
\end{aligned}$$

Now, it is evident that $\lim_{n\to\infty}\tilde{H}^N(f_0,\ldots,f_n) = -1$. Since in both cases the first single-element operation would run on average in $\Theta(N)$, we conclude that both of them are as bad as it gets. So, we define ***effective entropy*** $H^N(f_0,\ldots,f_n)$ (note there is no tilde symbol $\sim$ on top of H):

$$H^N(f_0,\ldots,f_n) = \max\left\{0, \tilde{H}^N(f_0,\ldots,f_n)\right\}.$$

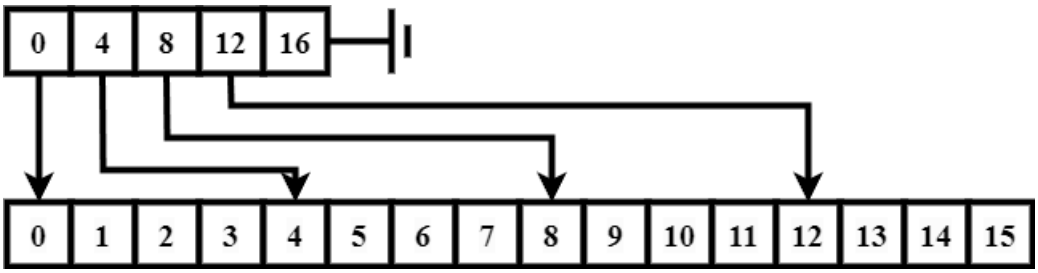


Figure 4.2: An indexed list with optimal finger configuration.

We may observe from Figure 4.2 that the indexed list L attains the entropy of 1 when its fingers are equidistant. This does not quite hold when n does not divide N evenly, yet we won't delve into implications of this assumption. The finger list in Figure 4.2 has entropy of 1. Note that all fingers are equidistant and have four hops between consecutive fingers. Such a list requires at most two hops from the closest finger in order to reach the desired element node.

What comes to poor finger configurations, one such is exemplified in Figure 4.3.

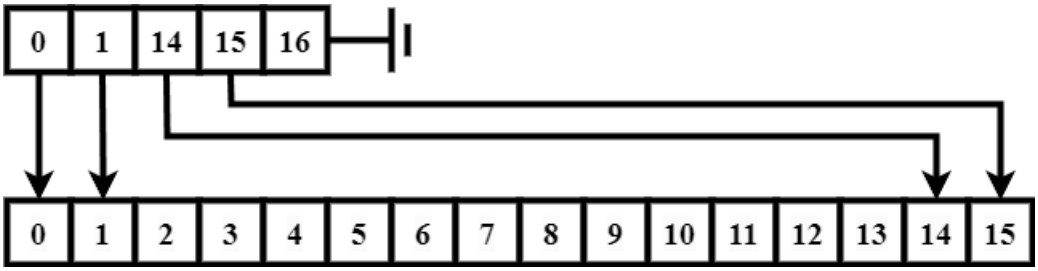


Figure 4.3: The finger list with one of the least optimal finger configurations. Here, we have $n_l = n_r = 2$.

The finger list in Figure 4.3 has entropy of -0.125 which is the minimum possible over all finger configurations of indexed list with size 16. Basically, the first single-element operation will run in $\Theta(N)$ average time. Also, the following three finger configurations attain the raw entropy of -0.125: `[0, 1, 2, 3, 16]`, `[0, 1, 2, 15, 16]` and `[0, 13, 14, 15, 16]`.

Next, we will justify why we keep fingers sorted by their respective indices. If we have opted to the unsorted finger layout, we could have added or removed a finger in constant time. However, under such arrangement, accessing a particular finger would have run in $\Theta(n) = \Theta(\sqrt{N})$ time. If we have assumed that the fingers are sorted by indices, the finger access would have run in $\Theta(\log \sqrt{N}) = \Theta(\log N) = o(\sqrt{N})$. Basically, if we assume that the read operations are much more frequent than the write operations, it is more reasonable to keep the finger sorted by indices.

Finally, we briefly present an observation that the average work effort for an indexed list reduces with growing entropy. The effort in question is computed as the sum of constant time

steps needed to perform the operation. Here, we note that we have implemented a simulation program that computes the effort values. Refer to Figures 4.4 and 4.5.

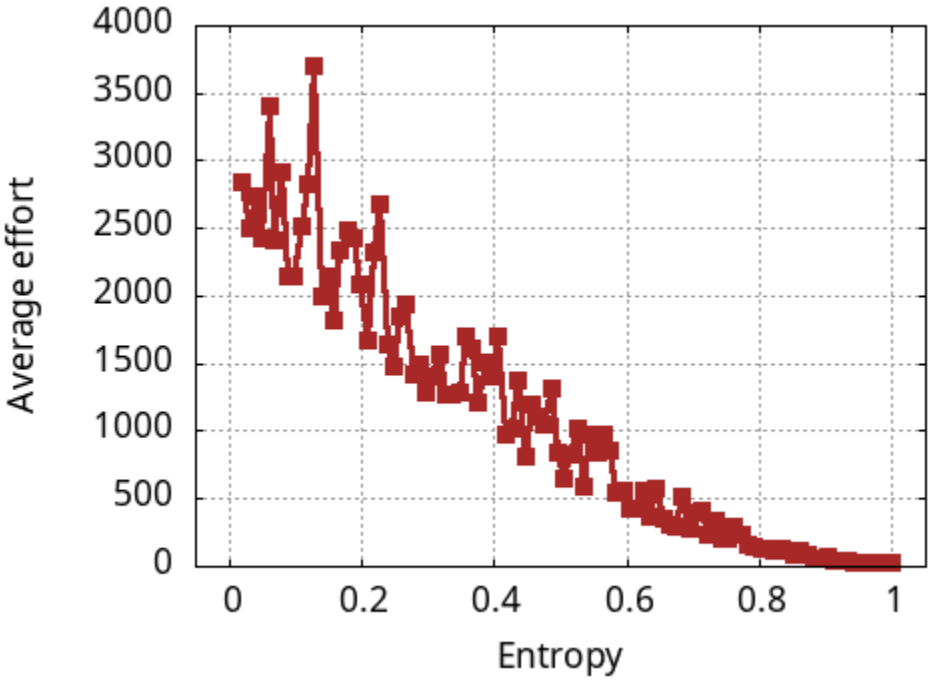


Figure 4.4: Average access effort while entropy is rising.

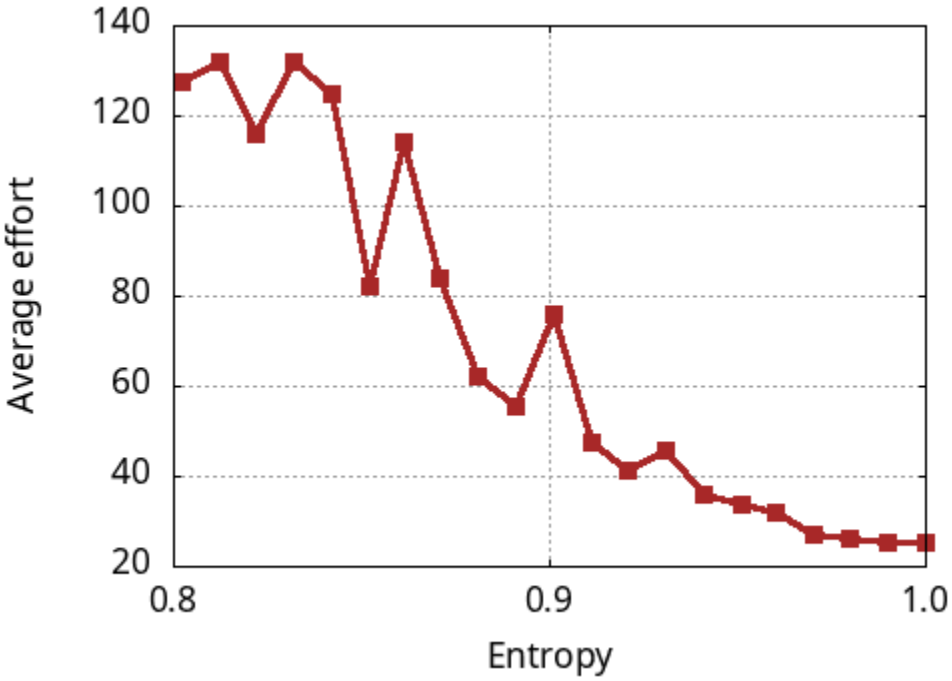


Figure 4.5: The tail of the above plot for entropy values of $[0.8, 1]$.

The Pearson correlation coefficient of the above two plots is $r \approx -0,9427$, so we can see

that the correlation is strongly positive between entropy and efficiency. What comes to data gathered for the two plots, we have run the following simulation[1]. What comes to that simulation, it produced a data file accessible at a Gist[2].

We have generated 100 simulations in total. Each simulation assumes a list of $N = 10000$ elements and – thus – $n = 100$ fingers. For each simulation, we generated $s \in [1 \ldots n]$. For each s, we have created a random $t \in [0 \ldots N - bn - 1]$. For s and t we have created a finger configuration $f_i = t + is$ for $i \in [0 \ldots n - 1]$.

Next, for each simulation, we iterated over all $j \in [0 \ldots N - 1]$, and for each j, we accessed the *closest* finger f_i and we add $|f_i.index - j|$ to a counter c that starts from 0. Finally, for that simulation run, we return a data point consisting of the current indexed list entropy and c/N. At this point, we note that we don't count the effort needed to access the closest finger since that operation is logarithmic in N via binary search and so, is negligible.

After collecting statistics for all the hundred simulation, the lowest Pearson correlation coefficient was -0.9427, the highest Pearson correlation coefficient was -0.8951. What comes to the mean and standard deviation of all the data sets, they are -0.9191 and 0.0098, respectively.

4.1.2 Indexed list normalization

One important aspect of indexed list is that it moves the fingers as it is being worked on in hope to increase entropy. The idea is that whenever we access a node in $L.C$, we rearrange fingers in $L.F$ such that – at least – the entropy does not decrease. First of all, the list normalization appears only if $n \geq 3$. If that is not the case, accessing an element runs in sequential manner akin to ordinary linked list. Otherwise, for the access index $i \in [\ldots N-1]$, we obtain the finger index i_f such that $i \leq L.F.fingers[i_f].index$ and the i_fth finger is a closest such finger. Then, we need to consider three different cases. The first case appears when $i_f = 0$. In such a case, we take two leftmost fingers f_0, f_1 and we move the finger f_0 to point to the **middle** of the range between the beginning of the list and f_1. The second case appears when $i_f = n - 1, n$. There, we take two last fingers f_{n-2}, f_{n-1} and we move f_{n-1} to point to the middle of the range between f_{n-2} and the tail of the list. Finally, the only remaining case is any case that does not fall into one of the aforementioned cases. This time, we consider three consecutive fingers f_{i_f-1}, f_{i_f} and f_{i_f+1}, and we simply move f_{i_f} to point between f_{i_f-1} and f_{i_f+1}.

Next, we will investigate how normalization affects entropy. To begin, we consider the

[1] https://github.com/coderodde/IndexedLinkedListEntropy
[2] https://tinyurl.com/3n6w47w4

following two metrics:

$$W_1^n = |b - a - n| + |c - b - n|,$$
$$W_2^n(m) = |m - a - n| + |c - m - n|.$$

The upper equation defines the partial entropy factor and the lower one defines the **parametrized partial entropy factor**. Now, we need a simple theorem:

Theorem 1. *Suppose we are given three consecutive fingers with indices $a \in \mathbb{N}_0, b, c \in \mathbb{N}$, $(0 \leq a < b < c \leq N)$. Also, suppose we are given a finger list of size n.*

Now, we claim that setting $m = (a+c)/2$ minimizes $W_2^n(m)$ and so we have that if we rewind the finger with the index b to m, the parametrized partial entropy factor minimizes, which, in turn, makes sure that the entropy of the entire indexed list does not decrease.

Proof. Let us set $A = a + n$ and $B = c - n$. Now,

$$W_2^n(m) = |m - A| + |B - m|.$$

Next, we need to derivate $W_2^n(m)$ with respect to m:

$$\frac{\mathrm{d}}{\mathrm{d}m} W_2^n(m) = \operatorname{sgn}(m - A) + \operatorname{sgn}(m - B).$$

Next, we need to find the root of $\mathrm{d}W_2^n(m)/\mathrm{d}m$:

$$\operatorname{sgn}(m - A) + \operatorname{sgn}(m - B) = 0$$
$$m = \frac{a + c}{2}.$$

Finally, we conclude that $W_2^n(m) = W_2^n((a + c)/2) \leq W_1^n$. □

We have conducted a simulation[1] that proves that average cost of accessing a node in the indexed list is minimized when $m = (a + c)/2$. This can be proved by a simple argument. Let

$$W_{a,m,b}(i) = \begin{cases} \min(i - a, m - i) & \text{if } i \text{ in } [a \ldots m], \\ \min(b - i, i - m) & \text{if } i \text{ in } [m + 1 \ldots b]. \end{cases}$$

be the effort to access the ith element in the index list with finger indices a, m, b $(a < m < b)$. Next, we define the average work of accessing a node in the list as

$$A(a; m; b) = \frac{1}{b - a + 1} \sum_{i=a}^{b} W_{a,m,b}(i).$$

The simulation proves that $A(a, (a + b)/2, b) \leq A(a, m, b)$ for any $m \in [a + 1 \ldots b - 1]$.

[1] https://tinyurl.com/9983jfaa

4.2 On natural finger configuration

By ***natural finger configuration*** we imply the finger configuration that stems from only appending elements to the tail of the list – and thus, appending fingers to the end of the finger list – and not running single-element operations that may alter the finger indices. For instance, for $n = 6$ the natural finger configuration will be `[0, 1, 4, 9, 16, 25]`.

First, we will prove that in the natural finger configuration the distance between two consecutive fingers is $\Theta(\sqrt{N})$. Secondly, we will prove that in the natural finger configuration, the entropy approaches $1/2$ as $n \to \infty$.

The following theorem proves that the distance of consecutive fingers is $\Theta(\sqrt{N})$.

Theorem 2. *As elements are being appended to an indexed list L (and so, fingers are being appended to $L.F.fingers$), the fingers remain evenly distributed in asymptotic sense.*

Proof. Suppose two last fingers f_{n-2}, f_{n-1} are given. Let $f_{n-2}.index = (I-1)^2$, $f_{n-1}.index = I^2$. Now we have

$$\begin{aligned} f_{n-1}.index - f_{n-2}.index &= I^2 - (I-1)^2 \\ &= I^2 - (I^2 - 2I + 1) \\ &= 2I - 1 \\ &= 2\left\lceil\sqrt{N}\right\rceil - 1 \qquad \text{By the data structure invariant.} \\ &= \Theta(\sqrt{N}). \end{aligned}$$

□

Also, the following theorem shows that the entropy of the finger list with natural finger configuration approaches $1/2$ as the number of fingers approaches infinity.

Theorem 3. *Suppose H_n is the entropy of the finger list of length n with natural finger configuration. We must have*

$$\lim_{n\to\infty} H_n = \lim_{n\to\infty}\left(1 - \frac{1}{n^2}\sum_{i=1}^{n}|2i-1-n|\right) = 1 - \overbrace{\lim_{n\to\infty}\left(\frac{1}{n^2}\sum_{i=1}^{n}|2i-1-n|\right)}^{l} = \frac{1}{2}.$$

Proof. Now, we need to prove that $l = 1/2$. To begin, via a routine calculation, we can derive the following:

$$\sum_{i=1}^{n}|2i-1-n| = 2\left(-\left\lceil\frac{n+1}{2}\right\rceil + n + 1\right)\left(\left\lceil\frac{n+1}{2}\right\rceil - 1\right) \approx \frac{n^2-1}{2}.$$

Above, the first equation is by WolframAlpha[1]. Finally,

$$l = \lim_{n\to\infty}\left(\frac{1}{n^2}\frac{n^2-1}{2}\right) = \lim_{n\to\infty}\left(\frac{1}{2}-\frac{1}{2n^2}\right) = \frac{1}{2}.$$

as expected. □

4.3 Running time analysis

In this section, we will attempt to derive via experimentation the running time expressions that take indexed list effective entropies into account. To begin with, we used a Java program `IndexedLinkedListEntropy`[2] in order to generate 100 ***raw*** data sets $\tilde{D}^k$, $k \in \{1,\ldots,100\}$, accessible in a Gist[3]. Each raw data set $\tilde{D}^k$ consists of 100 data lines (H_i^k, w_i^k), where H_i^k is the ith entropy of the data set $\tilde{D}^k$ and w_i^k is the work effort associated with H_i^k. Next, we will present the four ***running time schemes***: $g_1(n,H), g_2(n,H), g_3(n,H;\rho), g_4(n,H;\gamma)$. The first running time scheme is called *simple running time scheme* and it is defined as

$$g_1(n,H) = n^{2-H}.$$

The second running time scheme is called *partial running time scheme* and it is defined as

$$g_2(n,H) = n^{\tilde{g}_2(n,H)},$$

where

$$\tilde{g}_2(n,H) = 2 - H + \log_n(1 - 0.5\cos^2(\pi H)).$$

The third running time scheme is called *verbose partial running time scheme* and it is defined as

$$g_3(n,H;\rho) = n^{\tilde{g}_3(n,H;\rho)},$$

where

$$\tilde{g}_3(n,H;\rho) = 2 - H + \log_n(1 - 0.5\cos^2(\pi H)) + \log_n(1 - 0.6\times 2^{\rho}|H-0.5|^{\rho}).$$

Finally, the fourth running time scheme is called *semi-verbose running time scheme* and it is defined as

$$g_4(n,H;\gamma) = n^{\tilde{g}_4(n,H;\gamma)},$$

where

$$\tilde{g}_4(n,H;\gamma) = 2 - H + \log_n(1 - \gamma\cos^2(\pi H)).$$

[1] https://tinyurl.com/39vj836b
[2] https://github.com/coderodde/IndexedLinkedListEntropy
[3] https://tinyurl.com/3n6w47w4

Note that g_4 is the generalization of g_2: $g_2(n, H) = g_4(n, H; 0.5)$. For each such $\tilde{D}^k$, we have normalized it to $D^{k,g}$ via setting for each $i \in \{1, \ldots, 100\}$ in $D^{k,\tilde{g}}$ $(H_i^k, R_i^{k,\tilde{g}})$ instead of (H_i^k, w_i^k), where

$$R_i^{k,\tilde{g}} = \frac{w_i^k}{n^{\tilde{g}(n,H)}}.$$

Above, $\tilde{g}$ is one of the four running time schemes. Also, we will consider simple data set mean which is defined as

$$\bar{R}^{k,g} = \frac{1}{100} \sum_{i=1}^{100} R_i^{k,g},$$

and a simple data set standard deviation

$$\sigma^{k,g} = \sqrt{\frac{\sum_{i=1}^{100} (R_i^{k,g} - \bar{R}^{k,g})^2}{100}} = \frac{1}{10} \sqrt{\sum_{i=1}^{100} (R_i^{k,g} - \bar{R}^{k,g})^2}.$$

Next, for each data set $D^{k,g}$, we aim to produce a ***fitting curve*** $p^{k,g}(H) = A^{k,g} H^2 + B^{k,g} H + C^{k,g}$, which is a simple polynomial of degree two and a function of entropy H. Note that g is a running time scheme out of all four possible running time schemes $\{g_1, g_2, g_3, g_4\}$. For each such given $p^{k,g}(H)$, we wish to calculate three metrics: the mean of $p^{k,g}$ on the interval $[0, 1]$ $\mathfrak{M}(p^{k,g})$ as specified in Equation 2.3, the continuous version of the standard deviation on the same interval $[0, 1]$ $\mathfrak{S}(p^{k,g})$ as specified in Equation 2.4, and – finally – the mean distance between the fitting curve and a data point

$$\mathfrak{D}(p^{k,g}) = \frac{1}{100} \sum_{i=1}^{100} |p^{k,g}(H_i^k) - R_i^{k,g}|.$$

Next, we will discuss each of them. $g_1(n, H)$ is the easiest. When $H = 1$, $g_1(n, 1) = n = \Theta(\sqrt{N})$. When $H = 0$, $g_1(n, 0) = n^2 = \Theta(N)$, and so we have that all single-element operations run in $\Omega(\sqrt{N}) \cap \mathcal{O}(N)$ time.

