MASTER THESIS

DYNAMIC ELIAS-FANO ENCODING

Giulio Ermanno Pibiri

SUPERVISOR
Rossano Venturini

MASTER DEGREE
in
COMPUTER SCIENCE AND NETWORKING

Department of Computer Science
Department of Information Engineering

University of Pisa
Scuola Superiore Sant'Anna
Italy

a.y. 2013/2014

Colophon

This document was typeset starting from the typographical `classicthesis`, which is available for both LaTeX and LyX at:

http://code.google.com/p/classicthesis/

However, I heavily customized the template to satisfy my endless pursuit of layout perfection.

The dissertation was written using LaTeX MAKER 4.3 and the graphical help of Matlab R2013a, OmniGraffle Professional, Grapher, Keynote and Numbers. Moreover, Eclipse and Sublime Text make code writing a pleasant activity.

The machine that has made possible the birth of this dissertation is my beloved MacBook Pro, OS Version 10.10. Thank you once again, my old friend.

Giulio Ermanno Pibiri

To my family, endless source of love and inspiration.

To Arianna, whose heart is worth this effort and all others I will do.

Abstract

Data is extremely heterogeneous: it is presented to us in a large variety of formats. There is a basic need to find efficient ways to store these kind of data so that access and manipulation is facilitated.

In all practical applications, *space-efficiency* and *fast access* to data are key driving-parameters in the design of possible solutions.

In particular, the so-called *succinct data structures* have acquainted a lot of attention in recent years for their double promising goal: compress data with performance close to the information-theoretical lower bound while supporting *exceptionally fast access* to data paying a negligible, lower order term, space factor.

In this related context, we studied the problem of storing *monotone sequences of integers* and how to solve it using succinct data structures. Fundamental to this dissertation will be the *Elias-Fano integer encoding* of such sequences.

While most theoretical and practical results on Elias-Fano encoding regard *statically compressed* data structures, little attention has been devoted to dynamic ones. Therefore, in this dissertation we tackle the problem of applying this compression strategy to a *dynamic scenario*, showing achieved results and trade-offs. We will show that if such sequences grow in an *append-only* way (new integers are added only at the end of the sequence) the resulting data structure takes only a *negligible* space more and *very small* time degradation with respect to a static counter part.

A *fully dynamic representation* supports insertions and/or deletions of integers in any position of the sequence, at a price of a higher access time. We provide tools to understand how to mitigate such problem and find the right trade-off between time and space complexities.

In conclusion, some interesting applications of these succinct data structures are illustrated.

Acknowledgments

I come writing this page after a long journey of five years of academic studies. From the guy that left high school to the person I am now, I feel profoundly changed both in a professional and personal way. This page is meant to express my sincere gratitude to all people I have interacted with and make this change possible.

Universities of Florence and Pisa have been excellent cultural environments, full of intellectual stimulations and friends.

From my Bachelor degree studies I have to thank all the guys of Computer Engineering. This goes especially to Alessandro T. for his affinity of ideas and aptitude to study, to Gioia L., Daniele M. and Francesco G.. We have a lot more to share in the future.

These two last years in Pisa would not have been the same without the support of all dear friends I found in the Computer Science & Networking guys. I would like to especially thank Stefano R., Stefano L., Thomas F., Fabio L. and Francesco D. F. in the promise that our friendship will last forever.

The choice of this Thesis along with all the things I have learnt, was possible thank to the high quality of the offered courses: I think I have finally found my own field. Therefore a special thank goes to all professors involved in this Master Program: I have learnt from *all* of you.

In particular, I will never find enough words to thank Rossano Venturini for having supervised this work and all insightful conversations we had. He has been a patient supervisor, always at the right moment for suggestions.

Last but not least words of gratitude are for where heart is. Giovanni and Isabella are the best parents one could ever dream on. Their unconditional love, wise advices and support are my life-guide. The formidably strength and respect they provide me is a unique gift. Fiammetta is a lovely sister and growing up together has been an amazing experience. The sense of peace, calm and love Arianna transmits has been necessary and fundamental to me during these years.

These are the people with whom it is worth spending a life, therefore this Thesis is dedicated to you.

Florence, Italy
January, 2015

Contents

1 INTRODUCTION 1
1.1 This Work 2
1.1.1 Chapters overview 2

I BACKGROUND AND TOOLS 4

2 BASIC CONCEPTS AND NOTATION 5
2.1 Time and Space complexities 5
2.2 Asymptotic growth 6
2.2.1 Anonymous functions 7
2.3 Sequences and Bitvectors 8
2.3.1 Operations 9
2.4 Succinct data structures and bounds 10
2.5 Information-theoretical lower bounds 10
2.6 Entropy 12
2.7 Model of computation 12
2.8 Programming model 15
2.8.1 Java performance tuning 15
2.8.2 APIs 16

3 RANK & SELECT PRIMITIVES 18
3.1 Classic design 18
3.1.1 Rank 18
3.1.2 Select 20
3.2 Implementation 22
3.2.1 Population counting 22
3.2.2 Broadword programming 23

4 ELIAS-FANO INTEGER ENCODING 25
4.1 Elias-Fano scheme 25
4.1.1 Encoding 26
4.1.1.1 An example 27
4.1.2 Succinct representation of sequences 28

II ELIAS-FANO STRUCTURES 30

5 APPEND-ONLY 31

5.1 Known sequence length 31
5.1.1 Algorithmic description 31
5.1.2 Operations 33
5.1.3 Space complexity 35
5.1.3.1 Minimizing extra storage 38
5.1.3.2 Practical tool 40
5.1.4 Time complexity 41
5.1.4.1 Amortized analysis 41
5.1.4.2 Append 42
5.1.4.3 De-amortization 43
5.1.4.4 Access 43
5.1.4.5 Next Greater Or Equal 44
5.2 Unknown sequence length 45
5.2.1 Algorithmic description 45
5.2.1.1 Adaptive strategy 47
5.2.1.2 Reconstructing 48
5.2.2 Operations 50
5.2.3 Space complexity 51
5.2.4 Time complexity 55
5.2.4.1 Append 55
5.2.4.2 Access 55
5.2.4.3 Next Greater Or Equal 56
5.3 Implementation key-points and APIs 57
5.4 Brief summary 63
6 DYNAMIC 64
6.1 Algorithmic description 64
6.2 Theoretical improvement 67
6.3 Operations 70
6.3.1 Add/Remove 70
6.3.2 Access 72
6.3.3 Next Greater Or Equal 74
6.4 API 74
6.5 Brief summary 75
III PRACTICAL IMPACT 76
7 EXPERIMENTAL RESULTS 77
7.1 Memory footprint 77
7.2 Time measures 82
7.2.1 Append 82
7.2.2 Access 82
7.2.3 Next Greater Or Equal 85
7.2.4 Iterating over the sequence 85
7.2.5 Adding/Removing 86
8 APPLICATIONS 87

8.1 Compressed in-memory graphs 87
8.1.1 Preliminaries 87
8.1.2 The structure 90
8.1.3 Reordering of identifiers 92
8.2 Building a crawling index 93
8.3 Dynamic Inverted Lists 95

9 CONCLUSION AND RELATED WORKS 98
9.1 Summary of results 98
9.2 Open problems 99

BIBLIOGRAPHY 102

This page intentionally left blank

1 Introduction

The incredibly fast growth of computer hardware and software technologies in the past few decades have *radically* changed the world in which we are living: in the way we buy things (E-commerce); in the way we relate to other people (Social Networks); in the way we access and search information (Search Engines). Computers play a key role in everything we do and think of and it would be impossible to recall their applications in just few lines.

Instead, we would like to focus on the kind of *data* they can produce and process. We call *information* the *ultimate interpretation*, being from a human being or an automaton, of data by a precise *semantics*. As an example, the World Wide Web represents the biggest source of information for news regarding any aspects of the human life. In fact, computers search, manipulate and publish information over the Internet, mostly.

Because of the fast processors and large memories we can build nowadays, computations are also more data-intensive than ever. Therefore, the efficient treat and management of such information is so crucial that a new generation of scientists was born to accomplish these goals.

Data is *extremely heterogeneous*: it is presented to us in a large variety of formats. From completely unstructured data such as text, to semi-structured data such as HTML and XML files, to structured one we can find in relational databases. There is a basic need to find efficient ways to store these kind of data so that access and manipulation is facilitated. To achieve this, *knowledge of algorithms and data structures* has become indispensable for engineers since the their wise use largely outperform any kind of computer hardware improvement.

In all practical applications, *space-efficiency* and *fast access* to data are key driving-parameters in the design of possible solutions. In modern processors architectures, we can feed cache memories with the mostly used data so that their access is extremely fast if compared to main memory, disk and network in order. Therefore there is a fundamental need for *data compression* and an entire field of active research is working on it.

In particular, the so-called *succinct data structures* [18, 5] have acquainted a lot of attention in recent years for their double promising goal: compress data with performance close to the information-theoretical lower bound while supporting *exceptionally fast access*[1] to data paying a negligible, lower order term, space factor called *redundancy*. As we will notice later on in the text, succinct data structures should be engineered as much as possible since, otherwise, their redundancy factor may not be negligible at all due to large constants hidden by the asymptotic notation [30, 14, 33].

1 Usually constant time.

Succinct data structures benefit a lot from some important trends in computer architecture too. The first concerns *memory access patterns*. Since a bad exploitation of memory hierarchy is probably the first source of performance degradation in computer systems, we wish to induce *sequential and predictable* access patterns to memories.

Secondly, the now ubiquitous 64-bits architectures can process larger chucks of data in fewer accesses. This is of particular help in storing and retrieving data and *broadword programming techniques*, along with *bit-level manipulations* [30], exploit this new hardware feature.

1.1 This Work

This Thesis finds its origins in the aforementioned context and it is motivated by the above considerations. We studied the problem of storing *monotone sequences of integers* and how to solve it using succinct data structures. Fundamental to this dissertation will be the *Elias-Fano integer encoding* of such sequences [10, 11, 12]. As for illustrative purposes, it has been successfully applied to inverted-indices' compression, showing excellent compression ratio and query evaluation times [26]. Another meaningful example is its potential use in storing graphs: Facebook's engineers have recently adopted this compression strategy building Unicorn, an online, in-memory social graph-aware indexing system [7].

In particular, most theoretical and practical results on Elias-Fano encoding regard *statically compressed* data structures. In this dissertation we tackle the problem of applying this compression strategy to a *dynamic scenario*, showing achieved results and trade-offs. Monotone integer sequences we take into account are not static, but can grow over time. We will show that if such sequences grow in an *append-only* way (new integers are added only at the end of the sequence) the resulting data structure takes only a *negligible* space more than the static counter part with a *very small* time degradation.

We also take into account random deletions and additions from such compressed sequences. While being *not yet competitive* in query time with the append-only structures, it only implies a very small space redundancy and it is the first attempt to dynamize Elias-Fano compressed integer sequences.

These Elias-Fano succinct data structures will be natural good candidates for any engineered computation which handles large amount of integers applying compression on the fly. The implemented structures form a library, publicly available under proper license, in the hope that it will be useful for applications and further research.

1.1.1 Chapters overview

The Thesis is subdivided into three main parts.

1. *Background and Tools*. The very first part contains all needed background to fully comprehend subsequent chapters. More specifically, Chapter 2 deals with basics tools in algorithmic analysis and data compression that will appear in all the dissertation, such as asymptotic notation, succinct bounds, entropy and models of computation.

   Chapter 3 illustrates the best state-of-the-art algorithms and techniques to build efficient *rank & select* succinct data structures as they are the basis for all following chapters. Broadword programming concepts and their use in the design of fast rank & select structures are presented.

   Finally Chapter 4 shows the Elias-Fano integer encoding, the strategy at heart of this Thesis, with examples and bounds.

2. *Elias-Fano Structures*. This second part focuses on the main work performed for this Thesis and present the main achieved results.

   Chapter 5 introduces two append-only Elias-Fano-compressed succinct data structures along with their algorithmic descriptions and performance guarantees. As already mentioned, the most important result is showing they introduce a negligible space factor and very small time degradation with respect to their static counter parts.

   Chapter 6 naturally makes a step forward and describes how to fully dynamize the previously introduced append-only structures.

3. *Practical Impact*. This final third part is intended to show the practicality of the implemented structures. Therefore, Chapter 7 presents the large number of tests performed for what concern both space and time measurements. In this chapter we will confirm and stress the quality of the developed theoretical models.

   Chapter 8 contains three selected applications, one for each structure, where we show how the implemented library could be used. The first concerns the efficient storage of large in-memory graphs; the second the building of a crawling index for web pages; the third the storage of dynamic inverted lists.

   Finally, Chapter 9 sums up salient features of the presented material. We also point the reader to future directions and open problems the Author would be pleased to work on.

Part I

Background and Tools

2 Basic Concepts and Notation

This chapter deals with important tools and concepts that must be known to fully comprehend subsequent analysis. It also introduces part of the notation used in the text, along we some examples and further readings.

If not strictly necessary, we will omit redundant parentheses, with the aim of obtaining a compact and elegant math display. Throughout the text, numbering starts at zero, to spontaneously reflect how we are used to think when programming [9].

Basics

A set is, informally, an *unordered collection* of items. Given a set S, we denote with $|\cdot|$ the *cardinality operator*, such that $|S|$ is its cardinality, i.e., how many elements it contains. We indicate with $[n]$ the ordered set of the first n natural numbers, i.e., $[n] = \{0, 1, \ldots, n-1\}$ and $|[n]| = n$. Unless otherwise specified, all logarithms will be binary, i.e., $\lg n = \log_2 n$.

The *bit* is the minimum quantum of information of computer systems. A *byte* (B) is a unit of measure for bits, equivalent to 8 bits. In our practical experiments we assume[1] 1 KB (*Kilo byte*) $= 10^3$ B and 1 MB (*Mega byte*) $= 10^6$ B. 1 GB (*Giga byte*) $= 10^9$ B and 1 TB (*Tera byte*) $= 10^{12}$ B.

1 http://www.nist.gov/pml/wmd/metric/prefixes.cfm

2.1 Time and Space complexities

Analysis and resources of an algorithm

Analyzing an algorithm means predicting the resources that it will use. We refer to the voluntarily generic term *resources* as the set of all kind of artifacts an algorithm needs to properly compute its task and eventually terminate.

This set may include several items such as communication bandwidth and computer hardware, but most often we are primarily interested in knowing the *computational time* of an algorithm and its *space occupancy*.

We usually have that algorithms compute their job in a time which depends on the size of their input data. The memory they reference varies with this input data dimension too.

Algorithms and mathematical functions

Therefore, in this sense, an algorithm $\mathscr{A}$ differs from a function in the mathematical sense. However, we can use the same algebraic notation of functions to express algorithmic computations: starting from some input elements of an input data set I, $\mathscr{A}$ produces some output values belonging to an output data set O, by means of a function $f : I \to O$ which corresponds to the body definition of the algorithm itself, namely the set of instructions it is made up of.

Let us call the dimension of the input data set $n = |I|$. Then we have that

$$\mathscr{A} : I \xrightarrow{f} O.$$

Time and space complexities

We define the the computational time of an algorithm as the time it takes to perform all the elementary operations that uniquely define it. The space occupancy of an algorithm measures the amount of memory it uses during its execution.

The *time complexity* of algorithm $\mathscr{A}$ will be indicated as $T_{\mathscr{A}}(n)$ and its *space complexity* as $S_{\mathscr{A}}(n)$. If the space complexity refers to a data structure $\mathscr{D}$, we will use $S_{\mathscr{D}}(n)$ as well. A data structure is a way of representing and organizing information so that we can easily access and modify it. Whenever clear from the context we are referring to an algorithm or a data structure, we can relax the notation and drop the lower script specification letter.