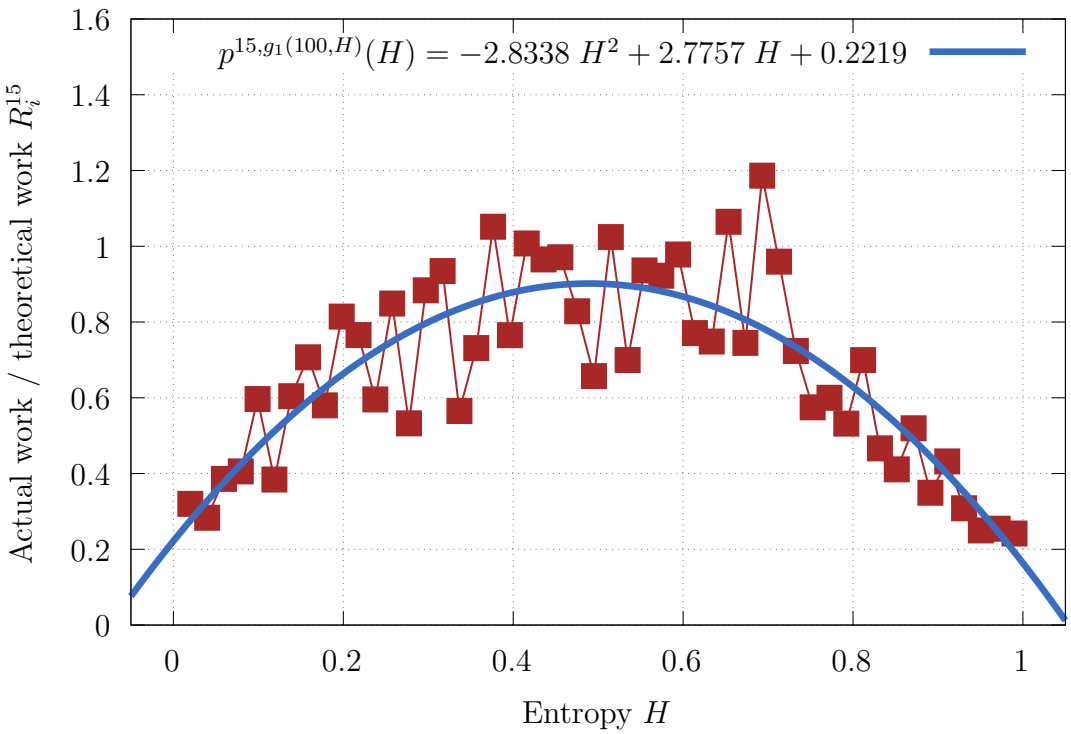


Figure 4.6: Data set $\tilde{D}^{15}$ normalized by $g_1(100, H)$ producing the normalized data set D^{15,g_1}.

In Figure 4.6 we see that most values $R_i^{k,g}$ are within the range $[0.2, 1.0]$ and so we can conclude that g_1 overestimates itself, especially at the entropy extremities $H = 0$ and $H = 1$: near $H = 0$, the ratio R_i^{15} is around 0.3, and near $H = 1$ it is around 0.2. The fitting curve supports those two observations. What comes to plot statistics, the mean $\bar{R}^{15}$ coordinate is roughly 0.6690, the standard deviation is $\sigma^{15} \approx 0.2541$, the mean value of the fitting polynomial p^{15,g_1} is $\mathfrak{M}(p^{15,g_1}) \approx 0.6652$, the standard deviation of the very same fitting polynomial is $\mathfrak{S}(p^{15,g_1}) \approx 0.2119$, and, finally, the average distance $\mathfrak{D}(p^{15,g_1}) \approx 0.1177$. From all those statistics it becomes apparent that we wish to lower the values of $g_1(n, H)$ in order to raise $\mathfrak{M}(p^{15,g_1})$ and $\bar{R}^{15,g_1}$ near the extremities $H = 0$ and $H = 1$. To this end, we consider the partial running time scheme $g_2(n, H)$.

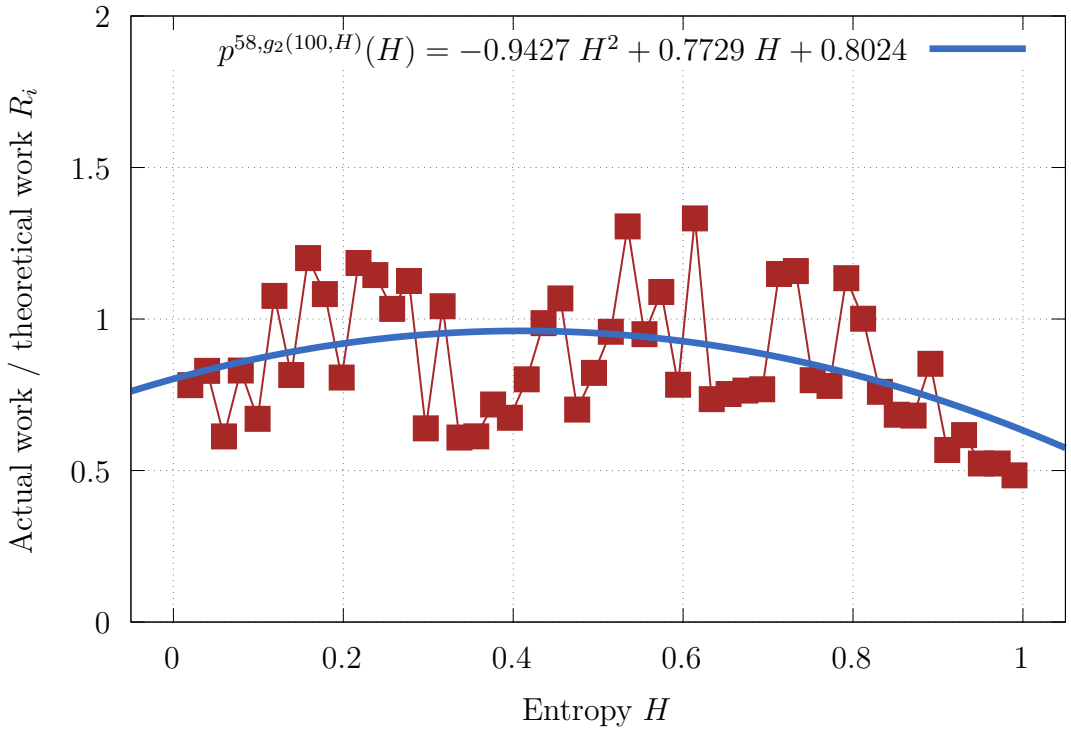


Figure 4.7: Data set $\tilde{D}^{58}$ normalized by $g_2(100, H)$ producing the normalized data set D^{58}.

In Figure 4.7, the theoretical work is given by $g_2(100, H)$. The mean y-coordinate – that is, actual to theoretical work ratio R_i^k – is $\bar{y} \approx 0.8745$ and the standard deviation is $\sigma \approx 0.2102$. The mean value of the fitting polynomial p is $\mathfrak{M}_p \approx 0.8747$, the standard deviation of the very same fitting polynomial is $\mathfrak{S}_p \approx 0.0857$, and the distance $\mathfrak{D}_p \approx 0.1629$. Basically, using g_2 instead of g_1 in order to normalize the raw data sets pays off.

Finally, we present our requirements for all five metrics. When searching for a fitting polynomial p for yet another data set, we wish that $\mathfrak{M}_p$ and μ^k are as close to 1 as possible; $\mathfrak{S}_p$, $\mathfrak{D}_p$ and σ^k are as close to zero as possible.

What comes to the verbose-partial running time scheme g_3, we have compiled Table 4.1 presenting the optimal values of the aforementioned metrics and the ρ parameters that yield them.

Table 4.1: This table presents most optimal values of ρ parameters for five aforementioned metrics using verbose partial running time scheme $g_3(100, H, \rho)$.

Data set k	$\mathfrak{M}(p)$	$\mathfrak{M}(p)\ \rho$	$\mathfrak{S}(p)$	$\mathfrak{S}(p)\ \rho$	$\mathfrak{D}(p)$	$\mathfrak{D}(p)\ \rho$	$\bar{R}$	$\bar{R}\ \rho$	σ	$\sigma\ \rho$
$k = 20$	1.0014	2.7	0.0124	7.7	0.1690	10.0	0.9992	2.7	0.2018	9.4
$k = 40$	0.9989	3.2	0.0230	10.0	0.1674	10.0	0.9995	3.1	0.2126	10.0
$k = 60$	0.9989	2.8	0.0011	6.6	0.1707	10.0	1.0023	2.7	0.2070	8.2
$k = 80$	1.0013	2.5	0.0075	8.3	0.1533	10.0	0.9981	2.5	0.1895	10.0
$k = 100$	1.0014	2.8	0.0057	9.5	0.1862	10.0	0.9983	2.8	0.2163	10.0

In Table 4.1, we use notational short cuts $p \stackrel{\text{def}}{=} p^{k,g_3}$, $\mathfrak{M}(p) \stackrel{\text{def}}{=} \mathfrak{M}(p^{k,g_3})$, $\mathfrak{S}(p) \stackrel{\text{def}}{=} \mathfrak{S}(p^{k,g_3})$, $\mathfrak{D}(p) \stackrel{\text{def}}{=} \mathfrak{D}(p^{k,g_3})$, $\bar{R} \stackrel{\text{def}}{=} \bar{R}^{k,g_3}$ and $\sigma \stackrel{\text{def}}{=} \sigma^{k,g_3}$. If we aim to optimize $\bar{R}^{k,g_3}$ of a data set, we choose from Table 4.1 the data set $k = 40$ and, thus, $\rho = 3.1$. We plot it and its fitting curve below.

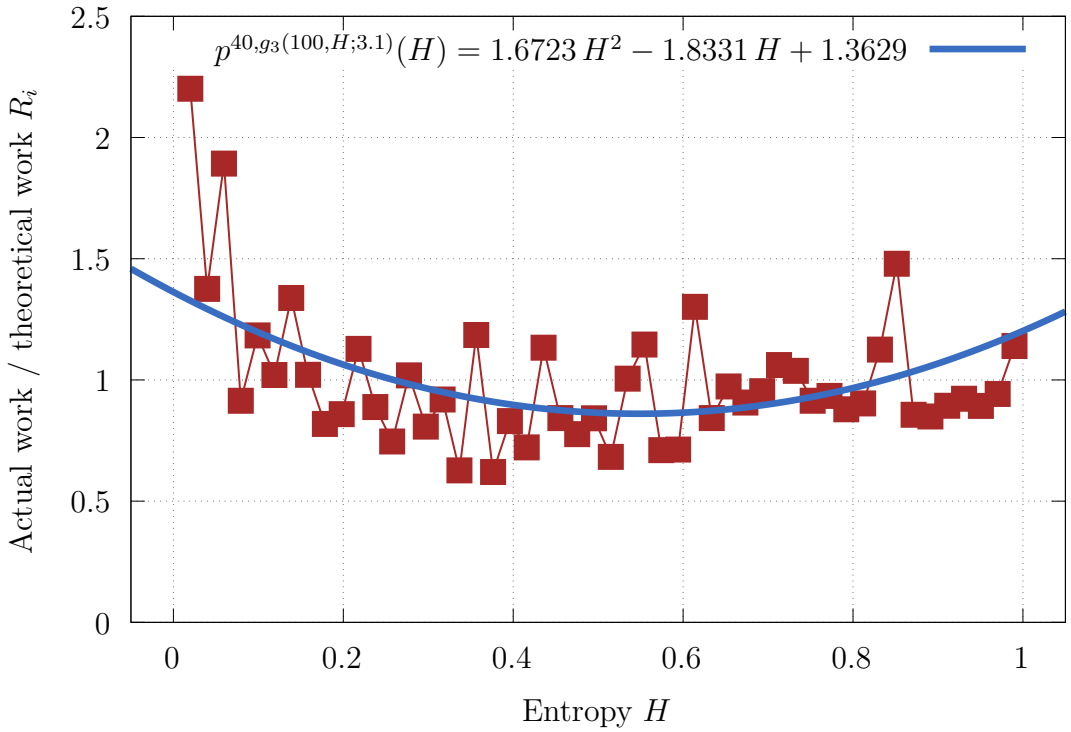


Figure 4.8: In this figure, we have the raw data set $\tilde{D}^{40}$ normalized by $g_3(100, H; 3.1)$ producing D^{40}.

Next, we will review the semi-verbose running scheme statistics.

Table 4.2: This table presents most optimal values of γ parameters for five aforementioned metrics using semi-verbose running time scheme $g_4(100, H, \gamma)$.

Data set k	$\mathfrak{M}(p)$	$\mathfrak{M}(p)\ \gamma$	$\mathfrak{S}(p)$	$\mathfrak{S}(p)\ \gamma$	$\mathfrak{D}(p)$	$\mathfrak{D}(p)\ \gamma$	$\bar{R}$	$\bar{R}\ \gamma$	σ	$\sigma\ \gamma$
D^{20}	0.9965	0.70	0.0438	0.66	0.1990	0.00	0.9953	0.70	0.2472	0.53
D^{40}	1.0042	0.69	0.0765	0.62	0.1832	0.00	1.0014	0.69	0.2410	0.52
D^{60}	0.9985	0.70	0.0345	0.67	0.1903	0.00	0.9972	0.70	0.2297	0.61
D^{80}	1.0061	0.72	0.0742	0.66	0.1706	0.00	1.0036	0.72	0.2146	0.60
D^{100}	1.0008	0.70	0.0645	0.64	0.2118	0.00	0.9988	0.70	0.2624	0.52

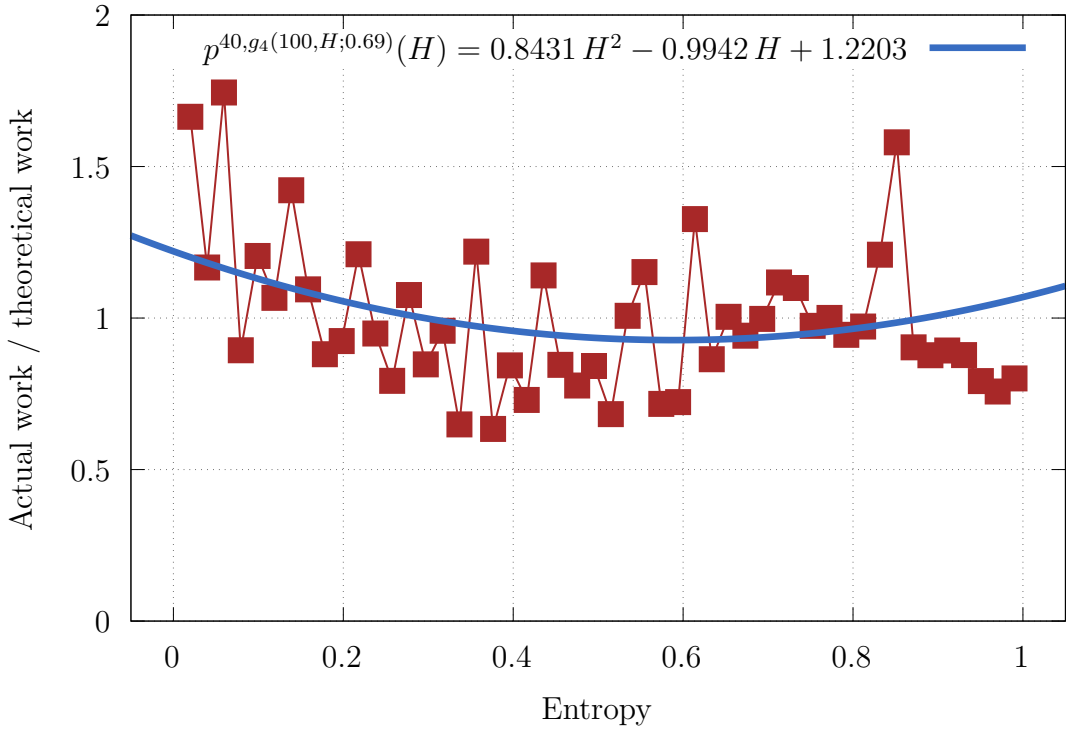


Figure 4.9: In this figure, we have the raw data set $\tilde{D}^{40}$ normalized by $g_4(100, H; 0.69)$ producing D^{40}.

Figure 4.9 shows us that using $g_4(100, H, 0.69)$ for raw data set normalization produces a slight improvement over g_1 but not that much as g_2. The data set mean $\bar{R}^{40,g_4}$ of D^{40} under aforementioned settings is ≈ 1.001, the data set standard deviation $\sigma^{40,g_4} \approx 0.2410$, the mean of $p^{40,g_4}(H)$ $\mathfrak{M}(p^{40,g_4}) \approx 1.004$, the standard deviation of $\mathfrak{S}(p^{40,g_4}) \approx 0.0765$ and the average distance of $p(H_i^{40})$ to R_i^{40,g_4} is $\mathfrak{D}(p^{40,g_4}) \approx 0.1832$.

What comes to the parameter γ, the value 0.69 maximizes the cohort of data sets for which the $\gamma = 0.69$ produced the closest fitting curve mean[1] and it produced the closest data set

[1] https://tinyurl.com/ym7jpcpt

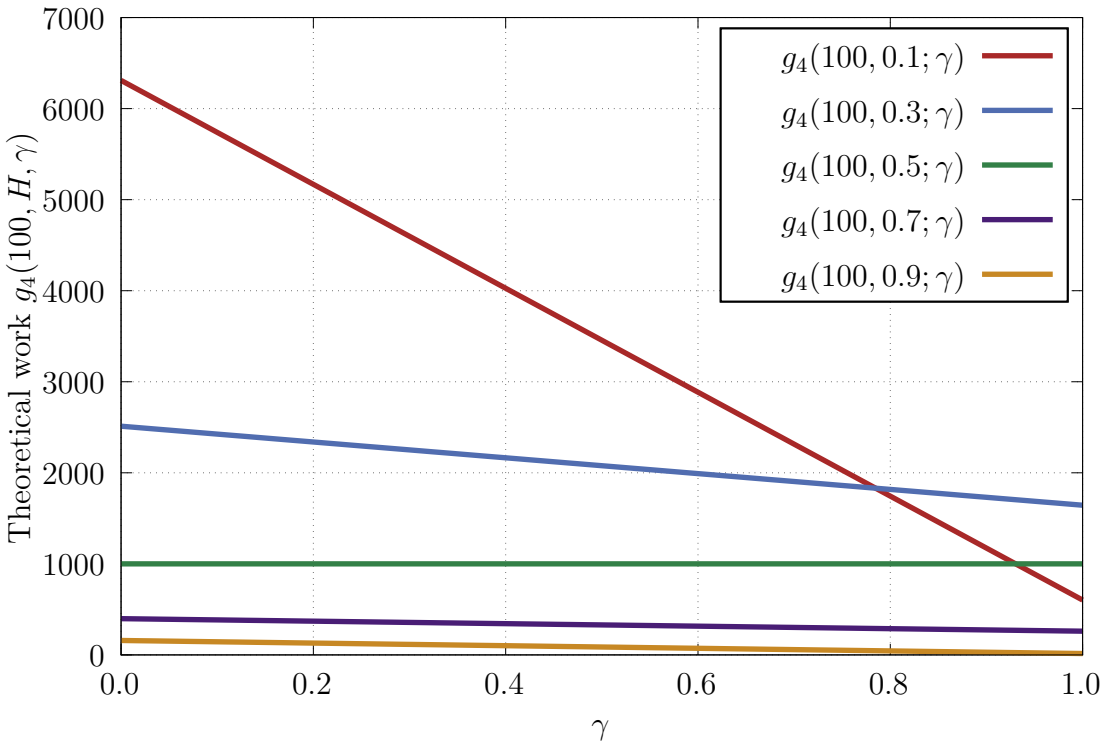


Figure 4.10: This plot shows that theoretical work effort of a single-element operation reduces substantially as the entropy H approaches 1. Also, it can be seen that – except for entropies H near zero, the work effort is not much affected by the value of $\gamma \in [0, 1)$.

mean[1].

Next, we conclude this chapter with a brief comparison of the running times of all the four list data types considered in this thesis. While all the single-element operations of an indexed list run in $\Omega(\sqrt{N}) \cap \mathcal{O}(N)$. However, one could choose a rough approximation that depends on the entropy:

$$\Theta(n^{2-H+\log_n(1-0.69\cos^2(\pi H))}).$$

The above running time expression is the best we can derive.