2.2 Asymptotic growth

Sometimes we are able to write precise formulas describing the exact time or space occupancy of an algorithm. However, such precision is usually not worth the *effort* to obtain it: when we deal with big input sizes, all multiplicative constants and additive terms in our formulas are dominated by the problem size itself [28, 6].

Whenever this situation happens, we are studying the *asymptotic efficiency* of algorithms and we only care about the order of growth of interesting quantities, namely execution time and memory.

Asymptotic notation

We typically use the *asymptotic notation* to describe the above mentioned quantities that was first introduced by the mathematician P. G. H. Bachmann in 1894. This notation, however, applies to *functions* and we should pay attention to not misuse it. We will make an extensive use of asymptotic notation throughout this dissertation.

Asymptotic tight bounds

Definition 2.2.1. For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1, c_2, \eta \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n), \forall\, n \ge \eta\}.$$

Therefore whenever we write that $f(n) \in \Theta(g(n))$ we mean there exist some positive constants, c_1 and c_2, for which our function $f(n)$ is sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for all n sufficiently large.[2] Usually we will drop this set notation and introduce some syntactic sugar to allow us to just write $f(n) = \Theta(g(n))$. In conclusion $f(n)$ is limited both from above and below by $g(n)$ starting from some value of the problem size: $g(n)$ is an asymptotically tight bound for $f(n)$.

As claimed before, when we use this notation, we do *not* specify for which choice of constants the condition $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ is true. We are interested in describing the asymptotic behaviour of $f(n)$, with a practical and easy-to-understand notation.

In general, if $p(n) = \sum_{i=0}^{d} \alpha_i n^i$ is a polynomial of degree d, where coefficients $\{\alpha_0, \alpha_1, \ldots, \alpha_d\}$ are constants, we have that $p(n) = \Theta(n^d)$.

2 As an example, we can verify that $1/4n^2 - 5n = \Theta(n^2)$. We wonder if there exist a choice of constants c_1, c_2 and η such that

$$c_1 n^2 \le \frac{1}{4}n^2 - 5n \le c_2 n^2,$$

$\forall\, n \ge \eta$. Dividing by n^2, it is quite easy to verify that for a choice of $c_1 = 1/4$, $c_2 = 1/84$ and $\eta = 21$, the above double-inequality is satisfied.

Using this simple consideration, we derive that, since each constant can be though of as a polynomial of degree 0, we can express any constant function with $\Theta(n^0) = \Theta(1)$.[3]

3 There is a little abuse in this notation, since we do not specify which is the variable going to infinity [6]. However, unless otherwise specified, we can tolerate this abuse whenever the variable is clear from the context.

Definition 2.2.2. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \exists \text{ positive constants } c, \eta \text{ such that } 0 \leq f(n) \leq cg(n), \forall\, n \geq \eta\}.$$

This notation is used to specify *asymptotic upper bounds*: if $f(n) = O(g(n))$ we mean that $f(n)$ is upper bounded by $g(n)$ for all n sufficiently large.[4]

Asymptotic upper bounds

Notice that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, so the first writing is a stronger condition than the second.

4 As an example, we can verify that $2n^2 = O(n^3)$. By applying the definition, for $c = 1$ and $\eta = 2$ the condition is satisfied.
Similarly we can show that $n^3 \neq O(n^2)$, since we will obtain $n \leq c$, which we cannot make always true, because c is a constant and n tends to infinity.

Definition 2.2.3. For a given function $g(n)$, we denote by $o(g(n))$ the set of functions

$$o(g(n)) = \{f(n) : \forall \text{ positive constant } c, \exists\, \eta > 0 \text{ such that } 0 \leq f(n) < cg(n), \forall\, n \geq \eta\}.$$

The main difference with the big-Oh notation is that while it requires $0 \leq f(n) \leq cg(n)$ for *some* positive constant c, the little-Oh notation requires that $0 \leq f(n) < cg(n)$ for *all* $c > 0$.

This notation is suggesting us that function $f(n)$ becomes *negligible* with respect to $g(n)$ when n grows infinitely[5], i.e., when

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

5 As an example, consider $\sqrt{n} = o(n)$. Then we can always determine $\eta = 1/c^2, \forall\, c > 0$. Similarly $\lg n = o(n)$. But $n/4 \neq o(n)$, in fact we would obtain the condition $c > 1/4$ which does *not* hold for *any* choice of c.

Similar definitions can be given for Ω and ω notations, less used in this dissertation. We suppose the reader be familiar with these concepts [6].

2.2.1 Anonymous functions

Of particular interests for our purposes is when the asymptotic notation is used inside a formula, say an equation or inequality. In such cases it is used to indicate some *anonymous function* that we do not care to mention.

For the sake of clarity, consider $3n^2 + n + 4 = 3n^2 + \Theta(n)$. This means that $3n^2 + n + 4 = 3n^2 + f(n)$, for some function $f(n) \in \Theta(n)$ (in this specific case $f(n) = n + 4$). $f(n)$ is clearly what we call an anonymous function: we do not specify its name and express it with the term $\Theta(n)$.

Practical use cases of such anonymous functions are very common. A famous, well known, example is when we write *recurrence relations*, as we do for Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. When we would like to express space occupancies of algorithms, anonymous functions are very useful too, as we will see next. We will use them in this way, mostly.

2.3 Sequences and Bitvectors

Many times in this thesis we will mention *sequences*, *arrays*, *list*, *vectors* and *strings*. While being all mathematically equivalent, each term has a different flavour according to the topic/field we are referring to. Typically, when speaking, usual common sense fills up the gap between abstraction and concrete usage of these terms and leads us to comprehension. In this way, if we mention a "vector" while speaking about algebraic linear operators, we immediately get the corresponding meaning of the term, and do not think (usually) about C++'s `std::vector`.

The elements these objects are made up of are also different. Arrays and vectors are usually constituted by numbers (integers or reals), while strings by characters from a given alphabet.

Trying to understand the common properties these terms have, we notice that *sequence* is the most abstract of all. Then we give the following, tentative, definition.

Sequences

Definition 2.3.1. Let Σ be a set, called the *alphabet*, of $\sigma = |\Sigma|$ symbols. Then for any set S of $n = |S|$ symbols drawn from Σ, we define a *sequence* s as a couple $s = \langle S, \pi \rangle$, where $\pi : [n] \to S$ defined as $i \mapsto s_i$.

Therefore, $s = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ and s_i denotes the i-th element of s. We indicate with $|s| = n$ the *length* of s. Note that π is a *labelling function*, associating an item of S with a position in s. Whenever we need to specify a subrange of s'values, we use the algebraic syntax for intervals, e.g., $s[i, j)$ refers to the subsequence $\langle s_i, s_{i+1}, \ldots, s_{j-1} \rangle$.

Now, if we are talking about algebraic vectors, $\Sigma = \mathbb{K}$ field; if we refer to strings of a text, an alphabet may be $\Sigma = \{c | c \in \{\text{set of ASCII symbols}\}\}$[6].

A particular case of interest for us is when $\Sigma = \{0, 1\}$. In this case, we call any sequence a *bitvector*.

6 http://www.ascii-code.com

Bitvectors

Definition 2.3.2. A *bitvector* is a sequence s, with $\Sigma = \{0, 1\}$.

For example, the following sequence is a bitvector

$$s = 010010010001100101001011101001101000$$

More commonly, we care about the number of bits set to 1 in a bitvector of length u bits. For this purpose, we will indicate with $b(n, u)$ a bitvector having n out of u bits set to 1. In the example given below, $s = b(15, 36)$.

2.3.1 Operations

We now need to define the main operations we would like to implement on a sequence. Given a generic sequence s of length n, defined over an alphabet Σ, for any $x \in \Sigma$, there are four basic operations we require to implement:

- $append_s(x)$ adds x to the end of s and increment n by one, i.e., $\pi(n) = x$;
- $get_s(i)$ returns the i-th element of s, namely $get_s(i) = s_i$, $\forall\ i \in [n]$;
- $rank_s(x, i)$ returns the number of elements equal to x in $s[0, i)$, $\forall$ $i \in [n]$;
- $select_s(x, i)$ returns the position of the i-th element equal to x in s or just -1 if there is no such element in s.

As an example, if s is the following string:

$\boldsymbol{f}$	$\boldsymbol{r}$	$\boldsymbol{e}$	$\boldsymbol{s}$	$\boldsymbol{h}$	$\boldsymbol{n}$	$\boldsymbol{e}$	$\boldsymbol{s}$	$\boldsymbol{s}$
0	1	2	3	4	5	6	7	8

then

$$\begin{array}{lll} get_s(0) = \boldsymbol{f} & rank_s(\boldsymbol{s}, 4) = 1 & select_s(\boldsymbol{r}, 1) = 1 \\ get_s(5) = \boldsymbol{n} & rank_s(\boldsymbol{s}, 8) = 2 & select_s(\boldsymbol{e}, 3) = -1 \\ \dots & \dots & \dots \end{array}$$

Whenever clear from the context we are performing such operations on s, we can drop the subscript letter.

In the case we are referring to a known bitvector, we can indicate the above operations with: $get(i)$, $rank_{0/1}(i)$ and $select_{0/1}(i)$ respectively (now, the subscript is used to specify the symbol).

As an example, if $b(7, 13)$ is:

1	1	0	0	0	1	1	0	1	1	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

then

$$\begin{array}{lll} get(5) = 1 & rank_0(6) = 3 & select_0(4) = 7 \\ get(10) = 0 & rank_1(4) = 2 & select_1(11) = -1 \\ \dots & \dots & \dots \end{array}$$

2.4 SUCCINCT DATA STRUCTURES AND BOUNDS

We have already introduced the concept of a data structure. Obviously, information can be represented in many different ways and, consequently, there is no data structure that fits well for *every* problem we would like to address.

Of particular interest for this thesis, are the so-called *succinct data structures*. They constitute a specific class of *compressed data structures*: data is first compressed using an ad-hoc compressor; secondly a *redundancy space-factor* is added to the data structure, in order to support the requested operations. Usually this additional factor depends on the time we want to perform the designed operations on the data. What we typically find out is the usual *trade-off* between space and time complexities: the less we compress the faster we go and viceversa. We will experimentally confirm this trend many times, in Chapter 5, 6 and 7.

Redundancy space factor

Therefore, what we expect is that our compressed data structure' s memory footprint is given by the sum of two contributions:

- a compressed-data-dependent factor;
- a redundancy factor.

What we mean by *succinct* is that, adopting certain techniques, the redundancy factor can be made small compared to the data size, i.e., a *lower order term* of the data-compressed factor. Usually it is so small to become a *negligible factor* indeed.

Now, suppose that a sequence s takes, *at least*, m bits to be represented. Then, by *using a succinct data structure for*[7] s, we will use $o(m)$ additional bits of information to implement all needed operations. The total space occupancy will now be: $m + o(m)$ bits. This quantity is also referred to as a *succinct bound*.

7 We also say *a succinct representation of*.

Succinct bounds

However a *theoretical* negligible redundancy of $o(n)$ bits does not always imply a *practically* negligible redundancy too. Some real-life datasets exhibit less space with a $O(n)$-bit data structures than with a $o(n)$-bit one. This is due the "hiding constants" effect of asymptotic notation. Clear distinction between theoretical and practical implementation of such data structures will be pointed out, whenever necessary, throughout the text.

Hiding constants

Next, we should understand *how many bits are at least needed to encode the data structures we will work with,* in order to be able to compare theoretical and experimental results. This will be the subject of the next section.

2.5 INFORMATION-THEORETICAL LOWER BOUNDS

In general, given a set X of combinatorial objects, the minimum amount of bits we need to *uniquely* identify each object $x \in X$ is $\lceil \lg |X| \rceil$ bits. This quantity is what we call an *information-theoretical lower bound*. This means, generally speaking, there is no hope of doing better than this for

the set X. But here is the trick, we shall say. In fact, if we further restrict X by making some assumptions on its elements, we can beat the lower bound.

As an example, consider the set X of all bitvectors of length u. Then we have than $|X| = 2^u$ and therefore we need at least u bits. If we restrict our attention to the class of all bitvectors of length u having n bits set to **1**, then we have $|X| = \binom{u}{n}$, i.e., all possible ways we can select n **1**s from a set of u bits. Therefore, for a bitvector $b(n,u)$ the information-theoretical lower bound is

$$\left\lceil \lg \binom{u}{n} \right\rceil \text{ bits.} \tag{1}$$

By doing some math, i.e., using Newton's coefficient formula[8] and Stirling's factorial approximation[9], we obtain

$$u \lg u - n \lg n + (u-n) \lg \frac{1}{u-n} - O(\lg u) \text{ bits}$$

and, by adding and subtracting $(u-n)\lg u$, we finally get

$$n \lg \frac{u}{n} + (u-n) \lg \frac{u}{u-n} - O(\lg u) \text{ bits.}$$

This function is symmetric and has a maximum in $n = u/2$, meaning that we can concentrate our attention to the values of the function for $0 \le n \le u/2$ [29], obtaining

Information-theoretical lower bound for bitvectors

$$\mathscr{B}(n,u) = n \lg \frac{u}{n} + O(n) \text{ bits,} \tag{2}$$

which represents a more practical way of describing the minimum number of bits we need to encode $b(n,u)$. Now a succinct representation of $b(n,u)$ will take $\mathscr{B}(n,u) + o(\mathscr{B}(n,u))$ bits.

Concerning a general sequence s of symbols drawn from an alphabet Σ of size σ, in absence of any additional information the best we can do is to encode each symbol of Σ using $\lg \sigma$ bits. Doing so, any sequence consisting of n symbols will take

Information-theoretical lower bound for sequences

$$\mathscr{S}(n,\sigma) = n \lg \sigma \text{ bits.} \tag{3}$$

In many cases, this bound is *not satisfactory* and can indeed be considered as a *worst case bound* since it corresponds to the space required by the, plain, binary representation of s. As similarly noticed before, we succinctly represent each sequence of n symbols drawn from that alphabet, with $\mathscr{S}(n,\sigma) + o(\mathscr{S}(n,\sigma))$ bits.

8 Recall that:
$$\binom{u}{n} = \frac{u!}{n!(u-n)!}, \ \forall u \ge n.$$

9 Recall that:
$$x! \sim \sqrt{2\pi x}\left(\frac{x}{e}\right)^x, \ \forall x \ge 0.$$

2.6 ENTROPY

The father of *Information Theory*, Claude Elwood Shannon, left us a powerful tool that he names *entropy* [29]. He was concerned with the problem of defining the *information content* of a discrete random variable $\mathbf{x} : \Sigma \to \mathbb{R}$, with distribution $p_x = \mathbb{P}\{\mathbf{x} = x\}$, $\forall x \in \Sigma$.

He defined the entropy of $\mathbf{x}$ as

$$H(\mathbf{x}) = \sum_{x \in \Sigma} p_x \lg \frac{1}{p_x} \text{ bits} \times \text{value}.$$

The quantity $\lg(1/p_x)$ bits is also called the *self-information* of $x \in \Sigma$. $H(\mathbf{x})$ give us an idea of how many bits we need to encode each value of Σ.

Let now s be a string of n characters drawn from an alphabet Σ. Let denote n_c the number of times character c occurs in s. Assuming *empirical frequencies as probabilities* (the larger is n, the better the approximation) [27], i.e., $p_c \simeq n_c/n$, we can similarly consider s as a random variable assuming value c with probability p_c. Therefore, we can define the entropy of a string[10] s as

10 This generally applies to a "text", since a text can be seen as a "long string".

$$H_0(s) = \sum_{c \in \Sigma} \frac{n_c}{n} \lg \frac{n}{n_c} \text{ bits}.$$

This quantity is also knows as the 0-*th order empirical entropy* of s. Notice that $nH_0(s)$ tell us how many bits should be required to represent s in binary.

How many bits do I need?

In the context of this Thesis, we are mainly interested in the case when s is a bitvector $b(n, u)$. In this setting, we have only two symbols and we immediately get

$$H_0(b(n,u)) = \frac{n}{u} \lg \frac{u}{n} + \frac{u-n}{u} \lg \frac{u}{u-n} \text{ bits},$$

which, as already noticed, is a symmetric function and we can focus on the interval $n \in [0, u/2]$. In this case we obtain $H_0(b(n,u)) = (n/u)\lg(u/n)$ bits. In conclusion, this quantity is related to the information-theoretical lower bound $\mathcal{B}(n,u)$ with the following relation

$$uH_0(b(n,u)) = \mathcal{B}(n,u) + O(\lg u) \text{ bits}.$$

2.7 MODEL OF COMPUTATION

In order to properly quantify the two fundamental complexities of an algorithm and, therefore, be able to analyze the algorithms proposed in this dissertation, we need a *model of computation*.

We will adopt a classic, theoretical, model of computation called *word-RAM* (Random Access Machine) model. This model tries to approach the

RAM model

behaviour of real (also modern, if properly tuned) processors. This goal is achieved by defining two main ingredients:

Elementary Operations

1. a set of *elementary operations* (elops), all executed in constant time worst case, $O(1)$;
2. a set of memory cells, that we can dereference, without loss of generality, with the integers in $[u]$. Every cell is accessed in $O(1)$ worst case. If $u = 2^w$, w bits are transferred from memory to the CPU in constant time.

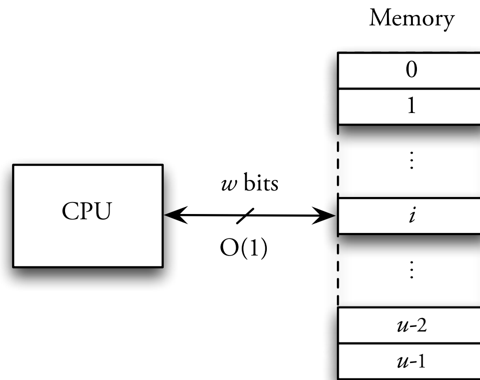

Figure 1: RAM model.

However it is under our responsibility to not abuse the word-RAM model, pretending it includes some instructions it should not. For illustrative purposes, think of an elementary sorting instruction. Than it means that we are able to sort in just one instruction, in constant time. This is, of course, a very unrealistic assumption [6].

We will not report here the complete set of elementary operations defined by the model, since it will be a tedious and less informative task. We can imagine this set coincide with the set of instruction of *ubiquitous C programming language*, including: arithmetic operations, bitwise operations, data movements and control structures (loops and conditionals).

Word-size

w is the word-size and it is a crucial parameter of the model. It defines how big is the memory used by the model. The only assumption made on w is that it is at least n, our input problem size, i.e.,

$$w = \Omega(\lg n) \text{ bits.}$$

This assumption is consistent to the fact that if w were not an $\Omega(\lg n)$, then we would not even be able to index all elements in our data ($n > u$).

The assumption is also revealing the power and the generality of the model: w changes with n and so does the modelled processor. This model was called *trans-dichotomous* and was formalized by Michael Fredman and Dan Willard in [13].

Space requirements of our algorithms will be measured in the *number of words addressed by the algorithm* and, similarly, the space of a data structure will be measured in the number of words this data structure consists in.

Analogously, we can express the time complexity of an algorithm $\mathscr{A}$ as the number of elementary operations performed during its execution and we will write

- $T_{\mathscr{A}}(n)$ [elops]
- $S_{\mathscr{A}}(n)$ [w]

Finally, notice that when $w = 32$ or $w = 64$, the model fairly matches the commodity PCs we use everyday.

New models of computation

The RAM model seems, however, too simple under many aspects. Primary, the fact that we only assume the presence of a single CPU and a monolithic block of memory. Revolutionary changes have been made to such a simple structure over the last decades: we are just in the middle of the multi/many-core technological trend and can enjoy faster and bigger memory hierarchies. In particular, nowadays computations are more data-intensive applications than ever. When we previously wrote that $n \to \infty$, it seems not just a theoretical assumption.

2-level memory model

For these reasons, scientists have developed new models of computations so that algorithms can meet and adopt to this new scenario of computing [32]. Figure 2 shows a *2-level memory model*, along with an overview of memory hierarchies of modern processors and their dimensions. The closer a memory level is to the CPU, the faster it is but also smaller and costly.

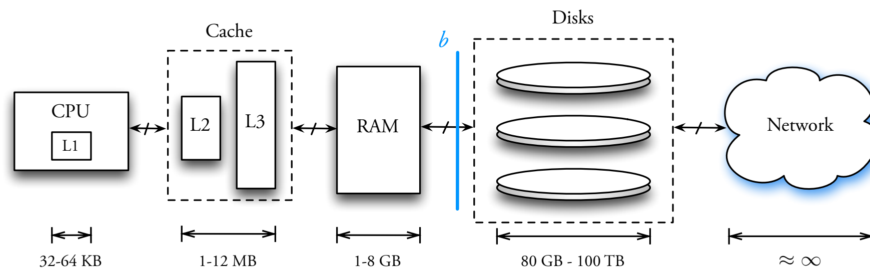

Figure 2: 2-level memory model. The picture shows a typical choice of b between Disks and RAM. In this case we can transfer several KBs with a single I/O.

The model is characterized by a *demarcation boundary* b that we can decide where to place. This boundary splits our model in two memory components: a fast one and bounded one; and a slower but unbounded one. Usual choices for b's placement are: between Cache and RAM or between RAM and Disks.

In any case, we are able to transfer several words (a *block*) across the two chosen memory levels, instead of only one as in the RAM model. This hypothesis is introduced to reflect how modern memories are read and written. Clearly, each memory level has its own technology-dependant parameters, such as dimension and access time. The operation we use to access, in R/W mode, a memory level is called an I/O (*input/output*) operation. Using this model, the efficiency of algorithms will be measured in number of performed I/Os.

We would like to conclude this section with a table reporting the main order of magnitudes of memory accesses and everyday-operations machine perform. It is worth noting that these numbers are not *interesting*

per se since they clearly depend on application/implementation-specific features (hardware, operating system, network technology, programming language...), but they are meaningful to *understand* the order of magnitude of such operations and be able to compare them.

Operation	Time
L1 cache reference	1 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
RAM reference	100 ns
Send 2 KB over a commodity network	0.4 μs
Disk seek (random I/O)	4 ms

Table 1: Latency Numbers Each Programmer Should Know: http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html.

2.8 Programming model

"An algorithm must be seen to be believed." D. E. Knuth

We firmly believe there is no algorithm without an implementation.

Therefore, all the algorithms we will describe in this text have been implemented using the *Java*[11] *programming language*. However, we only use a subset of Java which turns out to be a point of strength of our code rather than a limitation. We put emphasis on those constructs that can be easily found (or equivalent) in *any* programming language. In this way, translating our implementation in C++ code is, as an example, as simple as a one-to-one statement translation.

Whenever in the Thesis we mention some external library as a dependency, we will provide a reference to it.

11 https://www.java.com/en/

2.8.1 Java performance tuning

We have used version 1.8, the latest release of Java, along with the native HotSpot Virtual Machine designed by Oracle that supports JIT (Just-In-Time compilation) to improve code performance. All codes have been compiled with `javac` using the optimizing compilation flag `-O` and `-g:none` to reduce `.class` files footprint by removing debugging information. Java code formatting adheres to the *style rules*[12] of Google.

As two main guidelines to efficient code production we always recommend to

1. "stay high-level" since good algorithmic and data-structure design is the best promising source of performance improvement and, moreover, *"premature optimization is the root of all evil."* [19];

2. use a *code profiler* to better understand where most computing time is spent by the program (a good tool for Java is YourKit[13]).

12 https://google-styleguide.googlecode.com/svn/trunk/javaguide.html

13 https://www.yourkit.com/

In the following we report the most important Java performance tips the Author has used to guarantee a *robust* and *efficient* code. Further suggestions and details will be given in Chapters 5 (Section 5.3) and 7.

OBJECTS. We wrote *simple constructors* and keep a *minimal inheritance hierarchy*. We also *reduce the number of temporary objects* avoiding to create them in loops or in very frequently called methods. We *reuse* objects whenever possible.

LOOPS. We do *not use any method calls* nor *casts* within loops. We make use of `final` local variables to test for the terminating condition of a loop, since local variables are faster accessed than instance/class variables. We also avoid using wrapper methods to access class fields, but just allow a `protected`-like access.

CLASS METHODS. We use *only iterative methods* to avoid repeated stack frames allocation due to recursive calls.

Although there is no explicit mechanism in Java to inline methods, we wrote short and simple methods and make them `final` to encourage inlining by the compiler.

We also limit the use of `Exceptions` which degrade performance and should *only be used for error conditions*, not for control flow.

OPTIMIZED UTILITY DATA STRUCTURES We wrote some utility data structures that are used as *backing structures* for the developing of advanced ones. *No range checking* is performed since those classes should not be used in a standalone fashion. *No primitive data wrapper objects* (like `Integer`, `Long`, ecc.) have been used but their primitive data types which are faster and do not suffer from extra space overhead. Those utility data structures are *pre-sized* whenever possible, since this improves performance significantly. Whenever we need an array copy, we use the optimized built-in function `System.arraycopy()` (which is faster than a simple `for` copying-loop).

General *strength reduction techniques* (substitute complex arithmetic operations with less expensive ones) have been adopted.

2.8.2 APIs

When writing code for a library intended to be used by others, a fundamental component is providing a proper documentation, describing the content of such library. As a good practice, we will present the methods of the implemented classes in *application programming interfaces* (APIs) that list the name of the class along with all implemented method signatures and a short description of each method [28].

In order to clearly show what our notation for APIs consists in, we report an example (excerpts) for the Java `ArrayList` class[14] that we will use in subsequent chapters.

We call a *client* code a fragment of code that uses a specific API. The purpose of an API is to *separate* as much as possible the provided code implementation from the client. The client, in fact, should know nothing

14 Full API description available at `http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`.

about the implementation; conversely the API should not rely on particular assumption on the client code (e.g., type checking or proper input parameters). Then, no matter who the client is, we can reuse the implemented library again and again. An elegant interpretation of this concept considers the API as a *contract* between the library implementation and the client code. As programmers, our goal is to *honor* the terms of the contract.

ArrayList *API*

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Description	Method
Appends the specified element to the end of this list.	`boolean add(E e)`
Inserts the specified element at the specified position in this list.	`void add(int index, E element)`
Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.	`boolean addAll(Collection<? extends E> c)`
Removes all of the elements from this list.	`void clear()`
Returns true if this list contains the specified element.	`boolean contains(Object o)`
Returns the element at the specified position in this list.	`E get(int index)`

3 Rank & Select primitives

The main character of this chapter is the bitvector, the ultimate representation of any computerized kind of information[1]. As we have seen in Section 2.3.1, there are few *primitives* we would like to implement efficiently on a bitvector. Among them *rank* and *select* have recently gained lot of attention since they constitute *essential building blocks* in the design of succinct data structures.

For a generic bitvector b of n bits, $rank(i)$ returns the number of bit set (1s) in the semi-open interval $[0, i)$ (we can also say "to the left of") and $select(i)$ returns the position of the i-th bit set, $\forall i \in [n]$. We can just focus on ranking and selecting set bits, since it is sufficient to observe that[2] $\forall i \in [n]$

$$rank_{0/1}(select_{0/1}(i)) = i - 1 \quad (4)$$

$$rank_{0/1}(i) = i - rank_{1/0}(i) \quad (5)$$

$$select_{0/1}(i) = i + rank_{1/0}(select_{0/1}(i)) \quad (6)$$

The formulas above fully describe the inner relations among the primitives. In the following we review most important and useful techniques used to build a data structure able to efficiently support these two operations. As we will see in Chapter 4, *rank & select* can be also employed for succinct data structures that store *monotone sequences of integers*.

1 It is interesting to note how *small* a single bit of information is. Yet, it allowed us to conceive the most powerful machine and largest artifact in the history of humanity, such as the *computer* and the *web*. By the way *"Big things have small beginnings."*

2 Please, pay attention to subscript symbols!

3.1 Classic design

Because the design of aforementioned primitives usually follows different approaches, we treat them separately.

3.1.1 Rank

Constant-time solution

The classical constant-time solution we are going to describe is quite simple and elegant [18].

Given our bitvector b of n bits, we follow the general design of:

1. Determining the *basic block size* to divide b along with an efficient way of counting the number of set bits inside each basic block.
2. Building an *index* (possibly multi-layered) that stores *aggregation information* for a specified number of basic, consecutive, blocks. This aggregation information is organized in *superblocks* and rep-

resents, for each superblock, how many set bits we have *up to* that superblock.

Almost all approaches to ranking leverage on this guideline and differ by how they choose block sizes; number of index layers and way of counting in a basic block.

The classical choice is to have just two layers for the index. We divide the bitvector in basic blocks of size $B_0 = \lfloor \lg n/2 \rfloor = O(\lg n)$; consecutive blocks are grouped together in superblocks of size $B_1 = B_0 \lfloor \lg n \rfloor = \lfloor \lg n \rfloor \lfloor \lg n/2 \rfloor = O(\lg^2 n)$. Therefore, the index can be structured using two arrays:

1. I_1 of $O(n/\lg^2 n)$ entries, also called *first-layer counters*, each of them storing the number of set bits up to its corresponding superblock. Each entry requires $O(\lg n)$ bits. This array constitute the first-layer of the index and takes overall $O(n/\lg n)$ bits;

2. I_2 of $O(n/\lg n)$ entries, also called *second layer counter*, each of them storing the number of set bits up to its corresponding basic block. Each entry requires $O(\lg \lg^2 n) = O(\lg \lg n)$ bits[3]. Array I_2 constitute the second-layer of the index and takes overall $O(n \lg \lg n/\lg n)$ bits.

3 Using definition 2.2.2, we have that

$$\lg \lg^2 n \leq c \lg \lg n = \lg \lg^c n,$$

which, passing to the argument of the logarithms is satisfied $\forall c \geq 2$, $\forall n \geq 0$.

In addition to this 2-level structure, we also add a pre-computed, *universal* (i.e., independent on the bitvector) table, T, storing the number of set bits for each position of every bitvector of length $\lfloor \lg n/2 \rfloor$. This table takes

$$O(2^{\lfloor \lg n/2 \rfloor} \lg n \lg \lg n) = O(\sqrt{n} \lg n \lg \lg n) \text{ bits.}$$

This is built in order to count in constant time inside a basic block (just access the proper entry in the table). The following picture sums up what explained.

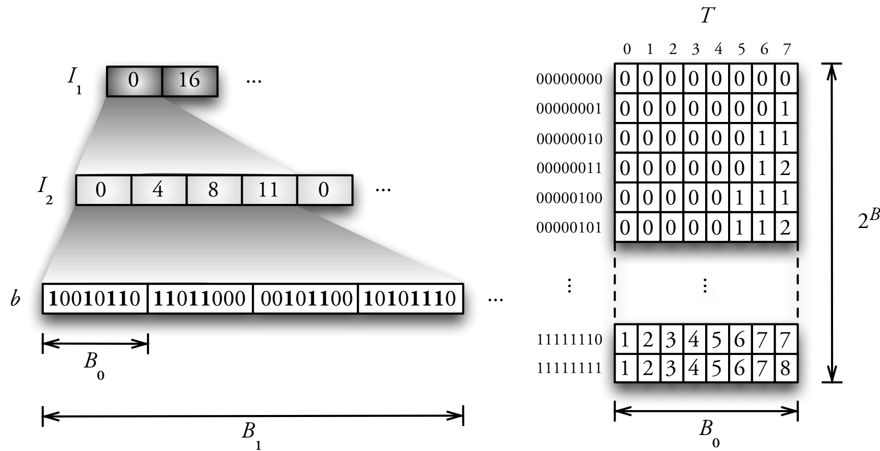

Figure 3: Ranking structure with a 2-level index and precomputed table. In this example $B_0 = 8$ bits and $B_1 = 32$ bits.