[1] https://tinyurl.com/dewe2bjw

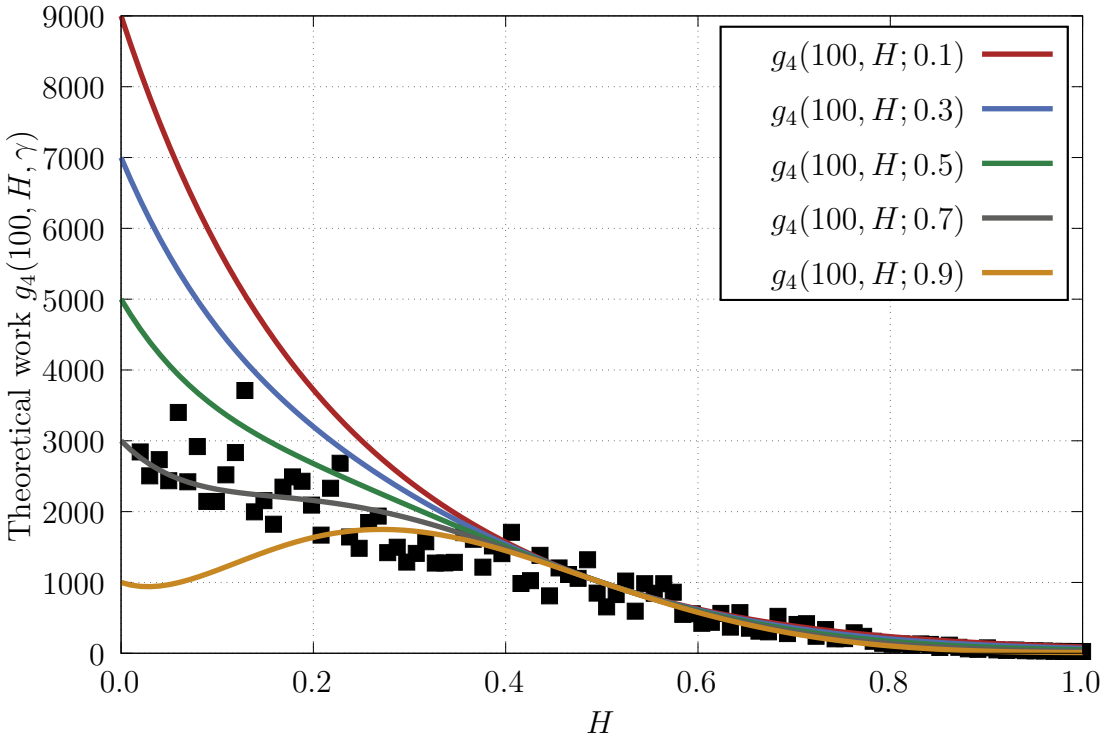


Figure 4.11: This plot shows that theoretical work effort is reducing more or less towards the $H = 1$. Also, it is evident that growing the value of γ improves the running time. Finally, note that, for example, $g_4(100, H; 0.6)$ closely resembles the plot in Figure 4.4, which implies that they more or less agree on their behaviour.

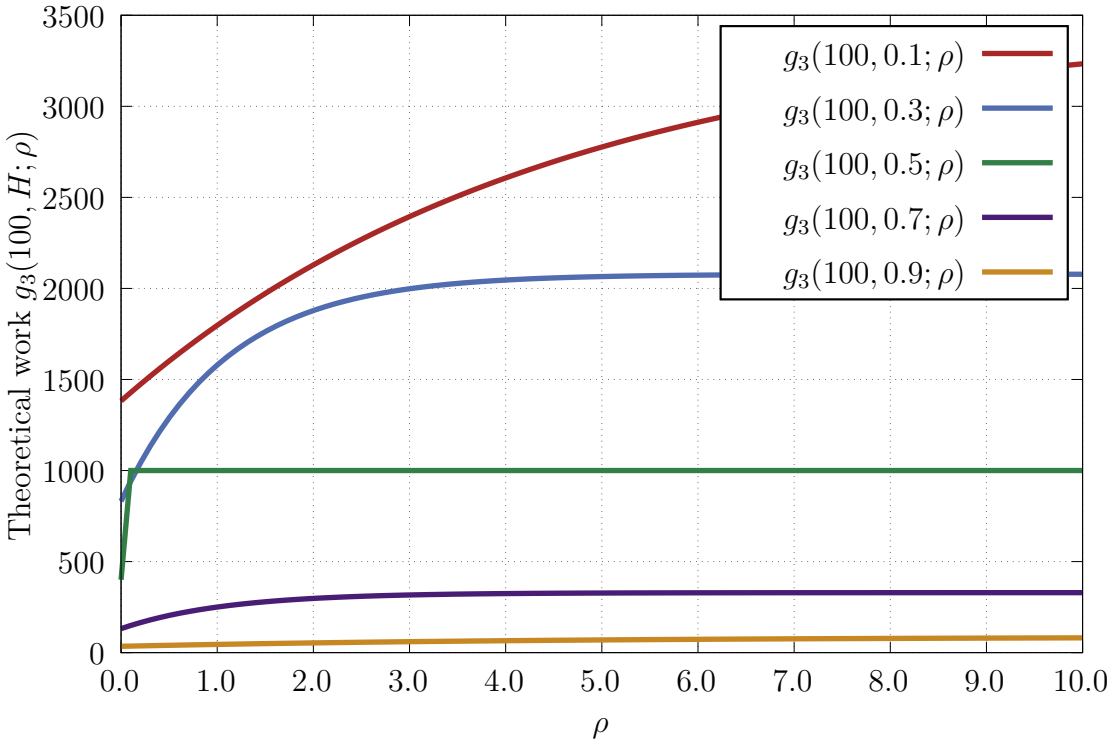


Figure 4.12: In this figure, one can observe that g_3 improves substantially as the entropy H grows. Also, note that running time is not really affected by the value of ρ.

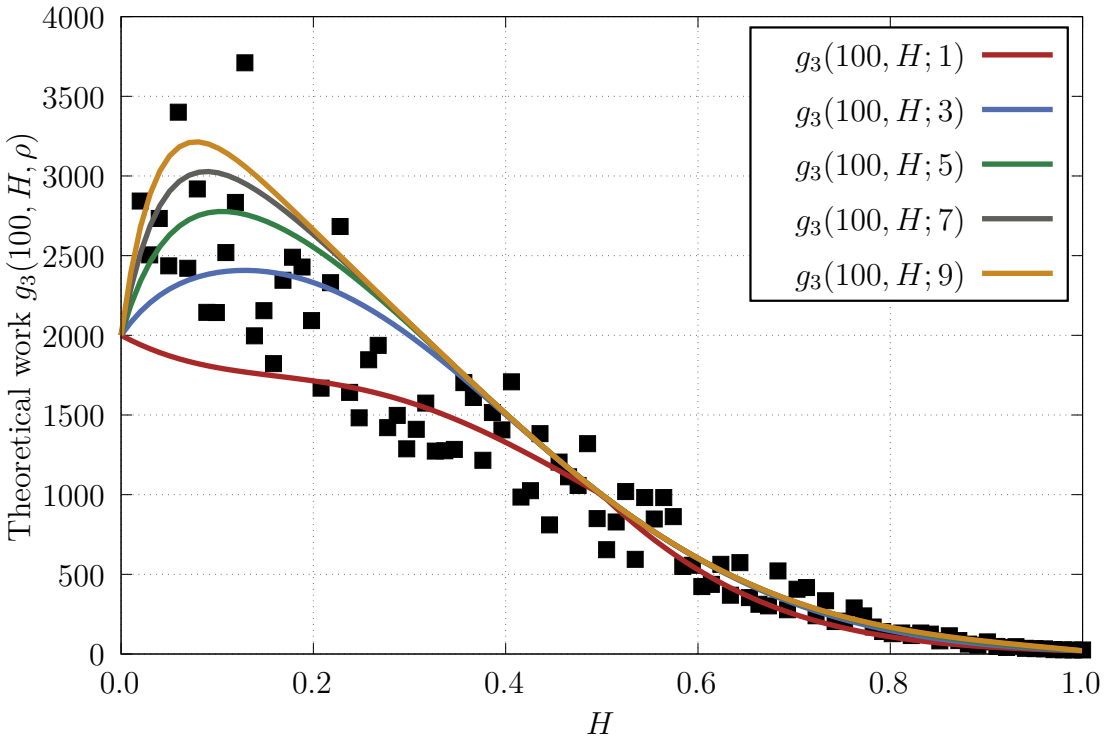


Figure 4.13: In this figure – just like in Figure 4.6, g_3 reduces as H approaches 1. What comes to the effect of ρ on the behaviour of g_3, on the range $H \in [0, 0.4]$, the running time improves as ρ grows noticeably.

Table 4.3: Running times for data structure/operation pairs.

Operation / List	Array	Linked	Tree	Indexed
PUSH-FRONT	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(\sqrt{N})$
PUSH-BACK	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(1)$
INSERT	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\sqrt{N})$
GET	$\Theta(1)$	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\sqrt{N})$
POP-HEAD	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(\sqrt{N})$
POP-TAIL	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(1)$
DELETE	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\sqrt{N})$
PUSH-FRONT-COLLECTION	$\Theta(N+M)$	$\Theta(M)$	$\Theta((M+N)\log(M+N) - N\log N)$	$\Theta(M+\sqrt{N+M})$
PUSH-BACK-COLLECTION	$\Theta(M)$	$\Theta(M)$	$\mathcal{O}(M+\log(M+N))$	$\Theta(M+\sqrt{N+M}-\sqrt{N})$
INSERT-COLLECTION	$\mathcal{O}(N)+\Theta(M)$	$\mathcal{O}(N)+\Theta(M)$	$\Theta(M\log N)$	$\Theta(M+\sqrt{N+M}-\sqrt{N})+\mathcal{O}(\sqrt{N})$
DELETE-RANGE	$\mathcal{O}(N)$	$\mathcal{O}(N)+\Theta(M)$	$\Theta(N\log N-(N-M)\log(N-M))$	$\mathcal{O}(\sqrt{N})+\Theta(M)$
ITERATOR-INSERT	$\mathcal{O}(N)$	$\Theta(1)$	$\Theta(\log N)$	$\mathcal{O}(\sqrt{N})$
ITERATOR-REMOVE	$\mathcal{O}(N)$	$\Theta(1)$	$\Theta(\log N)$	$\mathcal{O}(\sqrt{N})$

On finger configuration distribution

Before we take a look at the entropy bucket distributions, we need to define the concept of *entropy bucket* of width w. We take a range $R = [-1, 1]$, and split it into $2/w$ different equidistant buckets. Especially, for convenience, we require that w divides 2 evenly. Then, we split the R into $\langle [-1, -1+w), [-1+w, -1+2w), \ldots [1-w, 1), [1, 1]\rangle$. Note that we will have $2/w + 1$ distinct buckets and the rightmost bucket $[1, 1]$ is there to catch the entropy of $H = 1$. Next, we need to derive the actual bucket array B. We make it $2/w + 1$ array components long, and we keep iterating over all possible finger configurations. For each such configuration $C = \langle f_0, \ldots, f_n\rangle$, we compute the entropy of C and we convert its entropy H to the bucket index as follows:

$$i_{H;w} = \left\lfloor \frac{H}{w} \right\rfloor.$$

After computing the bucket index i, we increment $B[i]$. When the entire bucket array is computed, we derive a simple metric characterizing its behaviour. In order to do this, we need to define a *configuration split metric* $S_{B,w}$:

$$S_{B,w} = \underset{s\in\{-1,\ldots,2/w\}}{\arg\min} \left| \sum_{i=0}^{s} B[i] - \sum_{i=s+1}^{2/w} B[i] \right|.$$

Informally, the value of $S_{B,w}$ is an index into the bucket array B such that the total number of configuration in $B[0], B[1], \ldots, B[S_{B,w}]$ is as close to the total number of configurations in $B[S_{B,w}+1], B[S_{B,w}+2], \ldots, B[2/w]$ as possible.

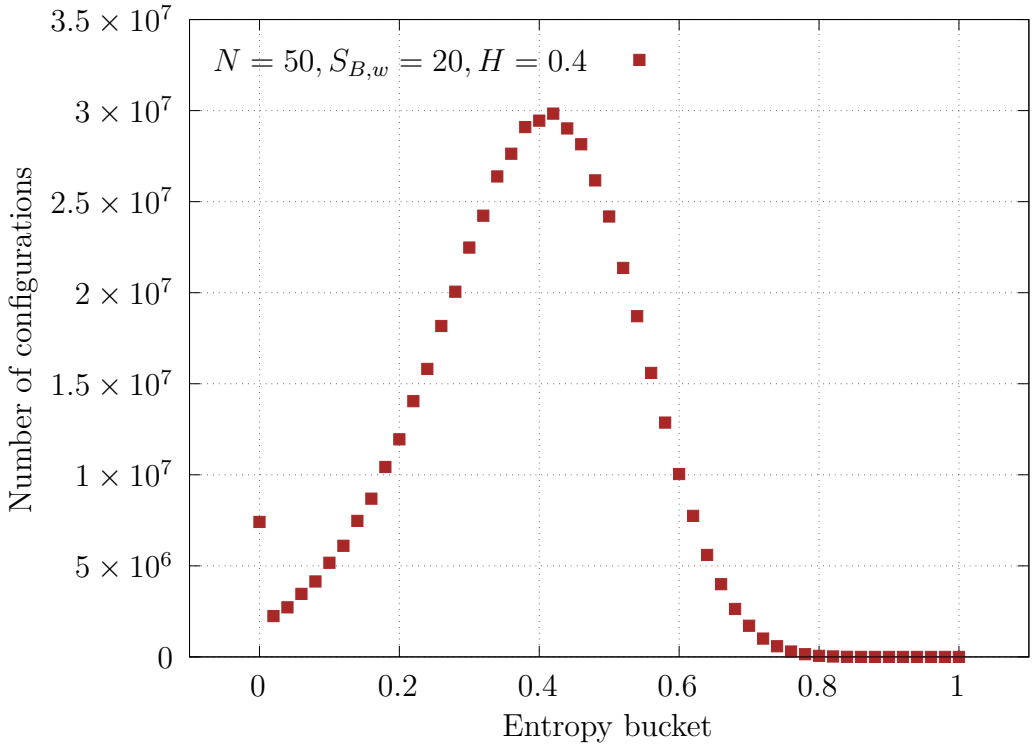


Figure 4.14: The entropy bucket distribution for the indexed list of size 50.

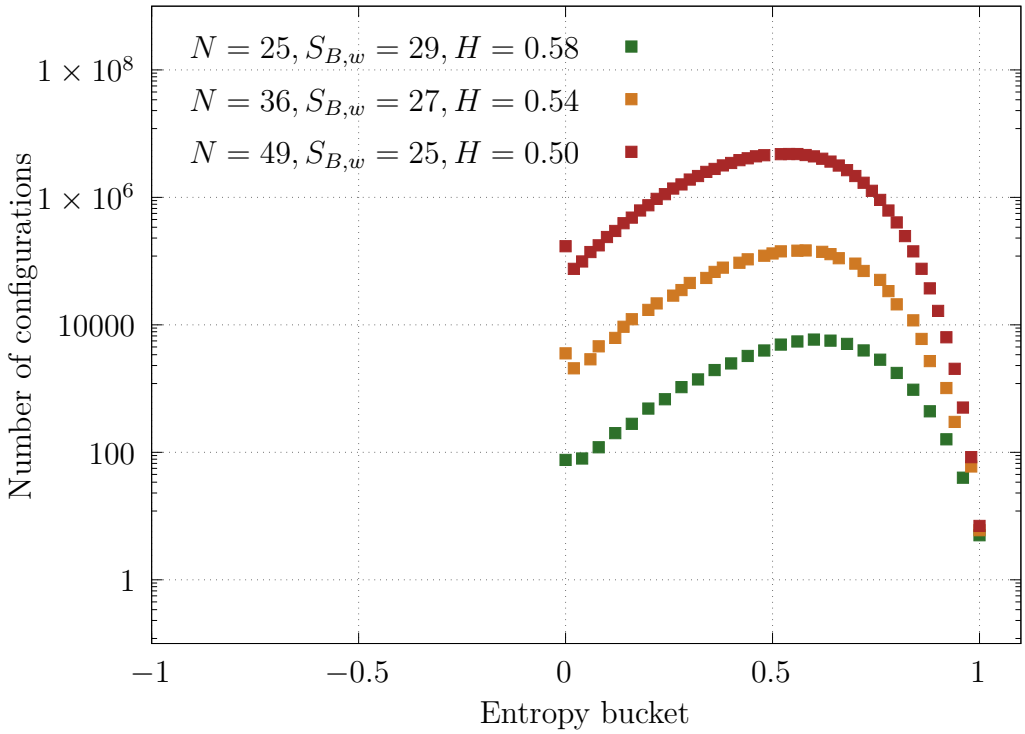


Figure 4.15: The entropy bucket distribution for the indexed list of sizes 25, 36 and 49.

In Figure 4.14 and Figure 4.15, we set for the presented buckets $w = 0.02$ so that the length

of B is 51. In Figure 4.14, we have inferred the bucket data from an indexed list of size 50 and, thus, 8 fingers. It immediately becomes apparent that the finger configurations resemble closely a normal distribution. The configuration split metric for that list is $S_{B,0.02} = 20$. Note also that in both Figures 4.14 and 4.15 the bucket $[0, w)$ has more configurations than its immediate neighbour $[w, 2w)$. This stems from the fact that we work with effective entropies converting all negative raw entropies to zero.

5 Experiments

In this chapter, we will present the results from extensive benchmarking we performed on all the list data structures under discussion: the dynamic table, the linked list, the tree list and the indexed list.

5.1 Benchmark specification

5.1.1 Set up of experiments

The benchmarking results we will present later in this chapter were run in the following environment.

Hardware, operating system and Java software

What comes to the hardware, we relied on the following set up:

- AMD Ryzen 7 5825U processor with eight (8) physical cores, 16 logical processors, base speed 2.00 GHz, 512 kB L1 cache, 4 MB L2 cache and 16 MB L3 cache.
- 16384 MB RAM memory, LPDDR5, running at 4266 MHz.

We ran the benchmark program with Java Development Kit 20 under Windows 11 Home 64-bit (10.0, Build 22631) and we have set the flags to JVM (`java.exe`) as `-Xms1000m -Xmx1000m` in order to allocate at the very beginning and at most ≈ 1 GB of RAM memory (that was sufficient for running the benchmarks). Also, in the benchmark, we set the running JVM process main thread priority to its maximum. Additionally, we set the process priority of the running JVM to real time. Finally, we have locked the CPU affinity of the benchmarking thread to a single core in order to not spend time on core migrations using the Java-Thread-Affinity project[2].

5.1.2 Benchmarked operations

The following list specifies how each list operations were benchmarked.

[2] https://github.com/OpenHFT/Java-Thread-Affinity

1. Push-Front pushes 2000 elements to the head of the list.
2. Push-Back pushes 10 000 elements to the tail of the list.
3. Insert inserts 2000 elements at random locations in the list.
4. Push-Front-Collection pushes 50 times a list of size 10 000 elements to the head of the list.
5. Push-Back-Collection pushes 50 times a list of size 10 000 to the tail of the list.
6. Insert-Collection adds 50 times a list of size 3500 at randomly chosen positions
7. Search access 500 times at randomly chosen positions.
8. Pop-Front pops 5000 elements from the head of the list.
9. Pop-Back pops 20 000 elements from the tail of the list.
10. Delete deletes 10 000 randomly chosen elements in the list.
11. Delete-Range keeps removing contiguous chunks of 500 elements from the list until the list load factor drops below 60%.

We denote the set of all lists under discussion as $\mathcal{L} = \{\texttt{dynamic}, \texttt{indexed}, \texttt{linked}, \texttt{tree}\}$. The set of all benchmark operations is $\mathcal{O} = \{\text{Push-Front}, \text{Push-Back}, \ldots, \text{Delete-Range}\}$, Also, the initial sizes are drawn from the set $\mathcal{S} = \{10^5, 2 \times 10^5, \ldots, 10^6\}$.

At this point, we need to draw the difference between the concepts of *benchmark operation* and *benchmark method*. The former denotes one of the operation $o \in \mathcal{O}$, while the latter denotes a single benchmark unit that is represented via a triplet $(l, o, s) \in \mathcal{L} \times \mathcal{O} \times \mathcal{S}$. The interpretation of such a triplet is that we start with the list of type l with the initial size of s, and we run the *benchmark operation* o on it. We denote the duration of a benchmark method (l, o, s) as $||(l, o, s)|| \in \mathbb{R}_{\geq}$.