Summing up these three contributions, we obtain

$$O\left(\frac{n}{\lg n} + \frac{n \lg \lg n}{\lg n} + \sqrt{n} \lg n \lg \lg n \right) = o(n) \text{ bits.} \tag{7}$$

In conclusion, storing our bitvector b plus the succinct data structure needed to answer *rank & select* queries takes $n + o(n)$ bits. Albeit the elegance of the approach and the above theoretical guarantee, even for large bitvectors such as $n = 2^{30}$ bits the extra $o(n)$ space is *not so negligible*, since terms in formula 7 will sum up to 6.67% + 60% + 0.18% = 66.85% of n [14]. This is due to the *high constants* hidden in the asymptotic notation. This is a crucial consideration concerning the practicality of such data structures. We will stress this consideration later on in Chapter 5.

Rank query evaluation

Now we show the algorithmic steps performed to answer a *rank* query in constant time, worst case. Assume the structure in Figure 3 and suppose we want to know $rank(i)$, for some $i \in [n]$. Then we will naturally resort on the three shown components (I_0, I_1 and T) as follows.

1. We determine the number of set bits, say c_1, up to the *superblock* where i resides: $c_1 = I_1[\lceil i/B_1 \rceil]$.
2. Then we determine the number of set bits, say c_2, up to the *basic block* where i resides: $c_2 = I_2[\lceil i/B_0 \rceil]$.
3. Finally we determine the number of set bits within the basic block where i is located, by looking up the proper counter, say c_3, in table T. Row index is just the basic block bitmap, while column index is the position of i *relative to that basic block*.

In conclusion, our query is evaluated as $T_{rank(i)} = c_1 + c_2 + c_3$ and it is obviously constant time worst case, $O(1)$.

3.1.2 Select

Selecting is *more complicated* than ranking. This is intuitively justified by the observation that we do not know which entry is needed. In processing $rank(i)$ we *know exactly* that i is located in $\lceil i/B_0 \rceil$ $\forall i \in [n]$, while in $select(i)$ the position of the i-th bit could potentially be in *any* basic block and we resort on *searching*.

Rank-based and position-based selection

As noticed for *rank*, also for *select* structures we can identify some common design choices. Two techniques are usually adopted: *rank-based* and *position-based* selection. Both these two approaches share the common step of (similarly to that of *rank*) choosing the size of the basic block along with an efficient way of selecting inside it.

For a *rank-based* selection, we re-use the previously described index for ranking. It actually need not to be the same for selecting as, for example, we could possibly change the size of a basic block. Assume, however, it is the same. We stress that the *structure is the same*, but its use is different: now, we have to *search* for the position of the i-th bit set. *Binary searching* is usually employed since first and second layer counters are trivially ordered. Therefore, by just applying three times binary search, we originate a simple algorithm for rank-based selection in $O(\lg n)$ steps. Notice that

we do not need table T. However if we are designing a succinct data structure able to answer *both rank* and *select* queries, we use the same index structure. Reasoning in this way, *select* uses the same extra space as *rank*, i.e., $o(n)$ bits.

Differently, a *position-based* selection makes use of a pre-computed array, say S, of *select* answers. This array is used to identify a position that is close (hopefully as close as possible) to the i-th set bit's position. We store the result of *select* queries for multiples of k **1**s. k can be chosen as we want, keeping in mind that the larger it is the less space S occupies but (possibly) the further the i-th set bit will be from the read value; conversely, the smaller k is the bigger S is be but we are likely to obtain a closer position to the i-th set bit. Formally, we first identify the largest j such that $jk \leq i$ by looking up in array S, then we identify the basic block where the i-th one lies in (with the help of an auxiliary index or not) and perform a selection on it (linear[4] or binary search). Assuming the best algorithm for searching, this is again $O(\lg n)$ time. The following picture shows this idea.

4 Since each basic block is assumed to be *small* in size, even a *linear scan* can be effective in practice.

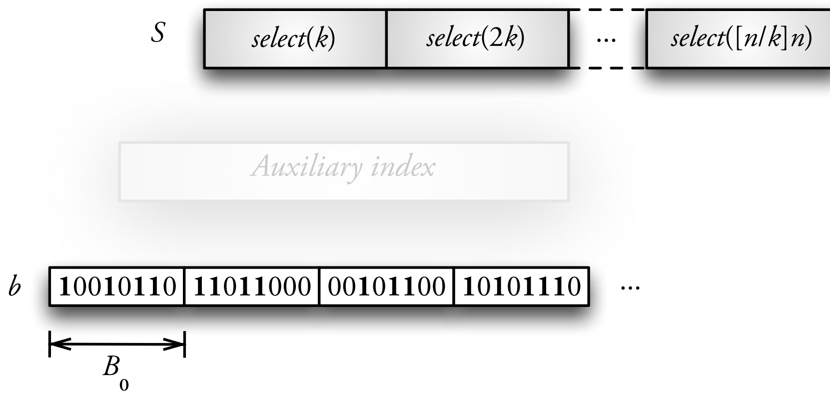

Figure 4: Position-based selection structure. Parameter k trades-off between space and time performance. B_0 is the size of a basic block and can be different from the B_0 of Figure 3. The auxiliary index appears like a ghost since it is optional.

Without taking into account for the optional auxiliary index, we have $O(n/k)$ entries in S each consisting of $O(\lg n)$ bits. This reveals in a total space of

$$O\left(\frac{n \lg n}{k}\right) \text{ which theoretically is } \textit{not} \text{ a } o(n).$$

There exists a constant-time solution for *select* queries which is *much more involved* with respect to that of *rank*. This solution was proposed by David Clark in his Ph.D. Thesis [5] and uses a three-level directory tree for a total of

$$\frac{3n}{\lceil \lg \lg n \rceil} + O(\sqrt{n} \lg n \lg \lg n) = o(n) \text{ extra bits.}$$

However, as already noticed by authors of [14] for Guy Jacobson constant-time solution for *rank*, also Clark's solution for *select* takes nearly 60% of extra space for $n = 2^{30}$ bits (not negligible).

3.2 IMPLEMENTATION

We now provide some useful guidelines [30, 14, 33] for the implementation of *rank & select* succinct data structures. There are three considerations, arising from *computer architecture*, that deeply influence the performance of such structures.

- *Cache misses* When dealing with *large* bitvectors that cannot fit inside cache memories, the overall performance is strongly influenced by *cache misses* (also called *cache faults*). Given the *very fine grain* of the operations performed by *rank & select*, even a single cache miss more can disrupt all programming optimization efforts. We recall from Section 2.7, that fetching a cache line from memory costs approximately 100 ns and this is enough time to perform hundreds of arithmetic and logic operations (less than $1/4$ ns $\times$ operation). Therefore, optimizing our code to be much faster than this fetch time will produce almost *no* gain in speed.
- *Parallelism* Given the fast spread of parallel architectures, we can *process more data in parallel* exploiting hardware registers and instructions.
- *Optimization order* We should optimize cache misses first, then branches, and finally arithmetic and logic operations. This is suggesting us the we should *avoid branching* whenever possible and code these structures to be *cache-aligned* (we fetch exactly one cache line for each query) and *64-bits aligned* (a query manipulates just one word).

With this tentative guideline in mind, we illustrate (and adopt) two ideas that play a key role in state-of-the-art implementations.

3.2.1 POPULATION COUNTING

We have already encountered a *population counting* technique when we build the 2-level structure for ranking: we have populated table T with counters keeping track of the number of bit set for all possible basic block configurations. The term *population counting* (in short, *popcount*) refers to counting how many bit set we have in a bitvector. Newer Intel® processors[5] (Nehalem and later architectures) make available a `popcnt` hardware instruction [17] that does the job for us.

Clearly, we can use `popcnt` instructions in order to count efficiently the number of set bits in a basic block. This makes a *great reduction in space, since we do not need to store table T any more*. Moreover, notice that, since 64 B = 512 bits is (usually) the size of a cache line, popcounting 512 bits for well cache-aligned bitvectors leads to exactly 1 cache miss. In fact, previous studies on *rank & select* data structures have demonstrated that their performance is inversely proportional to the basic block size. This is quite intuitive. The larger a basic block is, the less superblocks we have and we save space. But excessively enlarging this block size makes operations waste more memory accesses that are the most time-consuming

5 `http://www.intel.eu/content/www/eu/en/homepage.html`

sources of performance degradation. Therefore, choosing a basic block size of 512 bits is the way to go [30, 33].

3.2.2 Broadword programming

The term *broadword* has been introduced by Knuth in his masterpiece *The Art of Computer Programming*, Volume 4, in the fascicle about bitwise manipulation and tricks [20]. The underlying idea of broadword programming[6] is that we can use broad hardware registers (theoretically even larger than 64 bits) as *small parallel computers* able to process several pieces of information at a time. As noted by Vigna [30], broadword programming techniques are a promising source of speed-up in modern 64-bit architectures. The main advantage is that we gain more speed as the width of words increases with absolutely minimal effort.

6 Also known as SWAR (SIMD Within A Register).

Careful bitwise manipulation can be also used to count the number of set bits in a word (*sideways addition* algorithm) and avoid tests and branching (sequential code disruption). Moreover, wisely storing more information together in an *interleaved fashion*, whenever possible, makes us reduce cache misses.

We argue that a joint use of `popcnt` instructions, broadword programming techniques and interleaving can offer the *best* state-of-the-art implementation of *rank & select* data structures. By using some of these tools we can implement the structures described in [30, 14, 33].

Of particular interest for our purposes is the Java library Sux4J[7] written by Sebastiano Vigna. This library offers practical implementation of *rank & select* succinct data structures and uses the aforementioned tools. The latest release of Sux4J makes use of broadword programming + interleaving + `popcnt` instruction[8]. All implemented algorithms contain no tests, no branching and no precomputed tables are involved. We briefly summarized here the performance of his structures. We point the reader to [30] for a full description.

7 `http://sux.di.unimi.it`

8 In Java the population count instruction is the built-in static method `bitCount()` of `Integer` and `Long` classes: `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html`.

`Rank9` is the basic structure providing ranking. Its structure is a classical 2-level index as previously described. It guarantees practical constant time performance with at most 2 cache misses for each *rank* query and uses 25% additional space.

`Select9` is the other basic structure providing selection. It naturally exploits `Rank9` selection capabilities (rank-based selection) and uses 25%-37.5% additional space (25% for a backing instance of `Rank9`). It guarantees practical constant time evaluation with at most 4 cache misses per query.

`SimpleSelect` is another structure providing selection. It is a position-based solution which does not depend on `Rank9`. It uses a 2-level index plus a *spill list* (recording the exact position of each bit individually if the second-level counter does not suffice) and a broadword bit search algorithm. This solution uses more or less 13.75% extra space on evenly distributed bitvectors, providing very fast selections. This simple *select* data structure reveals to be very useful for its low-space occupancy when stor-

ing monotone sequences of integers in a succinct way, as we are going to describe in next chapter.

We have used some of the primitives implemented in Sux4J as black boxes in order to develop our own structures and exploit some bitwise tricks, along with interleaving, in our algorithms. Some meaningful examples will be presented in Chapter 5.

4

Elias-Fano integer encoding

Suppose we have a sequence $s = \langle s_0, \ldots, s_{n-1} \rangle$ consisting of n positive integers, possibly repeated, upper bounded by some value u, i.e., $s_i \in [0, u]$, $\forall i \in [n]$. u may be finite or not, known or unknown. Our goal is to represent this sequence as a bitvector in which each original integer is *self-delimited*, using as few as possible bits. This is the very basilar problem of *integer encoding*. We say that a compressed integer has a self-delimited representation if we know *exactly how many bits to read* in the bitvector resulting from the compression process for its retrieval.

Integer encoding

Applications are various, ranging from the storage and retrieval of large textual collections in (web) search-engines *inverted indexes* to *data compression*. We point the reader to [25] for a nice overview of most useful and important integer codes.

We can relax the constraint of positive integers, simply observing that each positive integer x can be mapped into $y = 2x$ and each negative one into $y = -2x + 1$. In this way we obtain a sequence of the above kind and by looking at the *parity* of the elements we can understand which inversion formula to apply. Moreover, once we have a sequence of positive integers, we can focus on the case of *strictly increasing integers*, since from s we can obtain the equivalent sequence s' of *prefix sums* where $s'_i = 1 + \sum_{k=0}^{i} s_k$, $\forall i \in [n]$. The operation is reversible by just "taking gaps". In this case it guaranteed that $n \leq u$. Formally, we provide this preliminary definition, useful in what follows.

Definition 4.0.1. Given a sequence $s = \langle s_0, \ldots, s_{n-1} \rangle$ of integers, we say that s is a *monotone sequence of increasing integers* if and only if $s_{i-1} \leq s_i$, $i = 1, \ldots, n-1$. We say it is *strictly increasing* if $<$ holds, instead of $\leq$.

4.1 Elias-Fano scheme

Let us fix a couple of conventions. Given an integer x, let $B(x)$ be its *binary* representation and $U(x)$ its *unary* representation, i.e., a binary number made up of[1] x 1s and a final 0. So $U(x)$ takes $x+1$ bits, $\forall x$. Example: $U(13) = 11111111111110$.

1 Swapping ones with zeros is possible, of course.

Now we illustrate an elegant scheme that works for monotone sequences of increasing integers. This encoding was independently proposed by Peter Elias [10, 11] and Robert Mario Fano [12].

4.1.1 Encoding

What Elias-Fano encoding consists in can be summarised by the following theorem.

Theorem 4.1.1. Given a monotone sequence s of n increasing integers, there exists an encoding for s that needs

$$\mathscr{C} = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n \text{ bits,} \tag{8}$$

where u is the maximum integer of the sequence, i.e., $u = s_n$.

Proof. We represent each integer of s in binary using $\lceil \lg u \rceil$ bits. Then we split each binary representation in two parts: a *higher* part consisting in the first[2] $\lceil \lg n \rceil$ bits that we call *higher bits* and a *lower* part consisting in the other $\ell = \lceil \lg u \rceil - \lceil \lg n \rceil = \lceil \lg(u/n) \rceil$ bits that we similarly call *lower bits*. Let us call h_i and ℓ_i the values of higher and lower bits of s_i respectively, $\forall i \in [n]$. The concatenation $L = \ell_0 \cdots \ell_{n-1}$ is stored explicitly and trivially takes $n\ell$ bits. Concerning the higher bits, we represent them in unary using a bitvector of $n + u/2^\ell = 2n$ bits as follows. We start from an empty, i.e. where there are no bit set, bitvector H and we set the bit in position $h_i + i$, $\forall i \in [n]$. Finally the Elias-Fano representation of s is given by the concatenation[3] $H \cdot L$, of H and L. ■

2 Start counting from the left.

3 Let $\cdot$ be the operator concatenating two bitvectors.

Now, in order to understand if this encoding is performing well or not, we would like to compare its space with the information-theoretical lower bound. In Section 2.5, we have seen that a bitvector $b(n,u)$ requires, at least, $n\lg(u/n) + O(n)$ bits. This formula is very similar to the space of Elias-Fano encoding, except from the fact that u and n, in $b(n,u)$, stand for the number of 1s and length of the bitvector respectively. So how do we relate these two formulas together? Suppose we have a very sparse bitvector, i.e. a bitvector in which the ones are small in number compared to the number of zeros. A, trivial, compact representation of such bitvectors can be obtained keeping track of the positions of the ones[4]. Reasoning in this way, we give birth to exactly a monotone sequence of length n (total number of 1s) of increasing integers where the maximum element is upper bounded by the length of the bitvector, which is u.

4 If ones are dominant in number, we reason the other way round and we store the positions of 0s.

We observe something more. For example, the following theorem.

Theorem 4.1.2. Indicating with $\mathscr{C}$ the space required by Elias-Fano encoding of a bitvector $b(n,u)$ of 0-th order entropy H_0, then

$$uH_0 \lesssim \mathscr{C} \leq u(H_0 + 2).$$

Proof. From Section 2.6 we know that $uH_0(b(n,u)) = n\lg(u/n) + (u-n)\lg(u/(u-n))$. We derive

$$\mathscr{C} = uH_0 + 2n + (u-n)\lg\frac{u-n}{u} \text{ bits, } u \geq n. \tag{9}$$

Let us now divide $\mathscr{C}$ by u: $\mathscr{C}/u$ represents how many bits we dedicate to the representation of each bit in $b(n,u)$. Since $u \geq u$, we distinguish two cases. The first is when $u = n$. Imposing this condition in equation 9 we immediately get[5] the upper bound $\mathscr{C}/u \leq H_0 + 2$. The other case is when $u \gg n$. In this case $2n/u \approx 0$ and $(u-n)/u \approx 1$, from which we obtain the lower bound $\mathscr{C}/u \gtrsim H_0$. ■

5 $\lg 0 = -\infty$ but this naturally rounded up to 0 in computer science.

In particular this theorem is telling us that the larger is u the closer the space of Elias-Fano is to the 0-th order entropy of the sparse bitvector it is compressing. In this way, it is possible to make $\mathscr{C}$ *arbitrary close* to uH_0. Furthermore, we claim that in practical situations the condition $u = n$ is almost never met, while condition $u \gg n$ is *very likely* to happen.

In conclusion the Elias-Fano encoding is, a *powerful tool to compress (sparse) bitvectors* and achieve compression performance very close to the 0-th order entropy of such bitvectors.

The pseudo code implementing the compression routine of Elias-Fano is show below. Its complexity is clearly $\Theta(n)$.

```
1:  procedure compress(s)
2:      n = s.length;
3:      u = s_{n-1};
4:      lowerBitsList = {};
5:      higherBits = [2n];
6:      for i = 0,...,2n-1 do
7:          higherBits[i] = 0;
8:      for i = 0,...,n-1 do
9:          add ℓ_i to lowerBitsList;
10:         higherBits[h_i + i] = 1;
```

Algorithm 1: Compression algorithm applied to a monotone sequence s of increasing integers.

4.1.1.1 An example

Let us show a concrete, yet toy, example. Let us suppose to have $s = \langle 3,4,7,13,14,15,21,43 \rangle$. Then $n = 8$ and $u = 43$. So the first step is to represent every integer using $\lceil \lg u \rceil = 6$ bits. Since $\lceil \lg n \rceil = 3$, both higher and lower part count 3 bits. We immediately derive that[6] $L =$ 011 100 111 101 110 111 101 011. Building up H can be done with the algorithm implicitly contained in the proof of previous theorem, i.e., starting from a bitvector of 16 zeros, we set the bit in position $h_i + i$, $i = 0,\ldots,7$. In Figure 5 we also use another method. Running the algorithm, we obtain $H = 1110111010001000$. Finally, the Elias-Fano representation of this sequence is $H \cdot L = 111011101000100001110011110111011110101 1$.

6 Space inserted for ease of visualization.

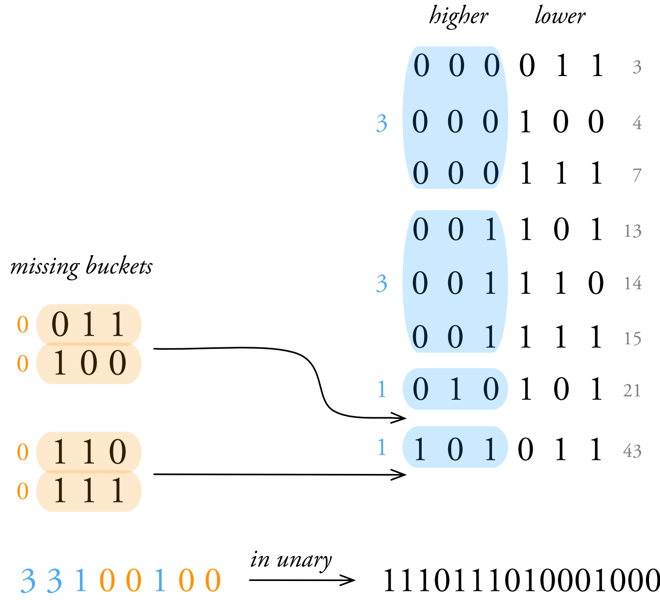

Figure 5: Elias-Fano encoding example. We can obtain H in the following, equivalent, way. We group in different bucket the same consecutive higher bit values. We also take into account for the missing buckets. Then we just count the cardinality (light blue numbers) of *each* bucket, including the missing ones too. Then just write in unary code these cardinalities.

4.1.2 Succinct representation of sequences

We have seen how to compress s, but not how we can access its elements, once compressed. The most important property of Elias-Fano encoding is that it supports the retrieval of any element of s *without the need of decompressing the entire sequence*, differently from several compression strategies. With careful design we can implement this operation in constant time. Given an integer $i \in [n]$, we have, basically, to re-link together h_i with ℓ_i, previously split-up. Let $\ell = \lceil \lg(u/n) \rceil$ be the number of lower bits. Then the retrieval of ℓ_i is trivial, we just need to read $L[i\ell, (i+1)\ell)$ bits. More interesting is the retrieval of the higher part h_i. Since we code the cardinality of buckets in unary, we have a zero whenever we change bucket. So we would like to know how many zeros are present in $H[0, select_H(1,i))$: this is $rank_H(0, select_H(1,i)) = select_H(1,i) - i$ for the relations we have seen at the beginning of the previous chapter. This quantity, read in binary, gives us the information we are searching for, i.e., h_i. Notice that Elias-Fano encoding is, therefore, self-delimiting. Summing up[7]

$$access_s(i) = h_i \cdot \ell_i = B(select(i) - i) \cdot L[i\ell, (i+1)\ell), \ \forall i \in [n].$$

7 Selections occurs on H, therefore we can just drop the subscript letter.

In this case $h_i \cdot \ell_i$ is naturally interpreted as a *bitwise addition* of programming languages, i.e., $access_s(i) = h_i << \ell_i | \ell_i$.

Therefore, we can use a succinct *rank & select* data structure to support *select* queries (we do not actually need *rank*) on bitvector H. This gives birth to an *Elias-Fano succinct data structure* for storing a monotone

Static Elias-Fano succinct data structure

sequence s of increasing integers and support fast access to its elements. The space needed is, in conclusion

$$EF(s[0,n)) = n\left\lceil \lg \frac{u}{n} \right\rceil + 2n + o(n) \text{ bits,} \tag{10}$$

where the redundancy, extra, factor $o(n)$ accounts for the ad-hoc selection structure that we call `selector`.

Then we can write the following pseudo code.

```
1: procedure access(i, selector, ℓ)
2:     return (selector.select(i) − i) ·L[iℓ, (i + 1)ℓ);
```

Algorithm 2: Random accessing.

The selection structure we use is `SimpleSelect` in our implementation. Clearly, if applications demand it, we can use `Select9` for even better selection performance, but tolerating a greater redundancy factor (37.5 %).

In the following of this dissertation we will refer to an Elias-Fano succinct data structure, as previously described, as a *static Elias-Fano data structure*. Its space occupancy is $EF(s[0,n))$ bits of formula 10 and we can access any compressed integer in practical constant time.

Static Elias-Fano data structure

Inspecting the above pseudo-code, we can immediately derive the time complexity of a get_s operation on a *static* Elias-Fano representation of s. For sufficiently large sequences, i.e., the ones that do not fit completely in cache, the time is dominated by one cache miss (cf in the following) to access the right portion of array L plus the time for the $select_s$ primitive. This results in

$$T^*_{get_s} = 1\,\text{cf} + T_{select_s}, \tag{11}$$

where $1\,\text{cf} \approx 100$ ns according to Section 2.7. We will use this formula for subsequent analysis.

We use the term *static* since we silently assume to already have s. But what if we do not have s in our hands?

Read next chapter.

Part II

Elias-Fano Structures

5 Append-only

This chapter deals with the problem at the heart of this dissertation. This is formalized as follows.

Problem. Starting from an empty monotone sequence s, append non-decreasing integers to s as to compress them and support fast $append_s$ and get_s operations.

We propose two succinct data structures that make use of the previously introduced Elias-Fano integer encoding. We have seen that Elias-Fano strategy is able to compress s achieving very close compression performance to the 0-th order entropy of the underlying bitvector[1]. This is possible since we have full knowledge of both n and u, namely the length of the sequence and the maximum sequence element respectively. We refer to such scenario as a *static* one, i.e., in which we *already know* s. But what happens, if, instead, we do not know n nor u? We develop a new strategy that makes possible to apply Elias-Fano integer encoding, even when we do not know any information a priori. Conversely to the static case, we build s incrementally, in an *append-only* fashion. Because of the lack of a priori information, we are going loose something in both space and query time with respect to the static case. However, such dynamic scenario occurs in a lot of practical situations, where we would like to apply compression *on the fly* (and not only at the end of the building process of s) while adding new integers to our sequence.

1 Section 4.1.1.

Static scenario

Dynamic scenario

Despite of such additional complexity, fundamental to this dissertation will be the proof that we are only loosing a *negligible factor* in space while introducing a *minimal* time degradation with respect to the static case.

5.1 Known sequence length

Given an empty sequence s, suppose we know its future length n but we *ignore* which integers will be appended to it. In particular, we do not know the maximum element of the sequence, u. The strategy we adopt is described in the following.

5.1.1 Algorithmic description

We maintain a buffer of to-be-compressed integers, of fixed length B. When we need to append a new integer to the sequence, we check if it is greater than or equal to the last inserted. If it is so, the integer is added to the buffer, otherwise we report an error (exception) to the user. The buffer

keeps growing until the maximum capacity is reached. At this point, we compress the B added integers using the Elias-Fano encoding and keeping three *extra information*: the number of lower bits, say ℓ_0, we need to read to randomly access an integer in $s[0, B)$; the maximum integer among the B compressed, say u_0, and a pointer to the selector's structure $selector_0$, supporting the *select* primitive on the bitvector resulting from the compression. Now we can empty the buffer and *repeat* the steps just described, with only one small difference: since we append increasing integers, we will compress the *difference* between current integers and previous bucket upper bound. This is fundamental[2]. As an example, we compress integer x belonging to k-th bucket as $x - u_{k-1}$.

2 See Property 5.1.1.

What we are implementing is a *bucketing strategy*, compressing each bucket (the buffer of B integers) when it is full.

Imagine we have added as many integers as to have $b_0, b_1, \dots, b_k$ formed buckets and we are currently forming b_{k+1}, then the following picture describes what our data structure looks like.

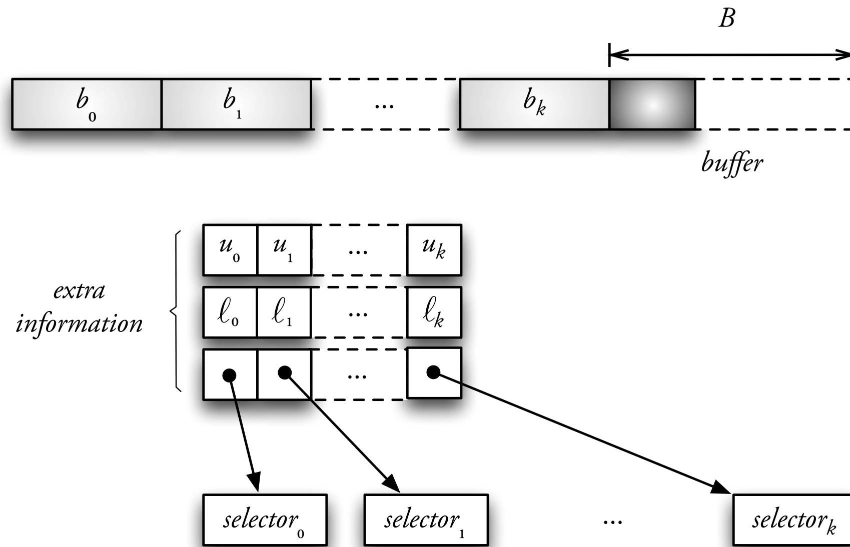

Figure 6: Append-only Elias-Fano data structure. The darker blended part represents the portion of uncompressed integers, while the light blended parts the already formed buckets. As we will explain next, in our Java implementation, $\{u_i\}_i, \{\ell_i\}_i$ and the reference to each selector's structure have been stored in proper resizing-arrays. Array-resizing implementation is the classical one described in [28], pages 136-141.

Notice that $b_0, b_1, \dots, b_k$ plus their corresponding extra information are static Elias-Fano succinct data structures, as described in Chapter 4.

5.1.2 OPERATIONS

We need to implement three operations on our data structure.

- The basic $append_s(x)$ and $get_s(i)$, as defined in Subsection 2.3.1;
- $nextGEQ_s(x)$ which returns the smallest integer of the sequence which is greater than or equal to x or just -1 if such value does not exist.

The pseudo code for the operations is shown below. We have omitted, for ease of presentation, the tests for correctness of the input parameters that can throw proper exceptions. They are present in the actual Java implementation.

```
1: procedure append(x)
2:     if buffer.isFull() then
3:         compress(buffer);
4:         buffer.clear();
5:     buffer.add(x);
6:     length++;
```

Algorithm 3: *compress* is the compression algorithm we have explained in Section 4.1.1. In this case, it is applied to compress the B integers stored in `buffer`. Variable `length` indicates the number of stored integers.

```
1: procedure get(i)
2:     b_j = i's bucket;
3:     offset = i % B;
4:     if s_i ∈ buffer then
5:         return buffer.get(offset);
6:     return b_j.access(offset, selector_j, ℓ_j) + u_{j−1};
```

Algorithm 4: *access* is the algorithm to access the i-th integer of an Elias-Fano succinct data structure we have explained in Section 4.1.2.

```
1: procedure nextGEQ(x)
2:     bucket = x ≤ 0 ? 0 : binarySearchOverU(x, 0, u.size());
3:     it = new iterator starting from bucket;
4:     while it has elements do
5:         v = it's next element;
6:         if v ≥ x then
7:             return v;
8:     return −1;
```

Algorithm 5: A 2-step *nextGEQ* pseudo code. This operation could be trivially implemented by iterating over the integers in the sequence and return the first integer $v \geq x$. But clearly this search can be speeded-up by first binary searching over the buckets' upper bound values and then iterating over a single bucket's integers.

```
1:  procedure binarySearchOverU(x, i, j)
2:      bucket = (i + j)/2;
3:      u_1 = u[bucket];
4:      if x = u_1 then
5:          return bucket − 1;
6:      u_2 = bucket < buckets ? u[bucket + 1] : last;
7:      if u_1 < x < u_2 then
8:          return bucket;
9:      if x ≥ u_2 and u_2 ≠ last then
10:         return binarySearchOverU(x, bucket, j);
11:     if x ≥ u_2 and u_2 = last then
12:         return buckets;
13:     return binarySearchOverU(x, i, bucket);
```

Algorithm 6: Suppose that u is the array storing buckets' upper bounds and buckets is the number of total formed buckets. The binary search over u identifies a bucket bucket such that: u[bucket] $\leq x <$ u[bucket + 1].

5.1.3 Space complexity

In this section we develop an expression for the space occupancy of the implemented, append-only Elias-Fano data structure. We will use the word-RAM model introduced in Section 2.7 to write such expression, recalling that $w = \lg u$ bits represents a memory word.