We are mostly interested in the following benchmark result. For all $l \in \mathcal{L}$, we have computed

$$T_l = \sum_{o \in \mathcal{O}} \sum_{s \in \mathcal{S}} ||(l, o, s)||,$$

which yields the total duration time for all benchmark methods operating on l. Also, we have measured the list/per-operation durations for all $(l, o) \in \mathcal{L} \times \mathcal{O}$

$$T_{l,o} = \sum_{s \in \mathcal{S}} ||(l, o, s)||.$$

What comes to the benchmark operations INSERT, INSERT-COLLECTION, SEARCH, DELETE and DELETE-RANGE, they rely on the pseudo-random number generators (PRNG for short). For each of the benchmark operations we have used exactly the same PRNG seed in order to make sure that on all lists the number sequence will be identical. To this end, we have used Java's `java.util.Random` class[1].

5.2 Reflection

Figure 5.1(a) presents the durations of the dynamic table. We see here that for the most part it performs fairly fast, yet operations PUSH-FRONT, INSERT, POP-FRONT and DELETE start to take a hit.

Figure 5.1(b) presents the durations of the linked list. While most operations seem to run fast, the linked list is inferior on INSERT which took, in this benchmark, the duration of 8985 milliseconds. Also, it is inferior on DELETE which took even 44328 milliseconds. Also, note that the total duration of all the benchmark operations without counting INSERT and DELETE is 5395 milliseconds unlike in indexed list that very same measure is 387 milliseconds.

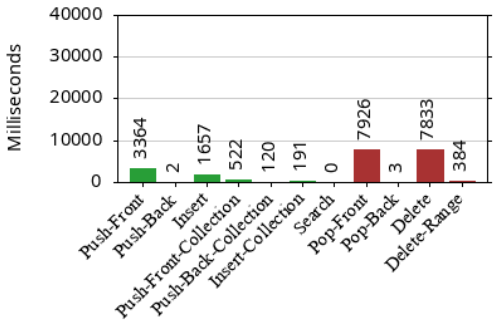
Figure 5.1(c) presents the durations of the tree list. It performs really fast with only PUSH-FRONT-COLLECTION being somewhat slower. We, however, need to note that in the tree list from Apache Commons Collections[2], operation PUSH-BACK-COLLECTION is optimized, while PUSH-FRONT-COLLECTION simply calls PUSH-FRONT for each element in the collection being added. In theory, one could have implemented PUSH-FRONT-COLLECTION using an symmetric algorithm, yet we decided to omit that opportunity.

Figure 5.1(d) presents the durations of the index list. One can observe that the indexed list outperforms all the other three list types despite the fact it runs not that fast as the tree list should have.

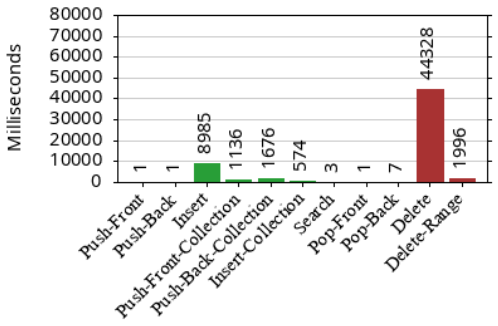
The total benchmark results ($T_l, l \in \mathcal{L}$) are plotted in Figure 5.2. It is evident that the indexed list outperforms all the other three list types. We could informally say that the indexed list "averages smoothly over operations".

[1] https://github.com/openjdk/jdk20/blob/master/src/java.base/share/classes/java/util/Random.java

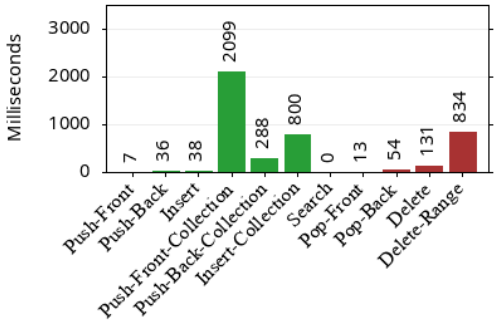
[2] TreeList.java



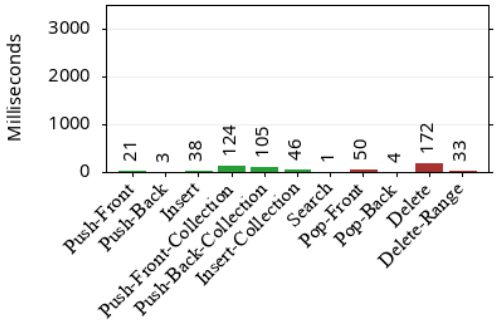
(a) Dynamic table



(b) Linked list



(c) Tree list



(d) Indexed list

Figure 5.1: Durations of all operations over all list implementations. Note that for Figure 5.1(a) the time axis proceeds up to 40000 milliseconds (40 seconds). For Figure 5.1(b) the very same time axis proceeds up until 80000 milliseconds (80 seconds). Also, for Figures 5.1(c) and 5.1(d) the very time axes proceed up until 3000 milliseconds (3 seconds).

Finally, we present the sum of all operations over all the four list implementations:

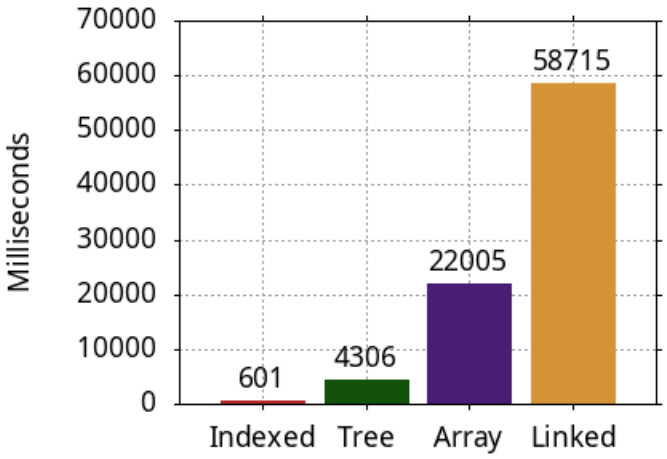


Figure 5.2: The height of each list type bar denotes the number of milliseconds each of the list has spent in the benchmark battery.

As we can see from Figure 5.2, on versatile use cases with large data the indexed list is superior even compared to the tree list. The indexed list has run in the benchmarking battery roughly seven times faster than the tree list, roughly 36 times faster than the dynamic table, and roughly 97 times faster than the linked list.

5.3 Additional experimentation

Since $L.C$ is a doubly-linked list, we have two (simple) options how to arrange it: (1) keep $L.C.head$ and $L.C.tail$ as above, or (2) instead of the two aforementoned references, keep a single ***sentinel node*** $L.C.sentinel$, such that, on empty list, $L.C.sentinel.next = L.C.sentinel.prev = L.C.sentinel$. Effectively, the option (2) would imply the circular list structure. Assuming the second option, prepending a new node would link it after $L.C.sentinel$ and before $L.C.sentinel.next$; also, appending a new node would link it before $L.C.sentinel$ and after $L.C.sentinel.prev$. Similarly, unlinking the head/tail node would not require handling any edge cases and would not require any `if`-statements.

Our simple experiment (see Figure 5.3), however, showed that using circular linked list structure requires larger effort from what we used to in this text.

Algorithm 1: ADD-FIRST(L, x)

```
1 n ← new node
2 n.data ← x
3 n.prev ← L.sentinel
4 n.next ← L.sentinel.next
5 L.sentinel.next.prev ← n
6 L.sentinel.next ← n
```

Algorithm 2: ADD-LAST(L, x)

```
1 n ← new node
2 n.data ← x
3 n.prev ← L.sentinel.prev
4 n.next ← L.sentinel
5 L.sentinel.prev.next ← n
6 L.sentinel.prev ← n
```

Algorithm 3: REMOVE-FIRST(L)

```
1 n ← L.sentinel.next
2 n.next.prev ← L.sentinel
3 L.sentinel.next ← n.next
4 n.prev ← nil
5 n.next ← nil
```

Algorithm 4: REMOVE-LAST(L)

```
1 n ← L.sentinel.prev
2 n.prev.next ← L.sentinel
3 L.sentinel.prev ← n.prev
4 n.prev ← nil
5 n.next ← nil
```

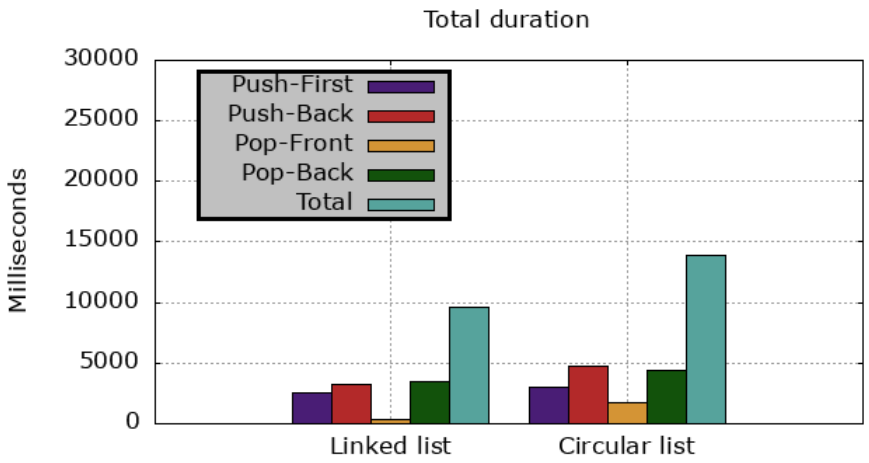


Figure 5.3: The running time of a linked and circular lists.

Table 5.1: This table elaborates on Figure 5.3. All integer entries are durations in milliseconds.

List	Push-Front	Push-Back	Pop-Front	Pop-Back	Total
Linked	2554	3240	314	3524	9632
Circular	3052	4722	1744	4409	13927

From Table 5.1, we can observe that the entire benchmark for the conventional linked list takes 27% less time than the circular one.

5.4 Benchmarking skip list structures

In this section, we will benchmark three skip list implementations and one red-black tree implementation of an ordered set. Also, we will concentrate on four use cases that produce the same content in skip lists, red-black trees and indexed lists. The additional structures are listed below.

1. Skip list as implemented in Pugh's paper [Efr24b].
2. `java.util.concurrent.ConcurrentSkipListMap` from JDK 20 [Git19].
3. `SkipListMap` which is a rewrite of `ConcurrentSkipListMap` whithout any concurrency constructs [Efr24c].
4. `java.util.TreeMap` which is the red-black tree implementation of an ordered set [Git22].

5.4.1 William Pugh's implementation

First, we need to exclude the Pugh's implementation of the skip list since it is slower than `ConcurrentSkipListMap` and `TreeMap` by almost two orders of magnitude. In a benchmark [Efr24a], we get the following results:

Table 5.2: Benchmarking Pugh's version against `ConcurrentSkipListMap`. The integer values in the cells are durations in milliseconds.

List / Operation	INSERT	CONTAINS	DELETE	TOTAL
Pugh's skip list	516	1423	696	2635
`TreeMap`	18	7	11	36
`ConcurrentSkipListMap`	31	12	16	59

As can be seen, in total, the Pugh's skip list implementation is approximately $\lg(2697/59) \approx \lg(45.7) \approx 1.66$ orders of magnitude slower that `ConcurrentSkipListMap`. The benchmark procedure is rather simple. We create a list $L = \langle 0, 1, \ldots 9999 \rangle$ and shuffle it randomly (the random seed is a fixed constant). Then, we add all the elements in L to both Pugh's skip list and `ConcurrentSkipListMap`. After that, we access all the elements in L in (the shuffled) order for both the data structures. Finally, we remove the elements in L from both the lists.

Next, we will investigate performance of three ordered set data structures:

- `TreeMap`,
- `SkipListMap`, the `ConcurrentSkipListMap` without concurrency facilities,
- `ConcurrentSkipListMap`.

The benchmark resides in GitHub[1]. Basically, we iterate the same benchmark battery ten times. For each iteration, we create a random integer array L of length one million array components. Each integer in the array is drawn uniformly and randomly from the set $\{0, 1, \ldots, 10^6 - 1\}$. Then, we add each integer in L to all the ordered sets. Secondly, we access each element in the ordered sets. Finally, in order to benchmark the DELETE operation, we, first, remove one million times integers that are drawn from uniform, random distribution of the set $\{0, 1, \ldots, 2^{31} - 1\}$. After that, we just remove all the elements in L.

[1] GitHub

Table 5.3: Benchmarking ordered set data structures.

List / Operation	INSERT	CONTAINS	DELETE	TOTAL
`TreeMap`	31	201	347	579
`SkipListMap`	41	131	302	474
`ConcurrentSkipListMap`	24	150	339	513

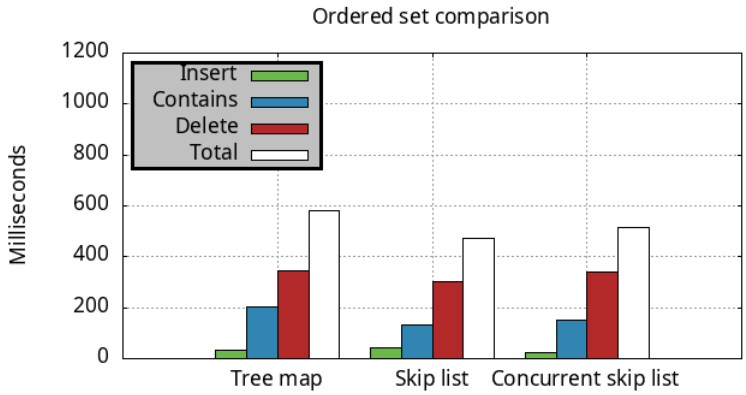


Figure 5.4: The plot plotting the duration times of each map data structure. For each structure, we show only the most fundamental operations: INSERT, CONTAINS and DELETE.

As we can see from Table 5.3, `SkipListMap` (the implementation of a skip list without concurrency constructs) is faster than `ConcurrentSkipListMap` and `TreeMap`. What comes to the difference between the two skip list data structures is the fact that operation in `SkipListMap` like

```
boolean flag;

if (b.next == n) {
    b.next = p;
    flag = true;
} else {
    flag = false;
}
```

is faster than

```
boolean flag = NEXT.compareAndSet(b, n, p);
```

even if `compareAndSet`[1] is defined as

[1] VarHandle.compareAndSet

```
public final native
@MethodHandle.PolymorphicSignature
@IntrinsicCandidate
boolean compareAndSet(Object... args);
```

which implies that the method `compareAndSet` is native and, thus, is implemented in a lower level language for a particular platform. Perhaps it even boils down to the `LOCK CMPXCHG` CPU instruction, yet we failed to verify that assumption regardless the effort.

Next, we will concentrate on four use cases of ordered sets and lists that produce the data structures with the same logical content:

Add asc. Add elements to an ordered set in ***ascending*** order via INSERT. Also, add elements in the exactly the same order to the indexed list via PUSH-BACK.

Add desc. Add elements to an ordered set in ***descending*** order via INSERT. Also, add elements in exactly the same order to the indexed list via PUSH-FRONT.

Remove asc. Remove the elements from the ordered sets in ***ascending*** order via DELETE. Also, remove the elements from the indexed list in ***ascending*** order via POP-FRONT operation.

Remove desc Remove the elements from the ordered sets in ***descending*** order via DELETE. Also, remove the elements from the indexed list in ***descending*** order via POP-BACK operation.

What comes to the actual elements to operate on in the benchmark, the ascending sequence is $\langle 0, 1, \ldots, 10^6 - 1\rangle$ and the descending sequence is $\langle 10^6 - 1, 10^6 - 2, \ldots, 1, 0\rangle$.

The respective benchmark resides in GitHub[1].

In the above benchmark, the benchmark method ADD ASC. simply adds the integers $0, 1, \ldots, 10^6 - 1$ to all the three data structures. Each element added to an ordered set is added via INSERT and the same element is added to the indexed list via PUSH-BACK. REMOVE DESC. removes the elements from the data structures in the reverse order $10^6 - 1, 10^6 - 2, \ldots, 1, 0$. Removal happens for the ordered sets via DELETE and via POP-BACK for indexed list. ADD DESC. adds the elements to the empty data structures in the order $10^6 - 1, 10^6 - 2, \ldots, 1, 0$. Once again, adding here for the ordered sets happen via INSERT and for the indexed list via PUSH-FRONT. Finally, REMOVE ASC. removes all the elements from all the data structures

[1] GitHub

in order $0, 1, \ldots 10^6 - 1$. For the ordered sets, it happens via DELETE operation, and for the indexed list, it happens via POP-FRONT.

Table 5.4: Benchmarking ascending/descending sequences.

List / Operation	ADD ASC.	REM. DESC.	ADD DESC.	REM. ASC.	GET	Σ
`TreeMap`	191	105	220	108	114	738
`ConcurrentSkipListMap`	211	469	9	229	168	1086
`IndexedLinkedList`	70	14	1228	1661	700	3673

The following two plots summarize the durations of each data structure on each use case. In Figure 5.5 we compare all the four benchmark methods.

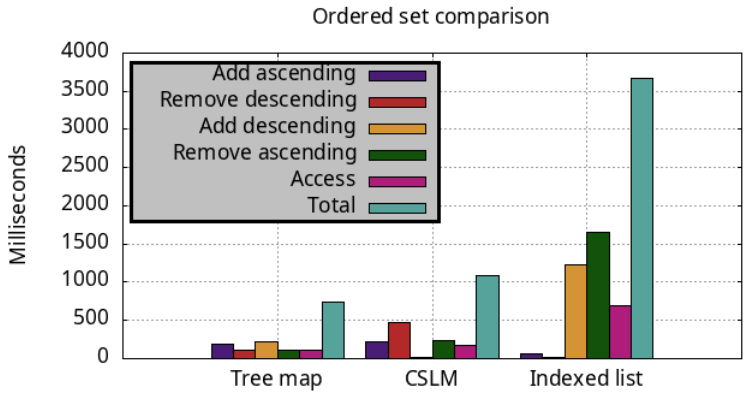


Figure 5.5: Durations of all the three data structures. For each structure we show durations of respective operations.

6 Conclusions

In this thesis, we have discussed a possibly novel list data structure implementation called *indexed list*. It exhibits reasonable speed-up due to an internal data structure called *finger list*. The actual container is arranged as a non-circular doubly-linked list. A single object in the finger list is called a *finger*. Each finger consists of only two fields: a pointer to a linked list node, and the appearance index of the node being pointed to.

The work of performing a single-element operation in the indexed list is the sum of consulting the finger list and rewinding a node pointed by the relevant finger to the target node. If we fix the size of an indexed list, then the work of the single-element operation minimizes when the number of fingers in the finger list is $\Theta(\sqrt{N})$.

The indexed list's performance is sensitive to the finger list distribution. If the fingers are distributed over the indexed list evenly, the running time of an single-element operation will be no worse than $\Theta(\sqrt{N})$.

In order to formalize the concept of finger list distribution, we derive the metrics called *entropy*, which takes values from the range $[0, 1]$. The more evenly the fingers are distributed, the closer the entropy is to the value of one. In other words, we wish to keep the entropy as high as possible. To this end, it is done by some indexed list operations that rearrange partially the finger list distribution such that the entropy does not decrease.

In Chapter 1, we briefly explained the overall outline of this thesis. Later in the Chapter 2, we presented the notational conventions used throughout in this work. Then, we defined formally the asymptotic notations. After that, we briefly defined the index list and its entropy. This is followed by useful definitions such as mean or standard deviation of a continuous function on a given range. Finally, we formally defined the list operations.

In the Chapter 3, we have discussed the four other list data structure implementations for the sake of comparison: a dynamic table, an ordinary linked-list, a tree list, and a skip list.

In the Chapter 4, we began at discussing the internal structure of the indexed list. Then, we (again) defined its entropy. Later, we have proved both formally and empirically that the running time of a single-element operation on the indexed list decreases as the entropy of the target indexed list grows towards entropy 1. After that, we discussed so called *indexed list normalization*, which is a rather simple heuristic that (most often) allows us to optimize the finger list distribution. Later in the very same chapter, we made an attempt to define the running time complexities of single-element operations that involve both the number of fingers

in the finger list and the current entropy. Finally, we conclude the chapter in question by deriving a simple metric called *entropy bucket distribution*. It communicates in what entropy sub-ranges each finger configuration belongs to. It became obvious, empirically, that the entropy of around 0.6 is most "populated" by finger list configurations.

In the Chapter 5, we run all four list data structures (all except the skip list) through identical benchmark batteries. It turned out that the indexed list survived the battery the fastest. The tree list was roughly seven times slower than the indexed list, the dynamic table was roughly 36 times slower than the indexed list, and, finally, the linked list was roughly 97 times slower than the indexed list. Despite that, we need to note that the indexed list is slower than the tree list in theoretical sense ($\Theta(\sqrt{N})$ versus $\Theta(\log N)$), and so, starting from some substantial N the tree list will outperform the index list. However, for semi-large data, the indexed list works well enough. In addition to those contributions, we have proved empirically that circular doubly-linked list is somewhat slower than an ordinary doubly-linked list.

Also, in the Chapter 5, we benchmarked the indexed list against skip lists and tree maps as in Java's `java.util.TreeMap`. Needless to say, the indexed list turned out to be inferior on most benchmarks. However, we note that indexed lists and skip lists/tree maps solve different problems: the indexed list is a dynamic sequence where duplicates are allowed and the data is not necessarily sorted by any order. On the other hand, tree maps and skip lists both implement an *ordered set*, which preserves the elements in ascending order and does not allow duplicates.

In the Appendix A, we present in rather verbose fashion the pseudo-code specifying all the basic 11 operations, that are pushing to both ends, popping from both the ends, inserting somewhere in between, removing somewhere in between, accessing an element, pushing a collection to both ends, inserting a collection somewhere in between, and, finally, deleting a sub-range of the indexed list. The chapter in question is concluded by a table summarizing all running time complexities for every list data structure/operation pair.

Finally, in the Appendix B, we prove formally that **Push-Back** operation on dynamic tables may be implemented such that it runs in amortized constant time.

All in all, we make a bold claim that indexed lists are rather efficient on semi-large data such as, for instance, 100 000 elements and above. As a reminder, the tree list is still faster in theory; the tree list will start taking precedence in efficiency after the data load becomes sufficiently large.

Bibliography

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. "An algorithm for organization of information". In: *Soviet Mathematics Doklady, 3, 1259-1263* (1962).

[Efr24a] R. Efremov. *Pugh's skip list benchmark against* ***`ConcurrentSkipListMap`***. 2024. URL: https://tinyurl.com/2e427xyx (visited on 06/11/2024).

[Efr24b] R. Efremov. *Pugh's skip list implementation*. 2024. URL: https://tinyurl.com/hkc328h6 (visited on 06/11/2024).

[Efr24c] R. Efremov. *The sequential rewrite of* ***`ConcurrentSkipListMap`***. 2024. URL: https://tinyurl.com/2czmnzru (visited on 06/11/2024).

[Git19] GitHub. *The concurrent skip list implementation from JDK 20*. 2019. URL: https://tinyurl.com/y339cbnm (visited on 06/11/2024).

[Git22] GitHub. ***`TreeMap.java`*** *from JDK 20*. 2022. URL: https://tinyurl.com/2p8ppxza (visited on 06/11/2024).

[HS24] N. L. Hjort and E. A. Stoltenberg. *Probability Proofs for Stirling (and More): the Ubiquitous Role of* $\sqrt{\mathbf{2}\pi}$. 2024. arXiv: 2410.19555 [math.PR]. URL: https://arxiv.org/abs/2410.19555.

[Knu97a] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, p. 244. ISBN: 0201896834.

[Knu97b] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, p. 280. ISBN: 0201896834.

[Knu97c] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, p. 110. ISBN: 0201896834.

[Knu97d] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, p. 239. ISBN: 0201896834.

[Mic02] M. M. Michael. "High performance dynamic lock-free hash tables and list-based sets". In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '02. Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, 73–82. ISBN: 1581135297. DOI: 10.1145/564870.564881. URL: https://doi.org/10.1145/564870.564881.

[MPS92] J. I. Munro, T. Papadakis, and R. Sedgewick. "Deterministic skip lists". In: *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '92. Orlando, Florida, USA: Society for Industrial and Applied Mathematics, 1992, 367–375. ISBN: 089791466X.

[Pug90] W. Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Commun. ACM*, 33(6) (1990), 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

Appendix A Algorithms

Throughout this section, we will adopt the following notation:

- L - the indexed list object being operated on;
- i - the index of an element in the list;
- i_f - the index of a finger in the finger list array;
- f - a finger object;
- x - the element to remove;
- y - the element to add;
- M - the size of the collection being added, or the number of elements to remove in the bulk clear operation DELETE-RANGE;
- $\mathcal{Y} = \langle y_0, y_1, \ldots, y_{M-1} \rangle$ - the collection being added;
- b - in element ranges, denotes the index of the leftmost element in those ranges;
- e - in element ranges, denotes the index of the one past the rightmost element in those ranges (that is, unlike b, it's an exclusive index).

We opt to show the algorithms according to pre-order traversal.

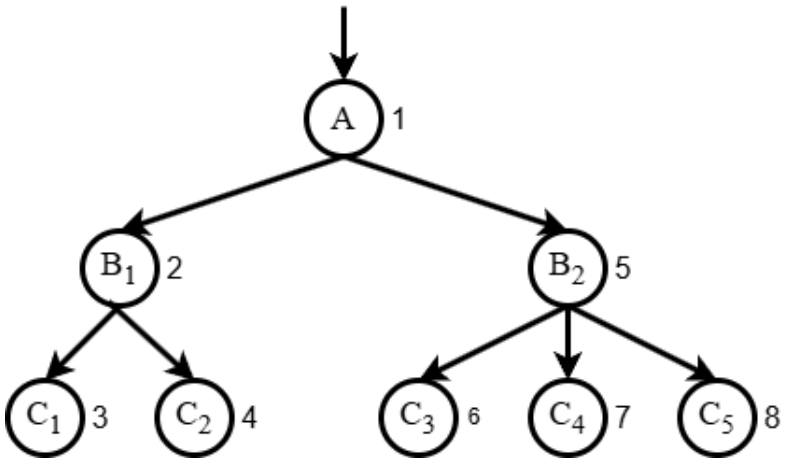


Figure A.1: Pre-order traversal over algorithm call tree. The given example algorithms are called $A, B_1, B_2, C_1, C_2, C_3, C_4, C_5$. If there is a directed arc $X \to Y$, where X and Y are algorithms, it implies that X calls Y. For this example call tree, we will present algorithms in the following order: $A, B_1, C_1, C_2, B_2, C_3, C_4, C_5$.

We make it explicit here, that our pseudo-code assumes a garbage-collected implementation. So, in non-garbage-collected languages such as C/C++ one must call **delete**/**delete**[] on data that must be released. Our Java implementation is available in GitHub[1].

A.1 Single element insertion operations

Insertion operations on individual elements are responsible for adding data to the indexed list. There is three of them provided: PUSH–FRONT(L, y) inserts the element y right before the first element of L. PUSH-BACK(L, y) inserts the element y right after the last element of L. Finally, INSERT(L, i, y) inserts the element y between $(i-1)$st and ith elements in L.

A.1.1 Insert at index proxy

The Algorithm 5 specifies a proxy method delegating to the proper insertion operation.

Algorithm 5: ADD-AT-INDEX(L, i, y)

```
1 if i < 0 or i > L.C.size then
2     error "Index out of range"
3 if i = L.C.size then
4     LINK-LAST(L, y)
5 else if i = 0 then
6     LINK-FIRST(L, y)
7 else
8     LINK-BEFORE(L, y, i, GET-NODE(L, i))
```

A.1.2 Push to back

The Algorithm 6 inserts an input element y right after the tail element in the indexed list.

[1] https://github.com/coderodde/IndexedLinkedList/blob/main/src/main/java/io/github/coderodde/util/IndexedLinkedList.java

Algorithm 6: LINK-LAST(L, y)

```
1  oldTail ← L.C.tail
2  u ← new node
3  u.datum ← y
4  u.prev ← oldTail
5  L.C.tail ← u
6  if oldTail = nil then
7      L.C.head ← u
8  else
9      oldTail.next ← u
10 Increment L.C.size.
11 if MUST-ADD-FINGER(L) then
12     APPEND-FINGER-FOR-NODE(L, u, L.C.size − 1)
13 else
14     Increment L.F.𝔈.index.
```

The Algorithm LINK-LAST(L, y) appends y after the tail node of the list L. It runs in amortized constant time. Also, the dynamic table executes the LINK-LAST(L, y) in amortized constant time. The linked list runs it in exact constant time, and the tree list runs it in exact $\Theta(\log N)$ time.

The Algorithm MUST-ADD-FINGER returns **true** if and only if one finger must be added to the finger index in order to restore the data structure invariant, which is $L.F.size = \lceil\sqrt{L.C.size}\rceil$. The Algorithm APPEND-FINGER-FOR-NODE actually implements adding a new finger right after the last non-sentinel finger in the finger index and just right on the left from $L.F.\mathfrak{E}$.

The actual PUSH-BACK(L, y) simply calls LINK-LAST(L, y). The operation MUST-ADD-FINGER relies on GET-RECOMMENDED-NUMBER-OF-FINGERS(L) which simply returns $\lceil\sqrt{L.C.size}\rceil$.

The procedure APPEND-FINGER-FOR-NODE(L, u, i) runs as follows:

1. Create a new finger f. Set $f.node \leftarrow u$ and $f.index \leftarrow i$.
2. Call APPEND-FINGER-IMPL(L, f).

Basically, most of the work in APPEND-FINGER-FOR-NODE is delegated to ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE.

Algorithm 7: APPEND-FINGER-IMPL(L, f)

```
1  ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L.F.size + 2,
                                         L.F.size,
                                         1,
                                         1)
2  L.F.fingers[L.F.size − 1] ← f
3  L.F.fingers[L.F.size].index ← L.C.size
```

Algorithm 8: ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L, `requestedCapacity`, `fingerRangeStartIndex`, `fingerRangeLength`, `elementRangeLength`)

```
1  if requestedCapacity > ||L.F.fingers|| then
2      nextCapacity ← 2 × ||L.F.fingers||
3      while nextCapacity < requestedCapacity do
4          nextCapacity ← 2 × nextCapacity
5      nextFingerArray ← new finger array of capacity nextFingerArray
6      ARRAY-COPY(L.F.fingers,
                  0,
                  nextFingerArray,
                  0,
                  fingerRangeStartIndex)
7      numberOfFingersToShift ← L.F.size − fingerRangeStartIndex + 1
8      ARRAY-COPY(L.F.fingers,
                  fingerRangeStartIndex,
                  nextFingerArray,
                  fingerRangeStartIndex + fingerRangeLength,
                  numberOfFingersToShift)
9      L.F.fingers ← nextFingerArray
10     L.F.size ← L.F.size + fingerRangeLength
11     SHIFT-FINGER-INDICES-TO-RIGHT(L,
                                     fingerRangeStartIndex + fingerRangeLength,
                                     elementRangeLength)
```

Algorithm 8: ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L, `requestedCapacity`, `fingerRangeStartIndex`, `fingerRangeLength`, `elementRangeLength`)

```
12 else
13     SHIFT-FINGER-INDICES-TO-RIGHT(L, fingerRangeStartIndex, elementRangeLength)
14     numberOfSuffixFingers ← L.F.size − fingerRangeStartIndex + 1
15     ARRAY-COPY(L.F.fingers, fingerRangeStartIndex, L.F.fingers,
                  fingerRangeStartIndex + fingerRangeLength, numberOfSuffixFinger)
16     L.F.size ← L.F.size + fingerRangeLength
```

The Algorithm 8 is responsible for two actions: it makes sure that the internal finger index array can accommodate a specified number of fingers, and, also, there is room for inserting a range of new fingers.

Algorithm 9: ARRAY-COPY(S, i_S, D, i_D, N)

```
1 if S = D and i_S ≤ i_D then
2     for i ← 0 to N − 1 do
3         D[i_D + N − 1 − i] ← S[i_S + N − 1 − i]
4 else
5     for i ← 0 to N − 1 do
6         D[i_D + i] ← S[i_S + i]
```

The Algorithm 9 is responsible for moving data sub-arrays here and there. If $S = D$ and $i_S \leq i_D$, we must do the copy starting from larger indices and proceeding to the lesser indices. Otherwise, there is a risk that some data will be overwritten. For example, consider what would happen if the Algorithm 9 consisted only of the lines 5 and 6. Under such an assumption on $S = D, i_S = 0, i_D = 1, N = 3$, the resulting array $S = D$ would become $\langle 1, 1, 1, 1\rangle$ and not $\langle 1, 1, 2, 3\rangle$ as would be expected.

if $S = D = \langle 1, 2, 3, 4\rangle$ and $N = 3$, $i_S = 0$, $i_D = 1$, ARRAY-COPY on those arguments will yield a result $\langle 1, 1, 1, 1\rangle$ unless we copy backwards, and not the desired $\langle 1, 1, 2, 3\rangle$.

Algorithm 10: SHIFT-FINGER-INDICES-TO-RIGHT(L, i_f, M)

```
1 for i ← i_f to L.F.size do
2     L.F.fingers[i] ← L.F.fingers[i] + M
```

The Algorithm 10 merely adds M to the index of each finger from the finger range $L.F.fingers[i_f, i_f + 1, \ldots, L.F.size]$.

A.1.3 Push to front

The Algorithm PUSH-FRONT(L, y) simply calls LINK-FIRST(L, y). Both are, thus, responsible for adding y right before the head node of the index list ($L.C.head$).

Algorithm 11: LINK-FIRST(L, y)

```
1  oldFirst ← L.C.head
2  u ← new node
3  u.datum ← y
4  u.next ← oldFirst
5  L.C.head ← u
6  if oldFirst = nil then
7      L.C.tail ← u
8  else
9      oldFirst.prev ← u
10 Increment L.C.size.
11 if MUST-ADD-FINGER(L) then
12     PREPEND-FINGER-FOR-NODE(L, u)
13 else
14     SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, 0)
```

The LINK-FIRST(L, y) runs in exact $\Theta(\sqrt{N})$ time. The same operation runs in dynamic table in exact linear time. The linked list runs it in exact constant time, and the tree list runs it in exact $\Theta(\log N)$ time.

Algorithm 12: PREPEND-FINGER-FOR-NODE(L, u)

```
1  f ← new finger
2  f.node ← u
3  f.index ← 0
4  if L.F.size + 1 = ||L.F.fingers|| then
5      newFingerArray ← new finger array with capacity 2 × ||L.F.fingers||
6      ARRAY-COPY(L.F.fingers,
                  0,
                  newFingerArray,
                  1,
                  L.F.size + 1)
7      L.F.fingers ← newFingerArray
8      SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, 1)
9      Increment L.F.fingers[L.F.size + 1].index
10 else
11     SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, 0)
12     ARRAY-COPY(L.F.fingers,
                  0,
                  L.F.fingers,
                  1,
                  L.F.size + 1)
13 L.F.fingers[0] ← f
14 Increment L.F.size
```

What comes to SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, i_f), it simply calls SHIFT-FINGER-INDICES-TO-RIGHT($L, i_f, 1$).

The Algorithm 11 (PUSH-FRONT) has two distinctive sub-operations: SHIFT-FINGER-INDICES-TO-RIGHT-ONCE and PREPEND-FINGER-FOR-NODE. As we stated, both run in $\Theta(\sqrt{N})$ time, yet, unlike SHIFT-FINGER-INDICES-TO-RIGHT-ONCE, PREPEND-FINGER-FOR-NODE is not called every time PUSH-FRONT is called. For that reason, it might be of interest to determine the amortized running time of PREPEND-FINGER-FOR-NODE.

Consider the below table where the row "Work" counts the number of fingers to shift one position towards the larger indices plus one for creating and inserting a new finger at the first

array component of the finger array when prepending a new finger:

Operation number i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Work $W(i)$	1	2	0	0	3	0	0	0	0	4	0	0	0	0	0	0	5	0	0	...
Accumulated work $A(i)$	1	3	3	3	6	6	6	6	6	10	10	10	10	10	10	10	15	15	15	...

Above, we have

$$A(i) = \sum_{k=1}^{i} W(k).$$

Now, we can deduce that

$$A(N) = \sum_{i=1}^{N} W(i) = \sum_{i=1}^{\left\lceil \sqrt{N} \right\rceil} i,$$

and so the total effort is given by

$$T(N) = N + A(N).$$

Now, the amortized running time of PREPEND-FINGER-FOR-NODE is:

$$\begin{aligned}
\frac{T(N)}{N} &= \frac{1}{N}\left(N + \sum_{i=1}^{\left\lceil \sqrt{N} \right\rceil} i\right) \\
&= 1 + \frac{1}{2N}\left\lceil \sqrt{N} \right\rceil \left(\left\lceil \sqrt{N} \right\rceil + 1\right) \\
&\leq 1 + \frac{1}{2N}\left(\sqrt{N} + 1\right)\left(\sqrt{N} + 2\right) \\
&= 1 + \frac{1}{2} + \frac{3}{2\sqrt{N}} + \frac{1}{N} \\
&\leq 4 \\
&= \Theta(1),
\end{aligned}$$

and so we see that the procedure PREPEND-FINGER-FOR-NODE runs in amortized constant time.

A.1.4 Insert at index

Algorithm 13: LINK-BEFORE($L, y, i,$ `successor`)

```
1  predecessor ← successor.prev
2  newNode ← new node
3  newNode.datum ← y
   # Link the nodes:
4  newNode.next ← successor
5  newNode.prev ← predecessor
6  successor.prev ← newNode
7  predecessor.next ← newNode
8  Increment L.C.size
9  if MUST-ADD-FINGER(L) then
10     f ← new finger
11     f.node ← newNode
12     f.index ← i
13     INSERT-FINGER-AND-SHIFT-ONCE-TO-RIGHT(L, f)
14 else
15     i_f ← GET-FINGER-INDEX-IMPL(L, i)
16     SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, i_f)
```

The Algorithm 13 (LINK-BEFORE) is responsible for inserting the input element y between the $(i-1)$st and the ith elements. The actual INSERT-AT-INDEX(L, i, y) simply delegates to the call LINK-BEFORE$(L, y, i,$ GET-NODE$(L, i))$. The running time of this algorithm is $\Theta(\log N) + \mathcal{O}(\sqrt{N})$. Both the dynamic table and the linked list degrade on this operation to $\mathcal{O}(N)$. Tree list, on the other hand, runs this in exact $\Theta(\log N)$ time.

Algorithm 14: INSERT-FINGER-AND-SHIFT-ONCE-TO-RIGHT(L, f)

```
1  beforeFingerIndex ← GET-FINGER-INDEX-IMPL(L, f.index)
2  ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L,
                                         L.F.size + 2,
                                         beforeFingerIndex,
                                         1,
                                         1)
3  L.F.fingers[beforeFingerIndex] ← f
```

The Algorithm 14 insert the input finger in its proper location. If it does not fit in due to the fact that the finger index array is full, a larger array will be created.

Algorithm 15: GET-FINGER-INDEX-IMPL(L, i)

```
1  count ← |L.F| + 1 # Account for L.F.𝔈 too.
2  idx ← 0
3  while count > 0 do
4      it ← idx
5      step ← ⌊count / 2⌋
6      it ← it + step
7      if L.F.fingers[it].index < i then
8          it ← it + 1
9          idx ← it
10         count ← count − step − 1
11     else
12         count ← step
13 return idx
```

The Algorithm 15 is the corner stone of the entire indexed list data type. For the input element index i, it returns a finger index of the finger i_f such that $i < L.F.fingers[i_f]$ and that finger is closest to the element i.

The actual INSERT-AT-INDEX(L, i, y) merely checks that the index i is in bounds and then delegates to LINK-BEFORE

The actual PUSH-FRONT(L, y) does nothing else but delegate to LINK-FIRST($L; y$).

The procedure SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, i) merely increments indices of all the fingers in $L.F.fingers\,[i \ldots |L.F|]$. which implies that SHIFT-FINGER-INDICES-TO-RIGHT-ONCE(L, runs in $\Theta(\sqrt{N})$ time.

To recap, the operation MUST-ADD-FINGER(L) returns **true** if and only if $\lceil\sqrt{N+1}\rceil > \lceil\sqrt{N}\rceil$. Note that also operation PREPEND-FINGER-FOR-NODE(L, u) runs in $\Theta(\sqrt{N})$ time, so the running time of PUSH-FRONT is also $\Theta(\sqrt{N})$.

The actual procedure PUSH-BACK(L, y) simply delegates to LINK-LAST(L, y). What comes to APPEND-FINGER-IMPL, it is defined as follows:

Next, we present the above ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE: Above, c is the requested capacity, i_f is the starting index of the finger to move to the right towards higher indices, l is the length of the finger portion to move to the right, and, finally, r is the number of elements affected. The procedure ARRAY-COPY closely mimics the Java's

`java.lang.System.arraycopy` method[1]. For completeness, we state it below.