Suppose we would like to encode our monotone sequence s of n non-decreasing integers with Elias-Fano as we did in Chapter 4. Let u be the maximum integer of the sequence (last element). We already show that the sequence takes[3]

$$S^*(n,u) = EF(s[0,n)) = n\lg\frac{u}{n} + 2n + o(n) \text{ bits} \quad (12)$$

3 Throughout the chapter, we drop the ceil notation $\lceil\cdot\rceil$ when we refer to the result of a division that the context imposes to be an integer quantity if it is not already. In fact, all these quantities have been rounded up to the next closest integer value in our Java implementation.

Now, the append-only version takes, at most, for each bucket b_i:

$$B\lg\frac{u_i}{B} + 2B + o(B) + 8\lg u \text{ bits,}$$

where we recall that $w = \lg u$ represents the word-size of the RAM model. The 8 lost words per bucket are due to:

- the additional information we need to store for each bucket, i.e., u_i, ℓ_i and a reference to $selector_i$ (3 words);
- padding bits for selectors' structures (4 words) and array of lower bits (1 word).

Notice that for the static version, $o(n)$ already includes all needed padding bits, since they are negligible.

Summing on all buckets, the *total space of append-only* is

$$S(n,u,B) = B\sum_{i=0}^{n/B-1}\lg\frac{u_i}{B} + \frac{n}{B}\cdot 2B + \frac{n}{B}\cdot o(B) + B\lg u + 8\lg u\cdot\frac{n}{B} \text{ bits,}$$

which simplifies to

$$S(n,u,B) = B\sum_{i=0}^{n/B-1}\lg\frac{u_i}{B} + 2n + o(n) + B\lg u + \frac{8n\lg u}{B} \text{ bits.} \quad (13)$$

The additional cost of $B\lg u$ bits is due to the buffer of uncompressed integers. Now, comparing formula 12 with 13, we start getting an idea of what may be the extra cost of the append-only version with respect to the static one. In the following we show how to sharpen this idea: we need to manipulate the summation in formula 13, which depends on sequence s.

We can proceed in two ways. The first one is trying to estimate the average u_i, $i = 0, 1, \ldots, n/B-1$. The second one is, as similarly done in [26], resorting on the following

Property 5.1.1. Given a monotone sequence s of integers, we can show that:

$$EF(s[0,k)) + EF(s[k,n)) \leq EF(s[0,n)),\ \forall\ 0 \leq k < n.$$

This property can be easily generalized to the case of an arbitrary number of splitting. Let us proceed in order, considering both of them.

For what concerns the first one, remember that u/n represents the average gap between two consecutive integers of s. So considering the we encode, for each bucket, the difference between each integer with the maximum integer of the previous bucket, we need to evaluate the entity of the average gap between two consecutive maxima. But this average gap is soon evaluated as $u_i = uB/n$, $i = 0, 1, \ldots, n/B - 1$. This is an *approximation*, done in order to let the formula be independent of the sequence's integers. The approximation lies in the fact that we are considering the average u_i, which can be different in reality. We will see, however, that this model is anyway *very accurate* by a large numbers of experiments in Chapter 7.

Substituting the found value for u_i in formula 13, we finally get

$$S(n,u,B) = n\lg\frac{u}{n} + 2n + o(n) + B\lg u + \frac{8n\lg u}{B} \text{ bits.} \quad (14)$$

We derive that the *upper bound $\mathcal{E}(n,u,B)$ on the extra space* is

Extra space function

$$\mathcal{E}(n,u,B) = B\lg u + \frac{8n\lg u}{B} \text{ bits.} \quad (15)$$

The upper bound on the space occupancy of append-only Elias-Fano encoding, can therefore be written as

Total space occupancy of append-only Elias-Fano data structure

$$S(n,u,B) = S^*(n,u) + \mathcal{E}(n,u,B) \text{ bits.} \quad (16)$$

The above formula is of *outermost importance*: it is telling us that the space occupancy of append-only Elias-Fano encoding of an integer sequence s is the one of the static Elias-Fano version (independent of the bucket size), plus an extra contribution, which depends on the bucket size B.

For what concerns the second way, we start demonstrating Property 5.1.1.

Proof. We split s in two subsequences, $s[0,k)$ and $s[k,n)$, and we encode them with Elias-Fano: the second subsequence compress the difference between its integers and s_{k-1}. In this way we have split the universe u as $u = u_1 + u_2$, where $u_1 = s_{k-1}$ and $u_2 = s_{n-1}$. For ease of notation, let $n_1 = k$ and $n_2 = n - k$. Thus $n = n_1 + n_2$.

Then we have

$$EF(s[0,k)) = S^*(n_1,u_1) = n_1 \lg \frac{u_1}{n_1} + 2n_1 + o(n_1), \text{ bits}$$
$$EF(s[k,n)) = S^*(n_2,u_2) = n_2 \lg \frac{u_2}{n_2} + 2n_2 + o(n_2), \text{ bits.}$$

Summing up the two spaces we obtain

$$n_1 \lg \frac{u_1}{n_1} + n_2 \lg \frac{u_2}{n_2} + 2n + o(n).$$

Now, comparing this result with the total Elias-Fano encoding space of s, which is

$$n_1 \lg \frac{u_1+u_2}{n_1+n_2} + n_2 \lg \frac{u_1+u_2}{n_1+n_2} + 2n + o(n),$$

the claim follows if only if

$$\frac{u_1}{n_1} \leq \frac{u_1+u_2}{n_1+n_2} \wedge \frac{u_2}{n_2} \leq \frac{u_1+u_2}{n_1+n_2}.$$

Recalling that $n_1 \leq u_1$ and $n_2 \leq u_2$, both inequalities simplify to $n_1 n_2 \leq u_1 u_2$ which is always true. ■

Therefore, we can rewrite formula 16 into

$$S(n,u,B) \leq S^*(n,u) + \mathcal{E}(n,u,B) \text{ bits.}$$

Calling $\Delta \geq 0$ the amount of bits that possibly separate $S(n,u,B)$ from $S^*(n,u) + \mathcal{E}(n,u,B)$, we can rewrite $S(n,u,B)$ as

$$S(n,u,B) = S^*(n,u) + \mathcal{E}(n,u,B) - \Delta \text{ bits.}$$

This formula is quite interesting because is telling us that if $\Delta \geq \mathcal{E}(n,u,B)$, i.e., what we gain in compression is even larger than the extra introduced overhead, our append-only strategy will actually prove to be *better* than a static one. This is due to the fact that append-only is compressing each bucket with a much smaller upper bound value with respect to u. This implies shorter compressed bitvectors, as we have seen in Chapter 4.

Now a fundamental question is posed: *does this condition hold in practical cases?* Since most of the contribution to $\mathcal{E}(n,u,B)$ comes from B and this is fixed[4], for very long sequences Δ is *very likely to grow* while B remains *relatively small*. In almost all experiments performed in Chapter 7, the introduced overhead is *not compensated* by the append-only compression gain. However, we will see in Chapter 8 some concrete example in which append-only can offer better compression with respect to a static encoding.

In conclusion, the satisfaction of $\Delta \geq \mathcal{E}(n,u,B)$ cannot be guaranteed nor predicted, since it depends on u_i, $i = 0,1,\ldots,n/B-1$, that are unknown at the time we are acquiring s.

4 Or, at least, grows slowly as we will see later on.

5.1.3.1 MINIMIZING EXTRA STORAGE

Now we are left with the task of choosing the best bucket size B, in order to minimize the additional space represented by formula 15.

From [5]

$$\frac{\partial \mathcal{E}(n,u,B)}{\partial B} = 0$$

we obtain

$$B^* = 2\sqrt{2n} \text{ integers.} \quad (17)$$

If we substitute this result in formula 15, we immediately get an upper bound on the extra space when choosing the best possible B. By doing so, we obtain the elegant result of

$$\mathcal{E}^* = \mathcal{E}(n,u,B^*) = 4\sqrt{2n}\lg u = 2B^* \lg u \text{ bits.} \quad (18)$$

This result is suggesting us, that the maximum extra space we can have, when choosing the best B, is equal to $2B^*$ words, i.e., *twice the buffer occupancy*. The space upper bound for the best choice of B, can be then rewritten as $S(n,u,B^*) = S^*(n,u) + \mathcal{E}^*$ bits.

Our ultimate dream would be that of proving that we are only adding $o(S^*(n,u))$ bits with respect to the static Elias-Fano encoding, thus achieving the so-called succinct bound for s. We can actually do better than this, surpassing the succinct bound and adding only $o(n)$ bits. In other words *we would like that the additional space remains negligible* for reasonably large values of n, u and optimal choice of B and ask for the minimum number of integers, say η, for which

Theorem 5.1.1. $\mathcal{E}^* = o(n)$ bits.

Proof. By definition 2.2.3, $\mathcal{E}^* = o(n)$ if and only if $\forall\, c > 0\ \exists\, \eta > 0$ such that $0 \leq \mathcal{E}^* < cn$, $\forall\, n \geq \eta$. Therefore by solving

$$2\sqrt{8n}\lg u < cn,$$

with respect to n we obtain $n > 32\lg^2 u / c^2$ so that we can conclude the claim holds $\forall\, n \geq \eta = 32\lg^2 u / c^2$. ■

Figures 7 and 8 clearly show what Theorem 5.1.1 states[6] for a certain $n \geq \eta$. Solving the following equation with respect to n

$$2\sqrt{8n}\lg u = n,$$

yields $n = 32\lg^2 u$ integers (this corresponds to choose $c = 1$ in Theorem 5.1.1). This value will give us an additional space of exactly n bits. Actually we require a different value for η and we need to go further with n to satisfy Theorem 5.1.1. The problem, here, is that we should fix a constant $c > 0$ in Theorem 5.1.1 and determine η consequently. We will see how

5 We must be rigorous. For fixed n and u, the formula only holds for integer choices of B. Therefore, in order to be able to differentiate it, we should consider its extension to the field to real numbers $\mathbb{R}$.

6 The plots have been generated for a fixed choice of $\lg u = 64$ bits: they are meaningful just for illustrative purposes.

to avoid this with another, more practical, approach in next section. In conclusion, the append-only data structure takes

$$S(n,u,B^*) = S^*(n,u) + o(n) \text{ bits}, \forall\, n = \omega(\lg^2 u). \tag{19}$$

This is the space complexity of static Elias-Fano *plus a small amount of memory*, i.e., $o(n)$ extra bits.

Notice that the previous bound is stronger than the succinct bound $S(n,u,B^*) = S^*(n,u) + o(S^*(n,u))$, therefore this is automatically satisfied. Formally, since we can lower bound $S^*(n,u)$ with

$$c(n\lg(u/n) + 2n) < c(n\lg(u/n) + 2n + o(n)),$$

as immediate from Theorem 5.1.1, we obtain that the succinct bound is satisfied $\forall\, n \geq \eta = 32\lg^2 u/(c^2(\lg(u/n)+2)^2)$. In conclusion we can say

$$S(n,u,B^*) = S^*(n,u) + o(S^*(n,u)) \text{ bits}, \forall\, n = \omega\left(\frac{\lg^2 u}{(\lg(u/n)+2)^2}\right).$$

Notice that since the quantity $(\lg(u/n)+2)^2$ is always greater than 1 (just recall that $u \geq n$), the succinct bound is *more easily* satisfied (for smaller values of n) with respect to the strict bound of Theorem 5.1.1. The reason we take into account the succinct bound too will be clear in next section.

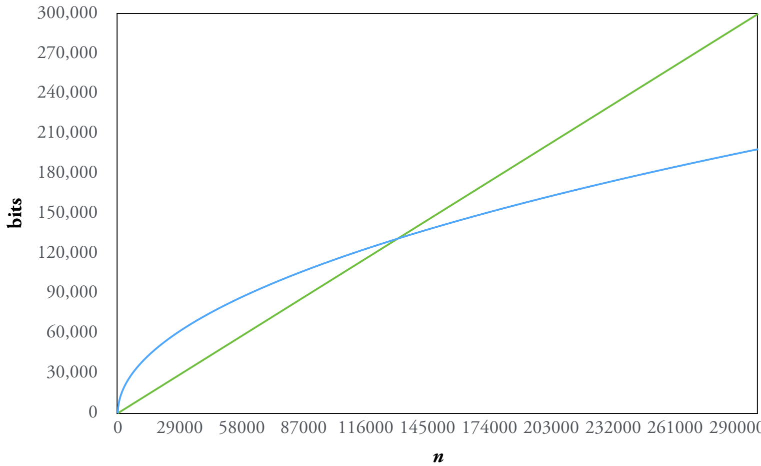

Figure 7: Curves intersecting. For $\lg u = 64$ bits, we obtain $n = 131072$ integers and an addition of exactly the same amount of bits.

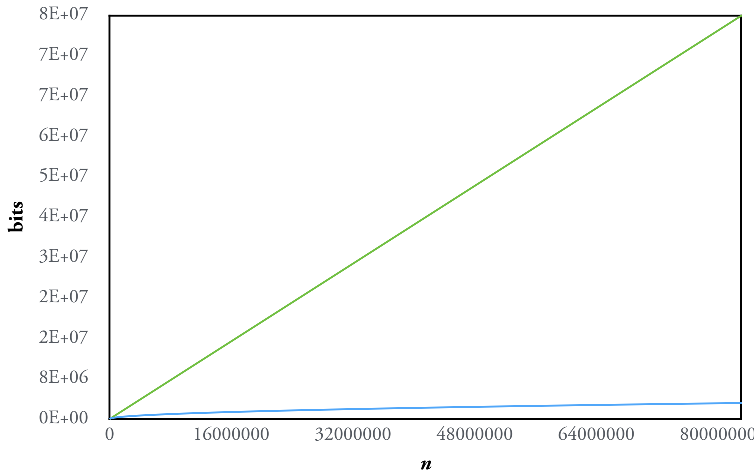

Figure 8: Asymptotic behaviour.

5.1.3.2 Practical tool

We would like to know for which amount n of stored integers, the additional space of an append-only Elias-Fano structure, with respect to a static Elias-Fano encoding, is below a certain threshold $0 < \varepsilon \leq 1$, fixed as small as we want. For this purpose, consider the following equation, where we ask for which n the extra memory is ε *times* of the Elias-Fano static space[7]

$$B \lg u + \frac{8n \lg u}{B} = \varepsilon\left(n\left\lceil \lg \frac{u}{n} \right\rceil + 2n\right). \tag{20}$$

7 We can neglect the $o(n)$ term in formula 12, since $n\lceil \lg(u/n) \rceil + 2n < S^*(n,u)$ and, therefore, equation 20 gives us an upper bound on such value of n.

For our purposes, the above formula is a *versatile and fundamental tool*. If we fix B, u and ε we can solve it with respect to n, i.e., finding the minimum n for which the extra space is below ε%. On the other hand, if we fix B, u and n and we solve the equation with respect to ε we immediately get an idea of how many bits we are adding to the static case. Notice that if we are using best B, then the formula reduces to $2B \lg u = \varepsilon(n \lg(u/n) + 2n)$. We mostly use it in the first way: we fix ε to be very small, e.g. $\varepsilon = 0.01$ (1% extra) or $\varepsilon = 0.02$ (2% extra) and, for a given B and u, we evaluate n.

For such values of ε, we can actually state that $n \geq \eta$ and the extra space used by our append-only data structure is *practically* $o(S^*(n,u))$. This is *the same* as theoretically fixing a constant $c > 0$ and calculating corresponding η in the bounds of previous section.