Algorithm 16: GET-NODE(L, i)

```
1  if |L.F| < 3 then
2      return sequentially scanned ith element in L.C
3  i_f ← GET-FINGER-INDEX-IMPL(L, i)
4  if i_f = 0 then
5      return GET-PREFIX-NODE(L, i)
6  if i_f ∈ {L.F.size − 1, L.F.size} then
7      return GET-SUFFIX-NODE(L, i)
8  a ← L.F.fingers[i_f − 1]
9  b ← L.F.fingers[i_f]
10 c ← L.F.fingers[i_f + 1]
11 Δ ← c.index − a.index
12 step ← ⌊Δ / 2⌋
13 saveBIndex ← b.index
14 nextBIndex ← a.index + step
15 b.index ← nextBIndex
   # Rewind the finger b.
16 if saveBIndex < nextBIndex then
17     for i ← 1 to nextBIndex − saveBIndex do
18         b.node ← b.node.next
19 else
20     for i ← 1 to saveBIndex − nextBIndex do
21         b.node ← b.node.prev
```

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#arraycopy-java.lang.Object-int-java.lang.Object-int-int-

Algorithm 16: Continuation of GET-NODE(L, i)

```
22 if i < nextBIndex then
23     leftDistance ← i − a.index
24     rightDistance ← b.index − i
25     if leftDistance < rightDistance then
26         return SCROLL-TO-RIGHT(a.node, leftDistance)
27     else
28         return SCROLL-TO-LEFT(b.node, rightDistance)
29 else
30     leftDistance ← i − b.index
31     rightDistance ← c.index − i
32     if leftDistance < rightDistance then
33         return SCROLL-TO-RIGHT(b.node, leftDistance)
34     else
35         return SCROLL-TO-LEFT(c.node, rightDistance)
```

The Algorithm 16 serves two purposes. First, it accesses the ith node. Secondly, it may rearrange fingers the closest to ith element in order to improve entropy.

Algorithm 17: GET-PREFIX-NODE(L, i)

```
1  a ← L.F.fingers[0]
2  b ← L.F.fingers[1]
3  aNode ← a.node
4  nextAIndex ← ⌊b.index / 2⌋
5  saveAIndex ← a.index
6  a.index ← nextAIndex
7  if saveAIndex < nextAIndex then
8      for k ← saveAIndex to nextAIndex − 1 do
9          aNode ← aNode.next
10 else
11     for k ← nextAIndex to saveAIndex − 1 do
12         aNode ← aNode.prev
13 a.node ← aNode
14 if i < nextAIndex then
15     leftDistance ← i
16     rightDistance ← nextAIndex − i
17     if leftDistance < rightDistance then
18         return SCROLL-TO-RIGHT(L.C.head, i)
19     else
20         return SCROLL-TO-LEFT(aNode, rightDistance)
21 else
22     return aNode
```

The Algorithm 17 handles the special case of the Algorithm 16 where the request is close to the head of the indexed list.

Algorithm 18: GET-SUFFIX-NODE(L, i)

```
1  a ← L.F.fingers[n − 2]
2  b ← L.F.fingers[n − 1]
3  bNode ← b.node
4  saveBIndex ← b.index
5  nextBIndex ← ⌊(a.index + L.C.size) / 2⌋
6  b.index ← nextBIndex
7  if saveBIndex < nextBIndex then
8      distance ← nextBIndex − saveBIndex
9      for k ← 0 to distance − 1 do
10         bNode ← bNode.next
11 else
12     distance ← saveBIndex − nextBIndex
13     for k ← 0 to distance − 1 do
14         bNode ← bNode.prev
15 b.node ← bNode
16 if i < nextBIndex then
17     leftDistance ← i − a.index
18     rightDistance ← nextBIndex − i
19     if leftDistance < rightDistance then
20         return SCROLL-TO-RIGHT(a.node, leftDistance)
21     else
22         return SCROLL-TO-LEFT(b.node, rightDistance)
23 else
24     leftDistance ← i − nextBIndex
25     rightDistance ← L.C.size − i − 1
26     if leftDistance < rightDistance then
27         return SCROLL-TO-RIGHT(bNode, leftDistance)
28     else
29         return SCROLL-TO-LEFT(L.C.tail, rightDistance)
```

The Algorithm 18 handles the special case of the Algorithm 16 where the request is close to the tail of the indexed list.

Algorithm 19: SCROLL-NODE-TO-RIGHT(u, Δ)

```
1 for i ← 1 to Δ do
2 |   u ← u.next
3 return u
```

The Algorithm 19 simply scrolls to the right a predefined number of steps towards nodes with larger indices. The algorithm SCROLL-NODE-TO-LEFT is defined analogously for SCROLL-NODE-TO-RIGHT, yet scrolls to the left (nodes with smaller indices).

INSERT delegates to LINK-LAST if the requested index i points to the rightmost free position. Also, it delegates to LINK-FIRST if the requested index i points to the leftmost occupied position. Otherwise, the requested element y belongs between $(i-1)$st and ith list elements, and that is performed by LINK-BEFORE, which follows below.

As one can see, in Algorithm 13, line 6, we call GET-NODE. While not really an insertion procedure, we present the pseudocode so far and hereafter in the "top-down" manner, and so we proceed to GET-NODE:

A.2 Bulk insertion operations

In this section, we will review all the operations responsible for adding collections to indexed lists. The proxy procedure for inserting bulk data/collections is given below.

Algorithm 20: ADD-ALL$(L, i, \mathcal{Y})$

```
1  if i < 0 or i > L.C.size then
2  |   error "Bad index."
3  if |Y| = 0 then
4  |   return
5  if L.C.size = 0 then
6  |   SET-ALL(L, Y)
7  else if i = L.C.size then
8  |   PUSH-BACK-COLLECTION(L, Y)
9  else if i = 0 then
10 |   PUSH-FRONT-COLLECTION(L, Y)
11 else
12 |   u ← GET-NODE(L, i)
13 |   INSERT-ALL(L, Y, u, i)
```

A.2.1 Setting a collection to an empty indexed list

Algorithm 21: SET-ALL($L, \mathcal{Y}$)

```
1  u ← new node
2  u.datum ← y_0
3  L.C.head ← u
4  prevNode ← L.C.head
5  for i ← 1 to |𝒴| − 1 do
6      u ← new node
7      u.datum ← y_i
8      prevNode.next ← u
9      u.prev ← prevNode
10     prevNode ← u
11 L.C.tail ← prevNode
12 L.C.size ← |𝒴|
13 ADD-FINGERS-AFTER-SET-ALL(L, M)
```

The Algorithm 21 is called by the Algorithm 20 only when this indexed list is empty. It runs in $\Theta(\|\mathcal{Y}\|)$ time. Also, all the other list data structures run it in the same time, even the tree list.

Algorithm 22: ADD-FINGERS-AFTER-SET-ALL(L, M)

```
1 numberOfNewFingers ← GET-RECOMMENDED-NUMBER-OF-FINGERS(L)
2 ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L,
                                        numberOfNewFingers + 1,
                                        0,
                                        numberOfNewFingers,
                                        |𝒴|)
3 Δ ← ⌊L.C.size / numberOfNewFingers⌋
4 SPREAD-FINGERS(L,
                 L.C.head,
                 0,
                 0,
                 numberOfNewFingers,
                 Δ)
```

The Algorithm 22 is called in order to distribute the initial fingers throughout the list.

Algorithm 23: SPREAD-FINGERS$(L, u, i, i_f, s, \Delta)$

```
1  f ← new finger
2  f.index ← i
3  f.node ← u
4  L.F.fingers[i_f] ← f
5  i_f ← i_f + 1
6  for i ← 1 to s − 1 do
7      i ← i + Δ
8      u ← SCROLL-NODE-TO-RIGHT(u, Δ)
9      f ← new finger
10     f.index ← i
11     f.node ← u
12     L.F.fingers[i_f] ← f
13     i_f ← i_f + 1
```

The Algorithm 23 performs the actual assignment of fingers. What comes to its parameters, u is the very first node in the range over which to distribute the s distinct fingers. i is the index of u, i_f is the index of the very leftmost finger to spread. Finally, Δ is the distance between two consecutive fingers in the range over which we are spreading the fingers.

A.2.2 Push collection to back

Algorithm 24: PUSH-BACK-COLLECTION($L, \mathcal{Y}$)

```
1  prev ← L.C.tail
2  oldLast ← L.C.tail
3  foreach y ∈ 𝒴 do
4      u ← new node
5      u.datum ← y
6      u.prev ← prev
7      prev.next ← u
8      prev ← u
9  L.C.tail ← prev
10 sz ← |𝒴|
11 L.C.size ← L.C.size + sz
12 ADD-FINGERS-AFTER-APPEND-ALL(L,
                                oldLast.next,
                                L.C.size − sz,
                                sz)
```

The Algorithm 24 is responsible for appending the input collection after the tail element. It runs in $\Theta(\log N + M + \sqrt{N+M} - \sqrt{N})$ time. For the dynamic table, this operation runs in amortized $\Theta(M)$ time. For the linked list, this operation runs in exact $\Theta(M)$. What comes to the tree list, it runs this operation in $\Theta(M + \log(N+M))$ time.

Algorithm 25: ADD-FINGERS-AFTER-APPEND-ALL(L, `first`, i, M)

```
1 numberOfNewFingers ← GET-RECOMMENDED-NUMBER-OF-FINGERS(L) − L.F.size
2 if numberOfNewFingers = 0 then
3 |   L.F.𝔈.index ← L.F.𝔈.index + M
4 |   return
5 i_f = L.F.size
6 MAKE-ROOM-AT-INDEX(L, i_f, numberOfNewFingers, M)
7 Δ ← ⌊M / numberOfNewFingers⌋
8 SPREAD-FINGERS(L,
                  first,
                  i,
                  i_f,
                  numberOfNewFingers,
                  Δ)
```

The Algorithm 25 distributes the new possible fingers along the appended sub-list.

Algorithm 26: MAKE-ROOM-AT-INDEX(L, i_f, M, l)

```
1 ENLARGE-FINGER-ARRAY-WITH-EMPTY-RANGE(L,
                                         L.F.size + 1 + M,
                                         i_f,
                                         M,
                                         l)
```

The Algorithm 26 moves a suffix of fingers to the right in order to make some room for new fingers. What comes to its parameters, i_f is the index of the leftmost finger that must be shifted to the right, M is the number of fingers for which we desire to make room, and l is the number of nodes/elements that would be shifted.

A.2.3 Push collection to front

In this subsection, we will discuss the procedure for adding a collection before the first element of a given indexed list. The algorithm in question follows:

Algorithm 27: PUSH-FRONT-COLLECTION($L, \mathcal{Y}$)

```
1  oldHead ← L.C.head
2  newNode ← new node
3  newNode.datum ← y_0
4  L.C.head ← newNode
5  prevNode ← L.C.head
6  for i ← 1 to |𝒴| − 1 do
7      newNode ← new node
8      newNode.datum ← y_i
9      newNode.prev ← prevNode
10     prevNode.next ← newNode
11     prevNode ← newNode
12 prevNode.next ← oldHead
13 oldHead.prev ← prevNode
14 sz ← |𝒴|
15 L.C.size ← L.C.size + sz
16 ADD-FINGERS-AFTER-PREPEND-ALL(L, sz)
```

The Algorithm 27 prepends the input collection before the head node of the indexed list. It runs in $\Theta(M + \sqrt{N})$ time. The linked list runs it in exact $\Theta(M)$ time, and the dynamic table runs it in $\Theta(M + N)$ time. The tree list runs it in $\Theta\Big((N + M)\log(N + M) - N \log N\Big)$ time.

The Algorithm ADD-FINGERS-AFTER-PREPEND-ALL is responsible for adding a number of new fingers for indexing the prefix of the indexed list that was created by copying $\mathcal{Y}$ to its beginning. It is defined as follows.

Algorithm 28: ADD-FINGERS-AFTER-PREPEND-ALL(L, sz)

```
1 numberOfNewFingers ← GET-RECOMMENDED-NUMBER-OF-FINGERS(L) − L.F.size
2 if numberOfNewFingers = 0 then
3 |   SHIFT-FINGER-INDICES-TO-RIGHT(L, 0, sz)
4 |   return
5 MAKE-ROOM-AT-INDEX(L, 0, numberOfNewFingers, sz)
6 Δ ← ⌊sz / numberOfNewFingers⌋
7 SPREAD-FINGERS(L,
                 L.C.head,
                 0,
                 0,
                 numberOfNewFingers,
                 Δ)
```

In the Algorithm 28, the call to MAKE-ROOM-AT-INDEX shifts the entire finger array to the right in order to make room for the new upcoming fingers. Also, the call to SPREAD-FINGERS is the actual routine responsible for setting all new fingers at the beginning of the finger list. Next, we proceed to MAKE-ROOM-AT-INDEX: What comes to SPREAD-FINGERS, it is defined as follows:

The algorithm SCROLL-NODE-TO-RIGHT(u, Δ) merely produces the node that is Δ positions to the *right* from the node u.

A.2.4 Insert a collection

In this subsection, we will discuss a general method for adding data to an indexed list.

Algorithm 29: INSERT-COLLECTION($L, \mathcal{Y}$, `successor`, i)

```
1  predecessor ← successor.prev
2  prev ← predeccessor
3  foreach y ∈ 𝒴 do
4      newNode ← new node
5      newNode.datum ← y
6      newNode.prev ← prev
7      prev.next ← newNode
8      prev ← newNode
9  prev.next ← successor
10 successor.prev ← prev
11 sz ← |𝒴|
12 L.C.size ← L.C.size + sz
13 ADD-FINGERS-AFTER-INSERT-ALL(L, predecessor.next, i, sz)
```

The Algorithm 29 runs in $\Theta(M + \sqrt{N+M} - \sqrt{N}) + \mathcal{O}(\sqrt{N})$ time. In the dynamic table, the same operation runs in $\Theta(M) + \mathcal{O}(N)$ time. So does the linked list. What comes to the tree list, it runs in exact $\Theta\Big((N+M)\log(N+M) - N\log N\Big)$ time.

The algorithm ADD-FINGERS-AFTER-INSERT-ALL is responsible for distributing the new fingers after the INSERT-COLLECTION procedure. Its definition follows.

Algorithm 30: ADD-FINGERS-AFTER-INSERT-ALL(L, `firstNode`, i, M)

```
1  numberOfNewFingers ← GET-RECOMMENDED-NUMBER-OF-FINGERS(L) − L.F.size
2  i_f ← GET-FINGER-INDEX-IMPL(L, i)
3  if numberOfNewFigners = 0 then
4      SHIFT-FINGER-INDICES-TO-RIGHT(L, i_f, M)
5      return
6  MAKE-ROOM-AT-INDEX(L, i_f, numberOfNewFingers, M)
7  Δ ← ⌊M / numberOfNewFingers⌋
8  SPREAD-FINGERS(L,
                  firstNode,
                  i,
                  i_f,
                  numberOfNewFingers,
                  Δ)
```

A.3 Access operations

The most fundamental procedure of the indexed list is accessing a closest finger to a given index.

Algorithm 31: GET(L, i)

```
1 return GET-NODE(L, i).datum
```

The above algorithm relies on GET-NODE, and as such, it is also attempted to increase the indexed list entropy by slightly moving the fingers local to the ith element. It runs in $\Theta(\log N) + \mathcal{O}(N)$ time. Note that if the entropy of the indexed list is good (near the 1), we would have the running time $\Theta(\log N) + \mathcal{O}(\sqrt{N})$.

A.4 Deleting an element from the indexed list

In this subsection, we will review the deleting operations.

Algorithm 32: DELETE-AT-INDEX(L, i)

```
1  if i < 0 or i ≥ L.C.size then
2      error "Invalid index."
3  closestFingerIndex ← GET-CLOSEST-FINGER-INDEX(L, i)
4  closestFinger ← L.F.fingers[closestFingerIndex]
5  local returnValue
6  local nodeToRemove
7  if closestFinger.index = i then
8      nodeToRemove ← closestFinger.node
9      MOVE-FINGER-OUT-OF-REMOVAL-LOCATION(L,
                                           closestFinger,
                                           closestFingerIndex)
10 else
11     Δ ← i − closestFinger.index
12     nodeToRemove ← REWIND-FINGER(closestFinger, Δ)
13     SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, closestFingerIndex + 1)
14     if Δ < 0 then
15         Decrement L.F.fingers[closestFingerIndex].index
16 returnValue ← nodeToRemove.datum
17 UNLINK(nodeToRemove)
18 Decrement L.C.size
19 if MUST-REMOVE-FINGER(L) then
20     REMOVE-FINGER(L)
21 return returnValue
```

The Algorithm 32 works as follows. If the ith element is pointed by a finger, that finger is pushed out of the ith element's way. Then the element in question is unlinked and removed normally. Otherwise, the affected finger indices are decremented by a unit. It runs in $\mathcal{O}(\sqrt{N})$ time. The same algorithm on dynamic tables and linked lists run in $\mathcal{O}(N)$ time. The tree list deletes an element at an index in exact $\Theta(\log N)$ time.

Algorithm 33: GET-CLOSEST-FINGER-INDEX(L, i)

```
1  i_f ← GET-FINGER-INDEX-IMPL(L, i)
2  return NORMALIZE(L, i_f, i)
```

As the name implies, the Algorithm 33 returns the index of the finger that is closest to the ith element.

Algorithm 34: NORMALIZE(L, i_f, i)

```
1  if i_f = 0 then
2      return 0
3  if i_f = |L.F| then
4      return |L.F| − 1
5  f_1 ← L.F.fingers[i_f − 1]
6  f_2 ← L.F.fingers[i_f]
7  Δ_1 ← i − f_1.index
8  Δ_2 ← f_2.index − i
9  if Δ_1 < Δ_2 then
10     return i_f − 1
11 else
12     return i_f
```

Given a finger index i_f and the element index i, the Algorithm 34 will return either $i_f - 1$ or i_f, whichever is closer to the ith element.

Algorithm 35: MOVE-FINGER-OUT-OF-REMOVAL-LOCATION(L, f, i_f)

```
1  if L.F.size = L.C.size then
       # Here, |L.F| is either 1 or 2.
2      if L.F.sze = 1 then
3          return # The only finger will be removed in DELETE-AT-INDEX.
4      if i_f = 0 then
5          L.F.fingers[0] ← L.F.fingers[1]
6          L.F.fingers[1] ← L.F.fingers[2]
7          L.F.fingers[0].index ← 0
8          L.F.fingers[1].index ← 1
9          L.F.fingers[2] ← nil
10         L.F.size ← 1
11     else if i_f = 1 then
12         REMOVE-FINGER(L)
13         L.F.𝔈.index ← 1
14     return
   # Try push the fingers to the right:
15 if TRY-PUSH-FINGERS-TO-RIGHT(L, i_f) then
16     return
   # Could not push the fingers to the right. Push to the left
17 if TRY-PUSH-FINGERS-TO-LEFT(L, i_f) then
18     return
   # Once here, the only free spots are at the beginning of the finger list:
19 for i ← 0 to L.F.size do
20     f ← L.F.fingers[i]
21     Decrement f.index.
22     f.node ← f.node.prev
23 SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, i_f + 1)
```

The Algorithm 35 moves the finger f either to the right or left, whichever is available. If none of the above apply, pushes the finger f towards the finger list prefix.

Algorithm 36: REMOVE-FINGER(L)

```
1 Decrement L.F.size
2 CONTRACT-FINGER-ARRAY-IF-NEEDED(L)
3 L.F.fingers[L.F.size] ← L.F.fingers[L.F.size + 1]
4 L.F.fingers[L.F.size + 1] ← nil
5 L.F.fingers[L.F.size].index ← L.C.size
```

The Algorithm 36 is responsible for removing the last non-sentinel finger from the finger index array.

Algorithm 37: CONTRACT-FINGER-ARRAY-IF-NEEDED(L, C)

```
1 if ||L.F.fingers|| = 8 then
2     return
3 if (C + 1) < ||L.F.fingers|| / 4 then
4     nextCapacity ← ||L.F.fingers|| / 2
5     while nextCapacity ≥ 2 × (C + 1) × 2 and nextCapacity > 8 do
6         nextCapacity ← ⌊nextCapacity / 2⌋
7     newFingerArray ← new finger array with capacity nextCapacity
8     ARRAY-COPY(L.F.fingers,
                 0,
                 newFingerArray,
                 0,
                 nextCapacity)
9     L.F.fingers ← newFingerArray
```

The Algorithm 9 attempts to contract the internal finger index array if it is sparsely utilized.

Algorithm 38: TRY-PUSH-FINGERS-TO-RIGHT(L, i_f)

```
1 for j ← i_f to |L.F| − 1 do
2     fingerLeft ← L.F.fingers[j]
3     fingerRight ← L.F.fingers[j + 1]
4     if fingerLeft.index + 1 < fingerRight.index then
5         for i ← j down to i_f do
6             L.F.fingers[i].node ← L.F.fingers[i].node.next
7         SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, j + 1)
8         return true
9 return false
```

Makes an attempt to push a finger to the right.

Algorithm 39: SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, i_f)

```
1  for j ← i_f to L.F.size do
2      Decrement L.F.fingers[j].index
```

The Algorithm 39 decrements all the indices from the finger range $L.F.fingers[i_f, i_f + 1, \ldots, L.F.size]$.

Algorithm 40: TRY-PUSH-FINGERS-TO-LEFT(L, i_f)

```
1  if i_f = 0 then
2      f ← L.f.fingers[0]
3      Decrement f.index
4      f.node ← f.node.prev
5      SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, 1)
6      return true
7  for j ← i_f down to 1 do
8      fingerLeft ← L.F.fingers[j − 1]
9      fingerRight ← L.F.fingers[j]
10     if fingerLeft.index + 1 < fingerRight.index then
11         for k ← j to i_f do
12             f ← L.F.finger[k]
13             f.node ← finger.node.prev
14             Decrement f.index
15         SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, i_f + 1)
16         return true
17 return false
```

Makes an attempt to push a finger to the left.

Algorithm 41: REWIND-FINGER(f, Δ)

```
1  u ← f.node
2  if Δ > 0 then
3      return SCROLL-TO-RIGHT(u, Δ)
4  else
5      return SCROLL-TO-LEFT(u, −Δ)
```

The Algorithm 41 merely scrolls from the desired node towards a target node a given number of steps given.

Algorithm 42: UNLINK(L, u)

```
1  next ← u.next
2  prev ← u.prev
3  if prev = nil then
4  |   L.C.head = next
5  else
6  |   prev.next ← next
7  |   u.prev ← nil
8  if next = nil then
9  |   L.C.tail = prev
10 else
11 |   next.prev ← prev
12 |   u.next ← nil
```

The Algorithm 42 unlinks the input node from the $L.C$.

The algorithm MUST-REMOVE-FINGER(L) returns a boolean value GET-RECOMMENDED-NUMBER-OF-FIN
$L.F.size$.

A.4.1 Popping from front

The Algorithm 43 removes the very first element in the given indexed list.

Algorithm 43: POP-FRONT(L)

```
1  returnValue ← L.C.head.datum
2  Decrement L.C.size
3  L.C.head ← L.C.head.next
4  if L.C.head = nil then
5  |   L.C.tail ← nil
6  else
7  |   L.C.head.prev ← nil
8  ADJUST-ON-REMOVE-FIRST(L)
9  if MUST-REMOVE-FINGER(L) then
10 |   REMOVE-FINGER(L)
11 L.F.𝔈.index ← L.C.size
12 return returnValue
```

The Algorithm 43 runs in exact $\Theta(\sqrt{N})$ time. It removes the head element from the indexed

list. The same operations is run by a linked list in exact constant time, and by the dynamic table it is run in exact linear time. The tree list performs this operation in exact $\Theta(\log N)$ time.

The Algorithm 44 scans a packed finger prefix and, once found, moves all finger indices on the right of that location one step to the left.

Algorithm 44: ADJUST-ON-POP-FRONT(L)

```
1 i ← ∞
2 for j ← 0 to L.F.size − 1 do
3     f ← L.F.fingers[j]
4     if f.index ≠ j then
5         i ← j
6         break
7     else
8         f.node ← f.node.next
9 SHIFT-FINGER-INDICES-TO-LEFT-ONCE-ALL(L, i)
```

A.4.2 Popping from back

Algorithm 45: POP-BACK(L)

```
1  returnValue ← L.C.tail.datum
2  Decrement L.C.size
3  L.C.tail ← L.C.tail.prev
4  if L.C.tail = nil then
5      L.C.head ← nil
6  else
7      L.C.tail.next ← nil
8  if MUST-REMOVE-FINGER(L) then
9      REMOVE-FINGER(L)
10 L.F.𝔈.index ← L.C.size
11 return returnValue
```

The Algorithm 45 removes the very last element from the indexed list. It runs in amortized constant time. The same algorithm runs in dynamic tables in amortized constant time, and in linked lists it runs in exact constant time. The tree list runs this operation in exact $\Theta(\log N)$ time.

A.4.3 Removing an element range

In this subsection, we will discuss the facilities for removing entire element ranges from indexed lists. The Algorithm 46 removes the element range $L[b, b+1, \ldots, e-1]$

Algorithm 46: DELETE-RANGE(L, b, e)

```
1  removalLength ← e − b
2  if removalLength = 0 then
3      return
4  if removalLength = 1 then
5      DELETE-AT-INDEX(L, b)
6      return
7  if removalLength = L.C.size then
8      CLEAR(L)
9      return
10 fromFingerIndex ← GET-FINGER-INDEX-IMPL(L, b)
11 toFingerIndex ← GET-FINGER-INDEX-IMPL(L, e)
12 currentFingers ← GET-RECOMMENDED-NUMBER-OF-FINGERS(L)
13 nextFingers ←
       GET-RECOMMENDED-NUMBER-OF-FINGERS(L, L.C.size − removalLength)
14 fingersToRemove ← currentFingers − nextFingers
15 (r_s, r_e) ← LOAD-REMOVE-RANGE-END-NODES(L, b, e)
16 REMOVE-RANGE-IMPL(L,
                     b,
                     e,
                     fromFingerIndex,
                     toFingerIndex,
                     fingersToRemove)
17 UNLINK-NODE-RANGE(L, r_s, r_e)
18 CONTRACT-FINGER-ARRAY-IF-NEEDED(L, L.C.size)
```

The Algorithm 46 runs in $\Theta(M) + \mathcal{O}(\sqrt{N})$ time. Both the dynamic table and the linked list run this operation in $\Theta(M) + \mathcal{O}(N)$ time, and the tree list runs it in $\Theta(N \log N - (N - M) \log(N - M))$.

What comes to the CLEAR operation, it prunes the finger list such that it contains only the end-of-finger-list sentinel $L.F.\mathfrak{E}$, and unlinks all the nodes from in $L.C$, sets $L.C.head \leftarrow L.C.tail \leftarrow$ **nil**, and, finally, sets $L.F.size \leftarrow L.C.size \leftarrow 0$. Also, it contracts the actual

finger list $L.F.fingers$ to the capacity of eight fingers.

GET-RECOMMENDED-NUMBER-OF-FINGERS(L, C) returns $\lceil\sqrt{C}\rceil$.

The Algorithm 47 is responsible for computing two nodes `startNode` and `endNode` such that the node range $\langle$`startNode`, `startNode`, $\ldots$, `endNode`$\rangle$ is to be removed.

Algorithm 47: LOAD-REMOVE-RANGE-END-NODES(L, b, e)

```
1 startNode ← GET-NODE-NO-FINGER-FIX(L, b)
2 endNode ← GET-NODE-NO-FINGER-FIX(L, e)
3 if endNode = nil then
4 |   endNode ← L.C.tail
5 else
6 |   endNode ← endNode.prev
7 return (startNode, endNode)
```

The Algorithm 48 does the same as GET-NODE(L, i) (that is, obtains the ith node), yet it does not modify the local fingers in any way. We need this in order not to break the inner workings of DELETE-RANGE.

Algorithm 48: GET-NODE-NO-FINGERS-FIX(L, i)

```
1 f ← L.F.fingers[GET-CLOSEST-FINGER-INDEX(L, i)]
2 Δ ← i − f.index
3 return REWIND-FINGER(f, Δ)
```

The following algorithm does actually remove both the element and finger ranges.

Algorithm 49: REMOVE-RANGE-IMPL(L, b, e, `fromFingerIndex`, `toFingerIndex`, `fingersToRemove`)

```
1  removalFingerRangeLength ← toFingerIndex − fromFingerIndex
2  removalRangeLength ← e − b
3  if removalFingerRangeLength ≤ fingersToRemove then
4      REMOVE-RANGE-IMPL-CASE-A(L, fromFingerIndex, toFingerIndex, b, e, fingersToRemove)
5      return
6  (n_prefix, n_suffix) ← LOAD-FINGER-COVERAGE-COUNTERS(L, fromFingerIndex, toFingerIndex, b, e, fingersToRemove)
7  numberOfFingersInPrefix ← fromFingerIndex
8  numberOfFingersInSuffix ← L.F.size − toFingerIndex
9  ARRANGE-PREFIX(L, b, numberOfFingersInPrefix, n_prefix)
10 ARRANGE-SUFFIX(L, e, numberOfFingersInSuffix, n_suffix)
11 REMOVE-FINGERS-ON-DELETE-RANGE(L, fromFingerIndex, fingersToRemove, removalRangeLength)
```

Algorithm 50: REMOVE-RANGE-IMPL-CASE-A(L, `fromFingerIndex`, `toFingerIndex`, b, e, `fingersToRemove`)

```
1  copyLength ← min(L.F.size − toFingerIndex, L.F.size − fingersToRemove) + 1
2  targetIndex ← max(0, min(fromFingerIndex, toFingerIndex − fingersToRemove))
3  sourceIndex ← targetIndex + fingersToRemove
4  ARRAY-COPY(L.F.fingers, sourceIndex, L.F.fingers, targetIndex, copyLength)
5  ARRAY-FILL(L.F.fingers, L.F.size + 1 − fingersToRemove, L.F.size + 1, nil)
6  L.F.size ← L.F.size − fingersToRemove
7  removalRangeLength ← e − b
8  SHIFT-FINGER-INDICES-TO-LEFT(L, targetIndex, removalRangeLength)
9  L.C.size ← L.C.size − removalRangeLength
```

The above algorithm handles a special case.

Algorithm 51: ARRAY-FILL(X, i_f, i_t, x)

```
1  for i ← i_f to i_t − 1 do
2      X[i] ← x
```

The Algorithm 51 simply sets each of $X[i_f], X[i_f + 1], \ldots X[i_t - 1textbf]$ to x.

Algorithm 52: SHIFT-FINGER-INDICES-TO-LEFT(L, i_f, M)

```
1  for i ← i_f to L.F.size do
2      Subtract M from L.F.fingers[i].index
```

The Algorithm 52 decrements all the finger indices by M for fingers at positions $i_f, i_f + 1, \ldots L.F.size$.

Algorithm 53: LOAD-FINGER-COVERAGE-COUNTERS(L, `fromFingerIndex`, `toFingerIndex`, b, e, `fingersToRemove`)

```
1  fingerPrefixLength ← fromFingerIndex
2  fingerSuffixLength ← L.F.size − toFingerIndex
3  listPrefixFreeSpots ← b
4  listSuffixFreeSpots ← L.C.size − e
5  freeFingerPrefixSpots ← listPrefixFreeSpots − fingerPrefixLength
6  freeFingerSuffixSpots ← listSuffixFreeSpots − fingerSuffixLength
7  freeSpots ← freeFingerPrefixSpots + freeFingerSuffixSpots
   # leftRatio is a floating-point number.
8  leftRatio ← TO-FLOAT(freeFingerPrefixSpots) / TO-FLOAT(freeSpots)
9  removalRangeLength ← toFingerIndex − fromFingerIndex
10 remainingFingers ← removalRangeLength − fingersToRemove
11 leftCoveredFingers ← ⌊leftRatio × remainingFingers⌋
12 rightCoveredFingers ← remainingFingers − leftCoveredFingers
13 return (leftCoveredFingers, rightCoveredFingers)
```

The Algorithm 53 returns two integers: the first one is the number of fingers (call it n_l) covered by the removal range at its beginning, and the second one is the number of fingers (call it n_r) covered by the removal range at its ending. The idea here is that n_l fingers will be pushed to the finger list prefix, and the n_r fingers will be pushed to the finger list suffix.

Algorithm 54: ARRANGE-PREFIX(L, b, `numberOfPrefixFingers`, `numberOfFingersToMove`)

```
1 MAKE-ROOM-AT-PREFIX(L, b, numberOfPrefixFingers, numberOfFingersToMove)
2 PUSH-COVERED-FINGERS-TO-PREFIX(L, b, numberOfPrefixFingers, numberOfFingersToMove)
```

The Algorithm 54 actually pushes the above mentioned n_l fingers out of removal range.

Algorithm 55: MAKE-ROOM-AT-PREFIX(L, b, numberOfPrefixFingers, numberOfFingersToMove)

```
1  if numberOfPrefixFingers = 0 then
2      return
3  targetFingerIndex ← numberOfPrefixFingers − 1
4  freeFingerSpotsSoFar ← b − L.F.fingers[targetFingerIndex].index − 1
5  if freeFingerSpotsSoFar ≥ numberOfFingersToMove then
6      return
7  while targetFingerIndex > 0 do
8      f1 ← L.F.fingers[targetFingerIndex − 1]
9      f2 ← L.F.fingers[targetFingerIndex]
10     Δ ← f2.index − f1.index − 1
11     Add Δ to freeFingerSpotsSoFar
12     if freeFingerSpotsSoFar ≥ numberOfFingersToMove then
13         break
14     Decrement targetFingerIndex
15 if freeFingerSpotsSoFar < numberOfFingersToMove then
16     index ← b − numberOfPrefixFingers − numberOfFingersToMove
17     u ← GET-NODE-NO-FINGERS-FIX(L, index)
18     for i ← 0 to numberOfPrefixFingers - 1 do
19         f ← L.F.fingers[i]
20         f.index ← index
21         f.node ← u
22         Increment index
23         u ← u.next
```

Algorithm 53: Continuation of MAKE-ROOM-AT-PREFIX(L, b, `numberOfFingersInPrefix`, `numberOfFingersToMove`)

```
24 else
25     startFinger ← L.F.fingers[targetFingerIndex − 1]
26     index ← startFinger.index
27     u ← startFinger.node
28     for i ← targetFingerIndex to numberOfPrefixFingers - 1 do
29         f ← L.F.fingers[i]
30         u ← u.next
31         f.node ← u
32         Increment index
33         f.index ← index
```

The Algorithm 55 prepares the finger list prefix for adopting n_l more fingers.

Algorithm 54: PUSH-COVERED-FINGERS-TO-PREFIX(L, b, `numberOfPrefixFingers`, `numberOfFingersToPush`)

```
1  if numberOfPrefixFingers = 0 then
2      index ← b − 1
3      u ← GET-NODE-NO-FINGERS-FIX(L, index)
4      for i ← numberOfFingersToPush − 1 down to 0 do
5          f ← L.F.fingers[i]
6          f.index ← index
7          Decrement index
8          f.node ← u
9          u ← u.prev
10 else
11     rightmostPrefixFinger ← L.F.fingers[numberOfPrerixFingers − 1]
12     index ← rightmostPrefixFinger.index + 1
13     u ← rightmostPrefixFinger.node.next
14     B ← numberOfPrefixFingers + numberOfFingersToPush − 1
15     for i ← numberOfPrefixFingers to B do
16         f ← L.F.fingers[i]
17         f.index ← index
18         Increment index
19         f.node ← u
20         u ← u.next
```

The Algorithm 54 actually pushes the n_l fingers to the finger list prefix.

Algorithm 55: ARRANGE-SUFFIX(L, e, i_f, `numberOfSuffixFingers`, `numberOfFingersToMove`)

1 MAKE-ROOM-AT-SUFFIX(L, e, i_f, `numberOfSuffixFingers`, `numberOfFingersToMove`)

2 PUSH-COVERED-FINGERS-TO-SUFFIX(L, e, `numberOfSuffixFingers`, `numberOfFingersToMove`)

The Algorithm 54 actually pushes the above mentioned n_r fingers out of removal range.

Algorithm 56: MAKE-ROOM-AT-SUFFIX(L, e, i_f, numberOfSuffixFingers, numberOfFingersToMove)

```
1  if numberOfSuffixFingers = 0 then
2      return
3  targetFingerIndex ← L.F.size − numberOfSuffixFingers
4  freeFingerSpotsSoFar ← L.F.fingers[targetFingerIndex].index − e
5  if freeFingerSpotsSoFar ≥ numberOfFingersToMove then
6      return
7  while targetFingerIndex < L.F.size − 1 do
8      f1 ← L.F.fingers[targetFingerIndex]
9      f2 ← L.F.fingers[targetFingerIndex + 1]
10     distance ← f2.index − f1.index − 1
11     Add distance to freeFingerSpotsSoFar
12     if freeFingerSpotsSoFar ≥ numberOfFingersToMove then
13         break
14     Increment targetFingerIndex
15 if freeFingerSpotsSoFar < numberOfFingersToMove then
16     index ← L.C.size − numberOfSuffixFingers
17     u ← GET-NODE-NO-FINGERS-FIX(L, index)
18     for i ← 0 to numberOfSuffixFingers − 1 do
19         f ← L.F.fingers[L.F.size − numberOfSuffixFingers + i]
20         f.index ← index
21         Increment index
22         f.node ← u
23         u ← u.next
```

Algorithm 56: Continuation of MAKE-ROOM-AT-SUFFIX(L, e, i_f, `numberOfSuffixFingers`, `numberOfFingersToPush`)

```
24 else
25     startFinger ← L.F.fingers[targetFingerIndex + 1]
26     index ← startFinger.index − 1
27     u ← startFinger.node.prev
28     for i ← targetFingerIndex down to i_f do
29         f ← L.F.fingers[i]
30         f.index ← index
31         Decrement index
32         f.node ← u
33         u ← u.prev
```

The Algorithm 56 prepares the finger list suffix for adopting n_r more fingers.

Algorithm 57: PUSH-COVERED-FINGERS-TO-SUFFIX(L, e, numberOfSuffixFingers, numberOfFingersToPush)

```
1  if numberOfSuffixFingers = 0 then
2      index ← e
3      u ← GET-NODE-NO-FINGERS-FIX(L, index)
4      for i ← 0 to numberOfFingersToPush − 1 do
5          f ← L.F.figners[L.F.size − numberOfFingersToPush + i]
6          f.index ← index
7          Increment index
8          f.node ← u
9          u ← u.next
10 else
11     leftmostSuffixFinger ← L.F.fingers[L.F.size − numberOfSuffixFingers]
12     index ← leftmostSuffixFinger.index
13     u ← leftmostSuffixFinger.node
14     for i ← 0 to numberOfFingersToPush − 1 do
15         f ← L.F.fingers[L.F.size − numberOfSuffixFingers − i − 1]
16         u ← u.prev
17         f.node ← u
18         Decrement index
19         f.index ← index
```

The Algorithm 57 actually pushes the n_r fingers to the finger list suffix.

Algorithm 58: REMOVE-FINGERS-ON-DELETE-RANGE(L, `fromFingerIndex`, `numberOfFingersToRemove`, `removalRangeLength`, n_{prefix})

```
1  if numberOfFingersToRemove ≠ 0 then
2      copyLength ← L.F.Size
3                   − fromFingerIndex
4                   − numberOfFingersToRemove
5                   − n_prefix
6                   + 1
7      ARRAY-COPY(L.F.fingers,
                  fromFingerIndex + n_prefix + numberOfFingersToRemove,
                  L.F.fingers,
                  fromFingerIndex + n_prefix,
                  copyLength)
8      ARRAY-FILL(L.F.fingers,
                  L.F.size − numberOfFingersToRemove + 1,
                  L.F.size + 1,
                  nil)
9      Decrement L.F.size by numberOfFingersToRemove
10 SHIFT-FINGER-INDICES-TO-LEFT(L,
                                fromFingerIndex + n_prefix,
                                removalRangeLength)
11 Decrement L.C.size by removalRangeLength
```

The Algorithm 58 actually removes the overdue fingers from the finger list $L.F$.

Algorithm 59: UNLINK-NODE-RANGE(L, `startNode`, `endNode`)

```
1  currentNode ← startNode
2  local nextNode
3  prevStartNode ← startNode.prev
4  nextEndNode ← endNode.next
5  do
6      nextNode ← currentNode.next
7      currentNode.datum ← nil
8      currentNode.prev ← nil
9      currentNode.next ← nil
10     currentNode ← nextEndNode
11 while currentNode ≠ nextEndNode;
12 if prevStartNode = nil then
13     L.C.head ← nextEndNode
14     nextEndNode.prev ← nil
15 else if nextEndNode = nil then
16     prevStartNode.next ← nil
17     L.C.tail ← prevStartNode
18 else
19     prevStartNode.next ← nextEndNode
20     nextEndNode.prev ← prevStartNode
```

The Algorithm 59 does the actual unlinking of the nodes from the removed range.