It should be clear, now, why we have introduced the succinct bound. Using the above formula and imposing the satisfaction of the strict bound in Theorem 5.1.1, i.e., $B \lg u + 8n \lg u / B = \varepsilon n$, will yield too much large values of n that, in conclusion, will be very *hardly satisfying* in practical situations. This fact is due to the potentially *high constants* hidden by the asymptotic notation (e.g., 32 in our case) and plays a *fundamental role* in the study and design of succinct data structures (as well as in many other situations). In such cases, as correctly pointed out by the authors

of [30, 14, 16], before the asymptotic advantage comes clearly into play the data structure is usually *too large*. Therefore, while it is theoretically and mathematically fair to claim an additional space of $o(n)$, it has *poor value* as a real-application, effective, data structure. This is a very common trend in the design of succinct data structures.

On the other hand, by resorting on the succinct bound $S^*(n,u) + o(S^*(n,u))$ we are able to embrace theoretical and practical worlds, achieving a good, practical, data structure.

5.1.4 Time complexity

In this section, we present the time complexity of the two fundamental operations of append-only Elias-Fano: `append()` and `get()` that we have previously introduced and explained.

5.1.4.1 Amortized analysis

When dealing with resizing-arrays, the usual strategy to provide a performance guarantee is to keep track of the total cost of all performed operations and then divide this total cost by the number of operations. What obtained is an *amortized cost* of the operations: in such cases we allow some expensive operations while keeping the *total cost* of operations low.

Amortized cost

Consider a resizing-array and suppose we want to insert n items in it. We assume, for simplicity, that n is a power of 2. Now, starting from an initial capacity of 2 elements (as in my implementation), we want to evaluate the total cost of performing n consecutive `append()` operations. The cost will be expressed in memory accesses, according to the fact that the RAM model is able to access any memory location in constant time, worst case.

The number of memory accesses for n consecutive insertions is easily evaluated as

$$n + 4 + 8 + 16 + \ldots + 2n = n + \sum_{i=2}^{\lg n+1} 2^i,$$

where the first term, n, accounts for each insertion performed in the array and the second term, the summation, is taking into account all memory accesses performed when the data structure doubles in size.

The geometric series is evaluated as[8]

$$\sum_{i=2}^{\lg n+1} 2^i = \sum_{i=0}^{\lg n+1} 2^i - 3 = 4n - 4.$$

In conclusion, we can say that n items are inserted paying $5n-4$ memory accesses. As a rule of thumb, we are paying 5 times the number of elements to insert. The number of accesses per insertion is constant and approximately equal to 5 in all meaningful cases. What we are doing is somehow spreading the cost of *very few expensive operations*[9], i.e., each

8 Recall that
$$\sum_{i=0}^{n-1} \alpha^i = \frac{1-\alpha^n}{1-\alpha}, \quad \forall \alpha \neq 1.$$

9 The number of operations that cause array-resizing is $\lg n - 1$.

time the array doubles its capacity and copies all elements, through a very large number of inexpensive operations.

Another way to look at this process is shown in Figure 9, as similarly done in [28].

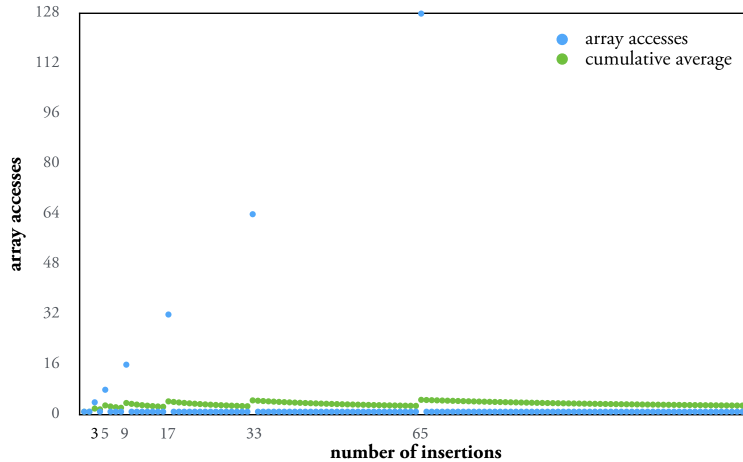

Figure 9: Amortized analysis. Green dots show how we keep the total cost of operations low.

5.1.4.2 Append

Using the result of the amortized analysis, we develop a formula describing a theoretical upper bound on the construction time of an append-only Elias-Fano integer sequence, i.e., how much time we spend in appending n monotonically increasing integers.

The time of n consecutive `append()` operations is made up of the following three contributions.

- n total insertions in `buffer`, which is *not* a resizing-array.
- $3n/B$ total insertions in the following resizing-arrays: `info`, `selectors` and `lowerBits` (storing each `lowerBitsList` of the compression algorithm in Section 4.1.1).
- n/B compression routines.

The cost of a compression routine is dominated by the loop we show in the following Java code fragment.[10]

```
long v;
for (int i = 0; i < B; i++)
{
  v = buffer[i] - previousUpperBound;
  lowerBitsList.set(i, v & lowerBitsMask);
  upperBits.set(( v >>> l ) + i);
}
```

It costs a total of $3B$ memory accesses.

10 In the illustrated snippet of code, `l` is the current number of lower bits. As immediate from the code, we are performing 3 memory accesses in each iteration of the loop.

By these considerations, we can write the total amortized cost of n consecutive `append()` operations as

$$T(n,B) = n + 5 \cdot \frac{3n}{B} + 3B \cdot \frac{n}{B} = 4n + \frac{15n}{B}.$$

By substituting the optimal value of $B^* = \sqrt{8n}$ in the above formula, we end up with

$$T(n,B^*) = 4n + 15\sqrt{\frac{n}{8}} \approx 4n, \ \forall n \text{ sufficiently large.} \qquad (21)$$

Comparing the result in formula 21 with the amortized cost of n resizing-array insertions, we get that the append-only Elias-Fano is *at least* $5/4 = 1.25$ times faster than a simple resizing-array.

5.1.4.3 De-amortization

We can use a classical de-amortization argument, similar to the one contained in [3], to let append operations be performed in constant worst case time as follows.

We maintain two buffers, say `buffer1` and `buffer2`, of B integers each instead of only one. We append incoming integers to `buffer1` until it gets full. At this point each successive *append* operation will perform two steps: we store the incoming integer in `buffer2` and perform an iteration of the `for` loop in the compression routine. When `buffer2` itself will become full, conversely `buffer1` will become empty so that we can swap the role of the two buffers and repeat the strategy. At any point in time (after the first B integers are appended to the structure), we have two buffers: one is full and is under compression; the other stores newly appended integers. In order to maintain the correctness of the data structure, the last appended integers (at most $2B-1$) will be accessed directly in the buffers. This guarantees a worst case running-time performance.

It should be clear that this de-amortization is perfectly possible but does not come for free: it automatically doubles the memory requirement[11] for the to-be-compressed integers (which is the dominant factor in formula 15) since it needs two arrays.

In conclusion, while it is *theoretically fair* to claim our data structure achieves constant time worst case running time for each *append* operation, we have not implemented this strategy in our Java codes for the reason given above.

5.1.4.4 Access

Looking at the pseudo codes from Section 4.1.2 and 5.1.2, we immediately see that we are paying, basically, the *identification* of the bucket where the requested integer lies plus an *access* operation. The additional bucket identification cost requires a constant number of operations (2 array accesses and a couple of bitwise manipulations in our implementation, as we are

11 It is possible to use only one buffer but we do not guarantee a constant running time. In fact, when the buffer becomes full, we compress an integer from it and store the next one overwriting the integer just compressed. The tricky point is that we need to initialize a `SimpleSelect` structure at each step of the `for` loop (which is already very costly) to maintain the correctness of the data structure. This initialization does not run in constant time, obviously.

going to see in Section 5.3) and this is clearly *constant time*. Therefore we should expect a *very close* level of performance to the access time of the equivalent static Elias-Fano data structure that, as seen in Chapter 4, requires practical constant time.

More precisely, now the *select* operation is *not* performed on the whole sequence s but only on a compressed portion consisting of B integers. Let indicate with T_{select_B} the time needed to perform a selection of this compressed portion of s. Then we can write

$$T_{get_s} = 3\,\mathrm{cf} + T_{select_B} = 2\,\mathrm{cf} + T^*_{get_s} + T_{select_B} - T_{select_s}. \tag{22}$$

As it is clear, now we spend two cache misses more with respect to the static version since we need to access additional information (u_i and ℓ_i for i-th bucket). However, since $B \ll |s|$ we have $T_{select_B} < T_{select_s}$ (their time difference is one cache miss on average, as we will see later on). In conclusion, while the append-only version of Elias-Fano introduces a time overhead due to the access to additional information, selection is performed faster and this automatically trades off with the equivalent get_s static time. We will validate this model with experiments in Chapter 7.

5.1.4.5 Next Greater Or Equal

From pseudo codes in 5.1.2, we derive that a $nextGEQ_s(x)$ operation costs $O\Big(\lg \frac{n}{B} + B\Big)$, which is $O(B) = O(\sqrt{8n})$ if we use the best possible bucket size. The first addend represents the cost of the binary search over buckets' upper bounds while the second one represents the cost of a linear scan through a bucket's integers.

5.2 Unknown sequence length

Now we take the theoretical results of the previous section, to implement our second data structure. Therefore, assume we can use our append-only Elias-Fano data structure as a black box for developing more advanced structures.

In most practical situations, we do not have any knowledge of both u and n. We simply do not know how much our sequence will grow and which elements will end up in it. This means, that we cannot choose the best possible bucket size, as we did in formula 17. We need to adopt an adaptive strategy in which B changes over time as n and u do. In the following we deeply analyse this strategy.

5.2.1 Algorithmic description

In order to dynamically adapt our data structure to the length of the sequence, we have to change B "every now and then". We would like to pick the best possible B for the current length of s, so that we can control extra space growth. For that reason, we start creating an append-only Elias-Fano data structure with an initial choice of B as we did in previous section. Let us call this initial value B_0. In equation 17, $B = 2\sqrt{2n}$, we can fix any of the two parameters, n or B, (namely, the one we know) and choose the other consequently. In this case we do not know n, so we fix B and compute best n, i.e., the number of integers for which that choice of B minimizes additional space. From $B = 2\sqrt{2n}$ we derive $n = B^2/8$.

So starting with B_0, we optimally compute n as $B_0^2/8$. This means that once we have added $B_0^2/8$ integers to s, B_0 will not be optimal any more and we have to change it. Suppose we change it using some strategy, that we are going to describe next, in a new B that we call B_1. Now simply reconstruct the sequence s with this new optimal bucket size B_1. Then compute optimal $n = B_1^2/8$ as before and keep adding new integers to s until we reach this new value of n. At that point, repeat the strategy we have just outlined, i.e., make a new choice for B and compute corresponding optimal n.

Suppose we have added as many integers to our sequence as to have changed B for k times. This means that we have reconstructed s for k times. *This process is necessary to minimize the needed extra memory.* If we do not do so and we create a new sequence to juxtapose the old one, this will reveal in a large memory waste for small sequences. The problem lies in the fact that from a certain value of n on, the *reconstruction process becomes too costly*, even if the more integers we add, the less frequent these reconstructions will be. This consideration implies that we will adopt an *hybrid strategy*, conceptually separated in two parts:

1. we reconstruct s with optimally chosen B values for k times, until we reach a threshold of added integers, say ϕ, for which we guar-

antee an upper bound on required extra memory and confirm that this is very small in practice;

2. when k reconstructions have been performed, we just keep on evaluating B and n as before, but we create a new append-only Elias-Fano data structure with such value of B, thus avoiding the potentially high cost of a reconstruction process.

In other words, ϕ represents the value of n for which we have a *reasonable trade-off* between reconstruction cost and extra space.

Therefore, after we have done k reconstructions, our new data structure, that we will call *adaptive append-only Elias-Fano*, is an *array of chunks* $c_0, c_1, \ldots, c_m$, each of these being an append-only structure. As similarly done for the append-only data structure, whenever we create a new chunk, all successive appended integers will be stored as the difference between their own values and maximum integer of previous chunk. This is necessary as we will see in Section 5.2.3. Notice that c_0 represents the first-created chunk, i.e., the append-only structure we have reconstructed for k times, while m represents the *total number of created chunks* except the first (surely created), which is, once again, unknown.

Adaptive append-only Elias-Fano data structure

If B_i is the chosen value of bucket size after we have changed it for i times, the structure is graphically represented in the following picture.

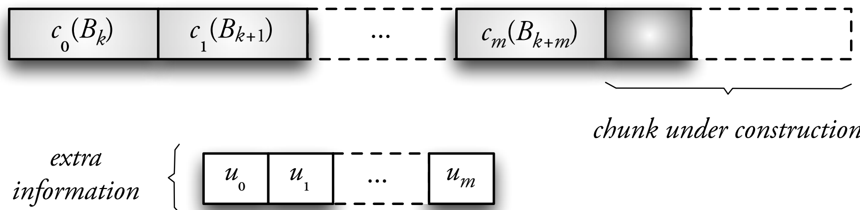

Figure 10: Adaptive append-only Elias-Fano data structure. We have specified each bucket size for each created chunk. u_i represents the maximum stored integer for the i-th chunk, for $i = 0, \ldots, m$, and it is stored inside its corresponding chunk. Light blended parts represent already formed chunks while the darker one the under-construction chunk.

We have not yet explained two fundamental points:

1. how we choose a new value of B when s keeps growing;

2. how we have chosen the threshold ϕ for which we stop reconstructing the sequence while guaranteeing an upper bound on extra memory. Moreover, we will see that the number of performed reconstruction is very low.

These two points are the subject of the next subsection.

5.2.1.1 Adaptive strategy

We now show the strategy we have adopted to change bucket size starting from an initial value B_0. We follow the classical approach of *doubling* the bucket size each time the corresponding value of n is reached. Moreover we observe that big values of B when s is small will significantly enlarge memory occupancy because most integers will be stored uncompressed. Therefore, we derive that, as a *rule of thumb*, small values of B are good for small sequences and conversely, big values of B for big sequences. Following this empirical rule, B_0 should be chosen small, e.g., equal to 16 or 32. In our implementation $B_0 = 32$.

We stop at $n = (2^7 B_0)^2/8$ integers as a good trade-off between B and n growth. In fact, remember that we are choosing $n = B^2/8$ and if B grows further beyond 4096, we start getting too large values of B which, in turn, will yield too much large values of n.

As an example, consider $(2^8 B_0)^2/8$ and $(2^9 B_0)^2/8$. For an initial choice of $B_0 = 32$, we will obtain 8 388 608 and 33 554 432 respectively. The difference between the two values of n is *huge* and, as already claimed, we will not exploit the possibility of using a better value of B for more than 25 million `append()` operations.

In other words, as already noticed, given the relation $B = 2\sqrt{2n}$, it is up to us to fix one parameter and obtain the other. Therefore, if we fix B to obtain n, the latter will grow quadratically; conversely if we fix n to obtain B, the latter will now grow as a square root.

This suggests us to use the "fast" relation $n = B^2/8$ until n becomes equal to $n_0 = (2^7 B_0)^2/8$ and next to use the "slow" relation $B = 2\sqrt{2n}$ that makes B growing much slower with respect to n and, therefore, more suitable for smaller spans of n. Once n_0 is reached, we start fixing n as large as twice the previous value and evaluating optimal bucket size as $2\sqrt{2n}$. In particular we have chosen $\phi = (2^6 B_0)^2/8$ integers.