Next, we will summarize the running times of all list implementations/operations:

Table A.1: Comparison of running times

Op / List	Table	Linked	Tree	Indexed
PUSH-FRONT	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(\sqrt{N})$
PUSH-BACK	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(1)$
INSERT(i)	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\Theta(\log N)$	$\Theta(\log N) + \mathcal{O}(\sqrt{N})$
GET	$\Theta(1)$	$\mathcal{O}(N)$	$\Theta(\log N)$	$\mathcal{O}(\log N) + \mathcal{O}(\sqrt{N})$
POP-HEAD	$\Theta(N)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(\sqrt{N})$
POP-TAIL	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$	$\Theta(1)$
DELETE	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\Theta(\log N)$	$\Theta(\log N) + \mathcal{O}(\sqrt{N})$
PUSH-COL-F	$\Theta(N+M)$	$\Theta(M)$	$\Theta((N+M)\log(N+M) - N\log N)$	$\Theta(M + \sqrt{N+M})$
PUSH-COL-B	$\Theta(M)$	$\Theta(M)$	$\Theta(M + \log(N+M))$	$\Theta(M + \sqrt{N+M} - \sqrt{N})$
INSERT-COL	$\mathcal{O}(N) + \Theta(M)$	$\mathcal{O}(N) + \Theta(M)$	$\Theta((N+M)\log(N+M) - N\log N)$	$\Theta(M) + \sqrt{N+M} - \sqrt{N}) + \mathcal{O}(\sqrt{N})$
DEL-R	$\mathcal{O}(N) + \Theta(M)$	$\mathcal{O}(N) + \Theta(M)$	$\Theta(N\log N - (N-M)\log(N-M))$	$\Theta(M) + \mathcal{O}(\sqrt{N})$

Appendix B Miscellanea

In this Appendix, we will go through auxiliary results.

B.1 Linked lists

Customary linked lists come in two flavours: *singly-linked lists* and *doubly-linked lists*. Both of them consist of nodes and interlinks between consecutive nodes. The singly-linked lists have only forward-links towards the next node, while the doubly-linked lists also have the backward-links.

B.1.1 Neareast node optimization

Unlike on singly-linked lists, in doubly-linked lists we can speed up element access. Instead of always starting the scan for the ith element from the head node, we find out in constant time the closest terminal node and start moving from it. For example, if we want to access the 9998th element of an 10 000 element list, it makes no sense to start scanning from the head node. Instead, we assume the tail node as the current node, and traverse the current node two times via backward-links.

On singly-linked list, the average running time is given by

$$\frac{1}{N}\sum_{i=1}^{N} i = \frac{1}{N}\frac{1}{2}N(N+1) = \frac{N}{2} + \frac{1}{2}.$$

What comes to the doubly-linked list, the average running time is given by

$$\frac{1}{N}\sum_{i=1}^{N} \min(i, N-i) = \begin{cases} D_1 \stackrel{\text{def}}{=} \dfrac{1}{N}\left(\displaystyle\sum_{i=1}^{(N-1)/2} i + \sum_{i=1}^{(N+1)/2} i\right) & \text{if } N \text{ is odd,} \\ D_2 \stackrel{\text{def}}{=} \dfrac{2}{N}\left(\displaystyle\sum_{i=1}^{N/2} i\right) & \text{if } N \text{ is even.} \end{cases} \tag{B.1}$$

Above, we have that

$$D_1 \stackrel{\text{def}}{=} \frac{N}{4} + \frac{1}{2} + \frac{1}{4N}$$

and

$$D_2 \stackrel{\text{def}}{=} \frac{N}{4} + \frac{1}{2}.$$

From the above, we see that, on average, accessing elements in the doubly-linked list requires around 2 times less element traversals than in singly-linked list.

B.2 Dynamic table expansion/contraction schemes

In this section, we will analyse how the choice of expansion/contraction schemes affects the running time of the PUSH-BACK/POP-BACK operations, respectively.

B.2.1 Arithmetic expansion scheme

To recap, in arithmetic expansion scheme we choose an integer constant $d \in \mathbb{N}$, the initial capacity $m \in \mathbb{N}$, and whenever we run PUSH-BACK(X, x) on a full dynamic table, we make the internal array $||X|| + d$ slots large. Now, let $E(n, m, d)$ be the total work of pushing n elements to the tail of a dynamic table with initial internal array capacity m and expansion factor d. We claim that the following equation holds:

$$E(n, m, d) = \overbrace{\sum_{k=0}^{\left\lceil \frac{n-m}{d} \right\rceil} (kd + m)}^{A} + \overbrace{\left(n - m - \left\lceil \frac{n-m}{d} \right\rceil d \right)}^{R}. \tag{B.2}$$

Above, A is the total work of expanding and filling (entirely) the internal array sufficiently many times in order to accommodate n elements, and R denotes the number of elements we could have removed from the full dynamic table such that its side becomes n.

Next, in order to prove Equation B.2 we need a couple of Lemmas.

Lemma 1. *If* $d, N > 1$ *and* $(N-1) \mod d \neq 0$,

$$\left\lceil \frac{N-1}{d} \right\rceil = \left\lceil \frac{N}{d} \right\rceil.$$

Proof. Let

$$\frac{N-1}{d} = a + \frac{b}{d},$$

where $a \in \mathbb{N}$ and $b \in \{1, 2, \ldots, d-1\}$. Now, we have that

$$\frac{N}{d} = a + \frac{b+1}{d},$$

where $b + 1 \in \{2, 3, \ldots, d\}$. At this point we see that

$$\left\lceil \frac{N-1}{d} \right\rceil = \left\lceil a + \frac{b}{d} \right\rceil = a + 1 = \left\lceil a + \frac{b+1}{d} \right\rceil = \left\lceil \frac{N}{d} \right\rceil.$$

□

Lemma 2. *Given positive integers $N, d > 1$ and assuming that $(N-1) \mod d = 0$, we have that*

$$\left\lceil \frac{N}{d} \right\rceil = \frac{N-1}{d} + 1.$$

Proof. The condition $(N-1) \mod d = 0$ implies that $N-1$ is divisible by d. Since $N-1$ is divisible by d, we can set

$$\frac{N-1}{d} \stackrel{\text{def}}{=} a \in \mathbb{N}.$$

Now, we must have that

$$\frac{N}{d} = a + \frac{1}{d},$$

and so

$$\left\lceil \frac{N}{d} \right\rceil = \left\lceil a + \frac{1}{d} \right\rceil = a + 1.$$

Finally,

$$\left\lceil \frac{N}{d} \right\rceil = \frac{N-1}{d} + 1.$$

□

Now, we need an inductive proof for Equation B.2:

Lemma 3. *Given parameters $m, d \in \mathbb{N}$, and denoting by n the number of elements to append, the equation B.2 holds.*

Proof. For $C(n, m, d)$ and $n \in \mathbb{N}_0$, we have a recurrence relation:

$$E(n, m, d) = \begin{cases} n & \text{if } n \in \{0, 1, \ldots, m\} \\ E(n-1, m, d) + 1 & \text{if } n > m \text{ and array has more space,} \\ E(n-1, m, d) + n & \text{if } n > m \text{ and array has no space.} \end{cases}$$

The first case is self-evident. For any $n \in \{0, 1, \ldots, m\}$, $E(n, m, d) = n$, so we concentrate on the second case. Since the array can accommodate at least one more element without expanding by d array components, we must have

$$\frac{n-1-m}{d} < \left\lceil \frac{n-1-m}{d} \right\rceil.$$

Now,

$$\begin{aligned} E(n-1, m, d) + 1 &= \sum_{k=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil} (kd + m) + n - 1 - m - \left\lceil \frac{n-1-m}{d} \right\rceil d + 1 \\ &= \sum_{k=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil} (kd + m) + n - m - \left\lceil \frac{n-1-m}{d} \right\rceil d \end{aligned}$$

By Lemma 1 (via setting $N = n - m$), we have

$$\left\lceil \frac{n-1-m}{d} \right\rceil = \left\lceil \frac{n-m}{d} \right\rceil.$$

Now, we see that when there is room for one more element, we have

$$E(n-1, m, d) + 1 = E(n, m, d).$$

What comes to the third equation, since the array is full, and needs expansion by $d > 0$ array components, we must have

$$\frac{n-1-m}{d} = \left\lceil \frac{n-1-m}{d} \right\rceil.$$

Now,

$$\begin{aligned} E(n-1, m, d) + n &= \sum_{k=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil} (kd + m) + \left(n - 1 - m - \left\lceil \frac{n-1-m}{d} \right\rceil d \right) + n \\ &= \sum_{k=0}^{\frac{n-1-m}{d}} (kd + m) + n - 1 - m - \frac{n-1-m}{d} d + n \\ &= \sum_{k=0}^{\frac{n-1-m}{d}} (kd + m) + n - 1 - m - (n - 1 - m) + n \\ &= \sum_{k=0}^{\frac{n-1-m}{d}} (kd + m) + n. \end{aligned}$$

Next, let us proceed further. Since we have

$$\frac{n-1-m}{d} = \left\lceil \frac{n-1-m}{d} \right\rceil,$$

(in other words, $(n-1-m)/d$ is an integer) we must also have by Lemma 2 (setting $N = n-m$)

$$\left\lceil \frac{n-m}{d} \right\rceil = \frac{n-1-m}{d} + 1.$$

Now,

$$\begin{aligned}
E(n,m,d) &= \sum_{k=0}^{\left\lceil \frac{n-m}{d} \right\rceil} (kd+m) + \left(n-m-d\left\lceil \frac{n-m}{d} \right\rceil\right) \\
&= \sum_{k=0}^{\frac{n-1-m}{d}+1} (kd+m) + \left(n-m-d\left(\frac{n-1-m}{d}+1\right)\right) \\
&= \sum_{k=0}^{\frac{n-1-m}{d}} (kd+m) + \overbrace{\left(\frac{n-1-m}{d}+1\right)d}^{A} + m + \overbrace{\left(n-m-(n-1-m+d)\right)}^{B} \\
&= \sum_{k=0}^{\frac{n-1-m}{d}} (kd+m) + \overbrace{n-1-m+d}^{A} + m + \overbrace{(1-d)}^{B} \\
&= \sum_{k=0}^{\frac{n-1-m}{d}} (kd+m) + n \\
&= E(n-1,m,d) + n.
\end{aligned}$$

□

Next, we need to derive the closed form of $E(n,m,d)$:

$$\begin{aligned}
E(n,m,d) &= \sum_{k=0}^{\left\lceil \frac{n-m}{d} \right\rceil} (kd+m) + \left(n-m-d\left\lceil \frac{n-m}{d} \right\rceil\right) \\
&= d \sum_{k=1}^{\left\lceil \frac{n-m}{d} \right\rceil} k + m\left(\left\lceil \frac{n-m}{d} \right\rceil + 1\right) + n - m - d\left\lceil \frac{n-m}{d} \right\rceil \\
&= \frac{d}{2}\left(\left\lceil \frac{n-m}{d} \right\rceil + 1\right)\left\lceil \frac{n-m}{d} \right\rceil + m\left\lceil \frac{n-m}{d} \right\rceil + n - d\left\lceil \frac{n-m}{d} \right\rceil \\
&= \frac{d}{2}\left\lceil \frac{n-m}{d} \right\rceil^2 + \left(m - \frac{d}{2}\right)\left\lceil \frac{n-m}{d} \right\rceil + n.
\end{aligned}$$

From the last formula in the above equality chain we can conjecture that

$$E(n,m,d) = \Theta\left(\left\lceil \frac{n-m}{d} \right\rceil^2\right) = \Theta\left(\frac{(n-m)^2}{d^2}\right) = \Theta((n-m)^2) = \Theta(n^2).$$

Finally, we conclude that – in the arithmetic expansion scheme – the amortized running time of PUSH-BACK an element is

$$\frac{1}{n}\Theta(n^2) = \Theta(n).$$

B.2.2 Geometric expansion scheme

In geometric expansion scheme, we are given a positive integer $m \in \mathbb{N}$ and a real value $q > 1$. The m denotes the initial capacity of the array that comprises a dynamic table, and $q > 1$ is the ***expansion factor***. Next, when the internal array of X is full, we create a new internal array X' with $||X'|| = \lfloor q||X|| \rfloor$ and copy X to X'. Clearly, expansion runs in $\Theta(N)$ time, yet it is only performed when the underlying array under the dynamic table becomes full. In Lemma 4, we prove the following result.

Lemma 4. *Appending an element to a dynamic table with geometric expansion scheme runs in amortized constant time.*

Proof. Suppose that the initial capacity of a dynamic table T is $m \in \mathbb{N}$ and whenever we need to enlarge the array in order to make room for successive pushes to back, we enlarge the internal array by the factor of $q > 1$. Now, as a slight technicality, we must have $\lfloor qm \rfloor > m$, or, namely, q must be sufficiently large in order to trigger enlarging of T for the first time (and, thus, for all the successive).

Suppose the total accumulated work for adding n elements to T is

$$W = m + mq + mq^2 + \cdots + \overbrace{mq^k}^{\approx n}.$$

We require k to be the smallest integer such that $mq^k \geq n$, which leads us to the following inequalities:

$$\begin{aligned} mq^k &\geq n \\ q^k &\geq \frac{n}{m} \\ \log_q q^k &\geq \log_q \left(\frac{n}{m}\right) \\ k &\geq \log_q n - \log_q m. \end{aligned}$$

Since k is required to be the smallest integer satisfying the previous inequality, we can set $k = \lceil \log_q n - \log_q m \rceil$. Also,

$$\begin{aligned} qW = mq + mq^2 + \cdots + mq^{k+1} \Rightarrow W - qW &= m(1 - q^{k+1}) \\ \Rightarrow W &= m\frac{1 - q^{k+1}}{1 - q}. \end{aligned}$$

Since $k = \lceil \log_q n - \log_q m \rceil$, we obtain

$$\begin{aligned} W &= m\frac{1-q^{\lceil \log_q n - \log_q m\rceil+1}}{1-q} \\ &\leq m\frac{1-q\cdot q^{\lceil \log_q n\rceil}}{1-q} \\ &\leq m\frac{1-q\cdot q^{\log_q n+1}}{1-q} \\ &= m\frac{1-q^2 n}{1-q} \\ &= \frac{m}{1-q} - \frac{mq^2 n}{1-q}. \end{aligned}$$

Now we have that

$$\begin{aligned} \frac{W}{n} &\leq \frac{m}{n(1-q)} - \frac{mq^2}{1-q} \\ &\leq \frac{m}{1-q} - \frac{mq^2}{1-q} \\ &= \frac{m(1-q^2)}{1-q} \\ &= \frac{m(1+q)(1-q)}{1-q} \\ &= m(1+q), \end{aligned}$$

which is constant since m and q are fixed parameters independent of n. (In our implementation, we set $q = 2$ and $m = 8$.) □

B.2.3 Arithmetic contraction scheme

In arithmetic contraction scheme, we choose $d \in \mathbb{N}$, and whenever we run Pop-Back, if there are d array components unused at the tail of the internal array X, we make its capacity $||X|| - d$ array components long. Suppose that the size of the dynamic table is n and $d, m \in \mathbb{N}$. Now, the total work of running Pop-Back until the table becomes empty is given by

$$\begin{aligned} C(n,m,d) &= n + \sum_{i=0}^{\left\lceil \frac{n-m}{d} \right\rceil - 1} (di+m) \\ &= \begin{cases} n & \text{if } n \in \{0,1,\ldots,m\}, \\ C(n-1,m,d)+n & \text{if } n > m \text{ and can contract the tabe,} \\ C(n-1,m,d)+1 & \text{if } n > m \text{ and can't contract the table.} \end{cases} \end{aligned} \tag{B.3}$$

Now, we will prove the equation B.3. Since $C(n, m, d) = n$ for all $n \in \{0, 1, \ldots, m\}$ is self-evident, we need to prove only the two last definitions. We start from the middle one. The condition that we can contract the table is $n > m$ and $(n - 1 - m) \mod d = 0$, so we must have

$$\begin{aligned}
C(n-1,m,d) + n &= n + (n-1) + \sum_{i=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil - 1} (di + m) \\
&= n + (n-1) + \sum_{i=0}^{\frac{n-1-m}{d} - 1} (di + m) \\
&= n + (n-1) + \sum_{i=0}^{\frac{n-1-m}{d}} (di + m) - d\left(\frac{n-1-m}{d}\right) - m \\
&= n + (n-1) + \sum_{i=0}^{\frac{n-1-m}{d}} (di + m) - (n-1-m) - m \\
&= n + \sum_{i=0}^{\frac{n-1-m}{d}} (di + m).
\end{aligned}$$

Now, we need to prove that

$$\frac{n-1-m}{d} = \left\lceil \frac{n-m}{d} \right\rceil - 1.$$

If we set above $N = n - m$, we get

$$\frac{N-1}{d} = \left\lceil \frac{N}{d} \right\rceil - 1.$$

The result follows from Lemma 2. What comes to the third case equation, the condition that we cannot contract the table is $(n - 1 - m) \mod d \neq 0$, and so, we have

$$\begin{aligned}
C(n-1,m,d) + 1 &= (n-1) + \sum_{i=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil - 1} (di + m) + 1 \\
&= n + \sum_{i=0}^{\left\lceil \frac{n-1-m}{d} \right\rceil - 1} (di + m) \\
&= n + \sum_{i=0}^{\left\lceil \frac{n-m}{d} \right\rceil - 1} (di + m) \\
&= C(n,m,d).
\end{aligned}$$

Above, if we set $N = n - m$, the result follows from the Lemma 1.

At this point, we may derive the closed form of $C(n, m, d)$):

$$\begin{aligned} C(n,m,d) &= n + \sum_{i=0}^{\left\lceil \frac{n-m}{d} \right\rceil - 1} (di + m) \\ &= n + m\left\lceil \frac{n-m}{d} \right\rceil + d \sum_{i=1}^{\left\lceil \frac{n-m}{d} \right\rceil - 1} i \\ &= n + m\left\lceil \frac{n-m}{d} \right\rceil + \frac{d}{2}\left\lceil \frac{n-m}{d} \right\rceil \left[\left\lceil \frac{n-m}{d} \right\rceil - 1 \right]. \end{aligned}$$

Finally, the running time of popping back of a list until it becomes empty is

$$\begin{aligned} \Theta(C(n,m,d)) &= \Theta\left(n + m\left(\frac{n-m}{d} \right) + \frac{d}{2}\left(\frac{n-m}{d} \right)\left(\frac{n-m}{d} - 1 \right) \right) \\ &= \Theta(n^2). \end{aligned}$$

At this point, we can conclude that the running time of popping the back of the list under arithmetic contraction scheme is

$$\frac{\Theta(n^2)}{n} = \Theta(n).$$

B.2.4 Geometric contraction scheme

The geometric contraction scheme is governed by the following Lemma:

Lemma 5. *Suppose we are given a dynamic table X. Let $0 < q_t < q_c = \gamma q_t < 1$, where $\gamma > 1$ (subject to $\gamma < 1/q_t$). The idea is that when $\alpha(X)$ drops below q_t, the dynamic table (with capacity $||X||$) is contracted to a dynamic table with capacity $q_c||X||$.*

Claim: While $\alpha(X) \in [q_t, q_c]$ ***and*** *the contraction of the dynamic table happens (when $\alpha(X)$ drops below q_t), the* Pop-Back *operation runs in* ***amortized constant time*** *for each element.*

Proof. In order to prove this claim, we will rely on *accounting method*. To this end, for each Pop-Back we will charge credit worth

$$1 + \frac{1}{\gamma - 1}$$

while $\alpha(X) \in [q_t, q_c]$. Since accumulation of credit happens on behalf of $q_c||X|| - q_t||X||$

elements, the total accumulated credit is

$$\begin{aligned}(q_c||T|| - q_t||T||)(1 + 1/(\gamma - 1)) &= (q_t\gamma||T|| - q_t||T||)(1 + 1/(\gamma - 1)) \\ &= q_t||T||(\gamma - 1)(1 + 1/(\gamma - 1)) \\ &= q_t||T||(\gamma - 1 + 1) \\ &= q_t||T||\gamma \\ &= q_c||T||.\end{aligned}$$

Since $A = q_c||X|| - q_t||X||$ covers the cost of deleting before the contraction occurs, and $B = q_t||X||$ is enough to pay for the actual contraction, we see that the total effort is $A + B = q_c||X||$, which is covered by the credits accumulated while deleting elements under condition of $\alpha(X) \in [q_t, q_c]$.

Finally, it is worthwhile to note that

$$\lim_{\gamma \to 1+} \left(1 + \frac{1}{\gamma - 1}\right) = \infty,$$

which implies that choosing γ too close to 1 might increase the amortized running time of single POP-BACK operation indefinitely. Basically, the closer γ is to 1 from the right, the closer the running time of POP-BACK is to $\Theta(N)$. Usually, throughout computer scientific literature most often $\gamma = 2$ ($q_t = 1/4, q_c = 1/2$, or $q_t = 1/3, q_c = 2/3$). □