Choice of ϕ

For an initial choice of $B_0 = 32$, our strategy will produce the following values for B and n.

$$
\begin{aligned}
B_0 &= 32 && \Rightarrow\ n = 128 \\
B_1 &= 64 && \Rightarrow\ n = 512 \\
B_2 &= 128 && \Rightarrow\ n = 2048 \\
B_3 &= 256 && \Rightarrow\ n = 8192 \\
B_4 &= 512 && \Rightarrow\ n = 32768 \\
B_5 &= 1024 && \Rightarrow\ n = 131072 \\
B_6 &= 2048 && \Rightarrow\ n = \phi = 524288 \\
B_7 &= 4096 && \Rightarrow\ n = n_0 = 2097152
\end{aligned}
$$

Using equation 20 we can evaluate the maximum extra space that our adaptive structure will use when storing ϕ integers. All we need to do is to plug in equation 20 a value for u. Unfortunately is this setting, *u is not*

known nor we can estimate it in any way. All we can do is to do the calculation in the worst ever possible scenario, i.e., when $u = n$. This is the case in which we have stored a sequence s of consecutive integers starting from 1. The space required by the Elias-Fano encoding of the sequence will be $2n + o(n)$ bits. In this setting, we derive an extra space[12] of 25%. Therefore we conclude *it is guaranteed* that when we stop reconstructing the sequence, our structure is not using more than 25% of the static Elias-Fano sequence storing the same integers. Moreover, notice that this case is *very unlikely to happen* in practical cases and u will be *much larger* than n yielding a much lower extra memory percentage. We will confirm this fact in Chapter 7 where we show some experimental results.

12 It will be actually lower, since in equation 20 we are neglecting the additional term $o(n)$.

Let us finally introduce some notation. We define B_i as

$$B_i \triangleq \begin{cases} 2^i B_0 & i = 0, \ldots, 7 \\ \sqrt{8 n_{i-7}} & i \geq 7 \end{cases} . \tag{23}$$

Let us call n_{i-1} the number of integers stored up to chunk c_i, $i = 1, \ldots, m$. This means that chunk c_i stores $n_i - n_{i-1}$ integers. Since, after the first n_0 integers, we double n each time we change bucket size, we have that chunk c_i stores $n_i = 2^{i-1} n_0$ integers, $i = 1, \ldots, m$.

5.2.1.2 Reconstructing

In this subsection, we derive an upper bound for the cost of reconstruction operations. First of all, notice that these operations are 7 in number. Therefore, as usual, we will spread the cost of very few expensive operations through a large number of low-cost operations. We proceed as follows.

Whenever we have to reconstruct our current sequence s, we know exactly its length, therefore we can allocate all resizing-array dimensions so that each reconstruction process *does not incur in any resizing operation*. Recalling the result of Section 5.1.4.2, we have that upon reconstruction we will pay:

1. the cost of having inserted $(B_i^2 - B_{i-1}^2)/8$ integers;
2. the cost of reconstructing as many integers we have so far inserted in s, i.e., $B_i^2/8$.

Summing up these two contributions, we obtain

$$4n + \frac{15n}{B_i} + 4\frac{B_i^2}{8} = \frac{7B_i^2 + 11.25B_i}{8}, \; \forall \; i = 0, \ldots, 6,$$

where $n = (B_i^2 - B_{i-1}^2)/8 = 3B_i^2/32$ and we can assume $B_{-1} = 0$.

Therefore for any $n \leq \phi$, we derive that the total number of memory accesses is

$$c = \frac{1}{8}\sum_{k=0}^{i}(7B_k^2 + 11.25B_k) + 4\lambda + \frac{15\lambda}{B_{i+1}}, \; n \in \left(B_i^2/8, B_{i+1}^2/8\right], \quad (24)$$

where $\lambda = n - B_i^2/8$, $i = 0, \ldots, 6$. Determining index i is immediate given n. Finally, the *amortized cost* for each `append()` operation is evaluated as c/n memory accesses.

More interesting is the case in which $n = \phi$, i.e., when we pay *all reconstructions*. Using the previous formula, we only need to sum up the 7 contributions, i.e., $1/8\sum_{k=0}^{6}(7B_k^2 + 11.25B_k)$. We derive that the total cost of performing ϕ `append()` operations is

$$\frac{7}{8}B_0^2\sum_{k=0}^{6}4^k + \frac{11.25}{8}B_0\sum_{k=0}^{6}2^k \quad (25)$$

that, neglecting the lower order term, is approximately $4778B_0^2$. Now, dividing for the total number of added integers ϕ, we get the amortized cost for each `append()` operation:

$$4778\frac{B_0^2}{\phi} \approx 9.33 \text{ memory accesses.} \quad (26)$$

In conclusion, comparing this result with the one we would obtain if we knew the length of the sequence, we obtain that in appending ϕ integers we are paying $9.33/4 - 1 = 1.33$ times more than a single `append()` operation. This extra time is constant and independent[13] on the initial bucket size B_0, so we get an amortized constant time for each `append()`. It worth noting that averaging this cost with the total number of added integers to the sequence, this amortized, extra, initial cost tends to become more and more negligible, as we will see later on in 5.2.4.1.

13 The lower order term in formula 25 has a negligible impact.

5.2.2 Operations

The three operations $append_s(x)$, $get_s(i)$ and $nextGEQ_s(x)$ for adaptive append-only Elias-Fano are naturally based on the corresponding operations of the append-only structures that constitute its chunks. The pseudo code for the operations is shown below.

```
1: procedure initialization
2:     length = 0;
3:     next = −1;
4:     choose initial B;
5:     n = B^2/8;
6:     chunks = [];
7:     create chunk c_0 with B;
8:     chunks.add(c_0);
9:     n_0 = (2^7 B)^2/8;
10: procedure append(x)
11:     if length > n then
12:         next++;
13:         if next < 7 then
14:             B = 2B;
15:             n = B^2/8;
16:             reconstruct c_0 with B;
17:         else
18:             n = 2n;
19:             B = √(4n);
20:             create a new chunk with B and add it to chunks;
21:     append (x− previous' chunk upper bound) to current chunk;
22:     length++;
```

Algorithm 7: Multiplying n by 8 instead of 4, at line 19, is an insidious bug! In fact, recall that each chunk stores $n_i = 2^{i-1} n_0$ integers, $i = 1, \dots, m$ and *not* $2^i n_0$. Appending a new integer to the sequence at line 21 is done with the *append* algorithm illustrated in Subsection 5.1.2.

```
1: procedure get(i)
2:     c_j = i's chunk;
3:     return c_j.get(i − n_{j−1}) + u_{j−1};
```

Algorithm 8: Again, getting an integer at line 3 is done with *get* algorithm illustrated in Subsection 5.1.2. Notice that $u_0 = 0$.

```
1: procedure nextGEQ(x)
2:     c_j = the chunk x lies in;
3:     return c_j.nextGEQ(x);
```

Algorithm 9: A 2-step *nextGEQ* pseudo code. The identification of the chunk where x lies, at line 2, can be done by binary searching over chunks' upper bounds as similarly done in Subsection 5.1.2.

5.2.3 SPACE COMPLEXITY

In this section we will derive an expression representing space occupancy and an upper bound to the the extra space with respect to the static case for the adaptive append-only structure. As for operations, we will use previous theoretical results for the append-only data structure.

As usual, remember that $w = \lg u$ are the number of bits used to encode a memory word.

As a first observation, we notice that the additional information represented by the maximum element of each created chunk has a *negligible impact* on the space occupancy of the data structure and can be discarded from the analysis.

We split the expression we want to derive in two parts: the first taking into account the space occupancy for $n \leq n_0$ (case **1.**); the second one for the opposite case, i.e, when $n > n_0$ (case **2.**).

1. Let us call u_0 the element s_{n_0}. This is an upper bound to every integer stored in the first chunk. Recalling definition 23, the space occupancy for $n \leq n_0$ integers is expressed as

$$S_1(n, u_0) = S_1^*(n, u_0) + \mathcal{E}_1(n, u) \text{ bits},$$

where, since by construction we have only one chunk which is an append-only data structure and we have already show in Section 5.1.3 that its static part is equal to the space of a static Elias-Fano encoding of the same sequence, $S_1^*(n, u_0) = S^*(n, u_0) = n\lg(u_0/n) + 2n + o(n)$ bits. $\mathcal{E}_1(n, u)$ is defined as

$$\mathcal{E}_1(n, u) = B_{i+1}\lg u + \frac{8n\lg u}{B_{i+1}},\ n \in \left(B_i^2/8, B_{i+1}^2/8\right] \text{ bits}, \tag{27}$$

$i = 0, \ldots, 6$. For any $n \leq n_0$, the construction of the data structure is described by the following picture.

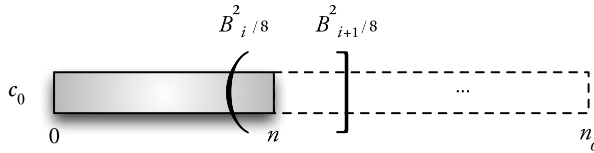

Figure 11: Construction of chunk c_0.

2. Now, for the case $n > n_0$, we similarly have

$$S_2(n, u) = S_2^*(n, u) + \mathcal{E}_2(n, u) \text{ bits},$$

where $S_2^*(n, u) = S^*(n_0, u_0) + \sum_{c=1}^{j-1} S^*(n_c, u_c) + S^*(n - n_j, u_j)$ and

$$u = \begin{cases} u_0 & n \leq n_0 \\ u_0 + \sum_{c=1}^{j-1} u_c + u_j & n > n_0 \end{cases}. \tag{28}$$

$\mathcal{E}_2(n,u)$ is equal to

$$\mathcal{E}_2(n,u) = \mathcal{E}_1(n_0,u) + \sum_{c=1}^{j-1} \mathcal{E}(n_c,u,B_{c+7}) + \mathcal{E}(n-n_j,u,B_i) \text{ bits.} \tag{29}$$

First terms of formulas $S_2^*(\cdot)$ and $\mathcal{E}_2(\cdot)$ account for the static and extra space respectively in chunk c_0; summations account for static and extra space in chunks up to the j-th, where u_c is the upper bound for the c-th chunk; last terms account for static and extra space in the chunk where n falls in[14], i.e., c_j. Notice that $j = i - 7$.

14 Notice that the determination of such chunk (that gives us index j) is immediate given n.

Doing some math, we can simplify the previous expression into

$$\mathcal{E}_2(n,u) = 2B_7 \lg u \left[1 + \frac{1-\sqrt{2^{j-1}}}{1-\sqrt{2}} \right] + \mathcal{E}(n-n_j,u,B_i) \text{ bits.} \tag{30}$$

For any $n > n_0$, the construction process is represented by the following picture.

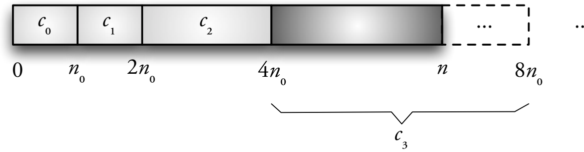

Figure 12: Light blended parts represent already constructed chunks; the darker part represents the under-construction chunk (c_3 in the example picture).

Upper bound on total space occupancy and extra memory for adaptive append-only Elias-Fano data structure

Finally, combining cases **1.** and **2.** together, we obtain the *upper bound on the space occupancy* and the *extra space function* for a adaptive append-only Elias-Fano data structure respectively

$$S(n,u) = \begin{cases} S_1(n,u) & n \leq n_0 \\ S_2(n,u) & n > n_0 \end{cases} \text{ bits,} \tag{31}$$

and

$$\mathcal{E}(n,u) = \begin{cases} \mathcal{E}_1(n,u) & n \leq n_0 \\ \mathcal{E}_2(n,u) & n > n_0 \end{cases} \text{ bits.} \tag{32}$$

Notice that B_i, in both $\mathcal{E}_1(\cdot)$ and $\mathcal{E}_2(\cdot)$, is uniquely identified by n and its definition is 23.

Now we have to show how the static parts of $S_1(\cdot)$ and $S_2(\cdot)$ are related to the space of the equivalent, static, Elias-Fano encoding storing the same integer sequence. As done before, we split the proof in two distinct cases. The first is when $n \leq n_0$ and we have already shown that $S_1^*(n,u_0) = S^*(n,u_0)$. More interesting is the case for $n > n_0$. We would like to show that

$$S^*(n_0,u_0) + \sum_{c=1}^{j-1} S^*(n_c,u_c) + S^*(n-n_j,u_j) \leq S^*(n,u), \text{ bits.} \tag{33}$$

Using property 5.1.1, inequality 33 follows automatically by adopting the splitting of s as in Figure 12 and consequently splitting u as in formula 28. Finally we can safely upper bound $S(n,u)$ by

$$S(n,u) \leq \begin{cases} S^*(n,u) + \mathcal{E}_1(n,u) & n \leq n_0 \\ S^*(n,u) + \mathcal{E}_2(n,u) & n > n_0 \end{cases} \text{ bits.} \tag{34}$$

Now our last goal is to understand how $\mathcal{E}_2(n,u)$ is related to $\mathcal{E}^*$ in order to be able to quantify the extra introduced space. From Section 5.1.3, we recall that $\mathcal{E}^* = 2B^* \lg u$ bits is the minimum extra we achieve in an append-only data structure choosing the best possible bucket size. Let us similarly call $\mathcal{E}_j \triangleq 2B_j \lg u$ bits, $j = i-7$. We wish to relate these two quantities and we proceed as follows.

Since $2^{j-1}n_0 < n \leq 2^j n_0 \ \forall\ j \geq 1$ we derive that

$$2B_j \lg u < \mathcal{E}(n,u,B_j) = B_j \lg u + \frac{8n \lg u}{B_j} \leq 3B_j \lg u,$$

that implies

$$\mathcal{E}_j < \mathcal{E}^* \leq \frac{3}{2}\mathcal{E}_j \iff \frac{2}{3}\mathcal{E}^* \leq \mathcal{E}_j < \mathcal{E}^*. \tag{35}$$

Then we will prove the following theorem.

Theorem 5.2.1. $\mathcal{E}_2(n,u) < \dfrac{\sqrt{2}}{\sqrt{2}-1}\mathcal{E}^*$ bits, $\forall\ n > n_0$.

Proof. Doing the calculations in term $\mathcal{E}(n-n_j,u,B_i)$ of $\mathcal{E}_2(\cdot)$, we obtain $\mathcal{E}(n-n_j,u,B_i) = \dfrac{4n}{\sqrt{2^j n_0}} \lg u \leq 4\sqrt{2^j n_0} \lg u$, where, since $n_j < n \leq n_{j+1} = 2^j n_0$, we can safely upper bound n with $2^j n_0$. Substituting this value in $\mathcal{E}_2(\cdot)$, we have

$$\mathcal{E}_2(n,u) \leq 2\lg u \left[2\sqrt{2n_0}\left(1 + \frac{1-\sqrt{2^{j-1}}}{1-\sqrt{2}}\right) + 2\sqrt{2^j n_0} \right].$$

Now recalling that $B_j = \sqrt{82^{j-1}n_0}$, we have

$$\mathcal{E}_2(n,u) \leq 2B_j \lg u \left(1 + \left(1 + \frac{1-\sqrt{2^{j-1}}}{1-\sqrt{2}}\right) / \sqrt{2^{j-1}}\right).$$

Now, it is sufficient to notice that the function

β function

$$\beta(j) = 1 + \left(1 + \frac{1-\sqrt{2^{j-1}}}{1-\sqrt{2}}\right) / \sqrt{2^{j-1}},\ j \geq 1 \tag{36}$$

has an horizontal asymptote in

$$\lim_{j\to\infty} \beta(j) = 1 + \frac{1}{\sqrt{2}-1} = \frac{\sqrt{2}}{\sqrt{2}-1} \approx 3.414.$$

In conclusion, using 35 we get

$$2B_j\beta(j)\lg u < 2\frac{\sqrt{2}}{\sqrt{2}-1}B_j \lg u < 2\frac{\sqrt{2}}{\sqrt{2}-1}B\lg u,$$

that is our claim. ∎

The theorem is telling us that the extra space needed by the adaptive strategy is always less than a constant times the minimum extra of append-only. Moreover, this constant is small and *at most* equal to approximately 3.4. This factor can be also considered as the "cost" we have to pay in a scenario in which we have no knowledge of both n and u.

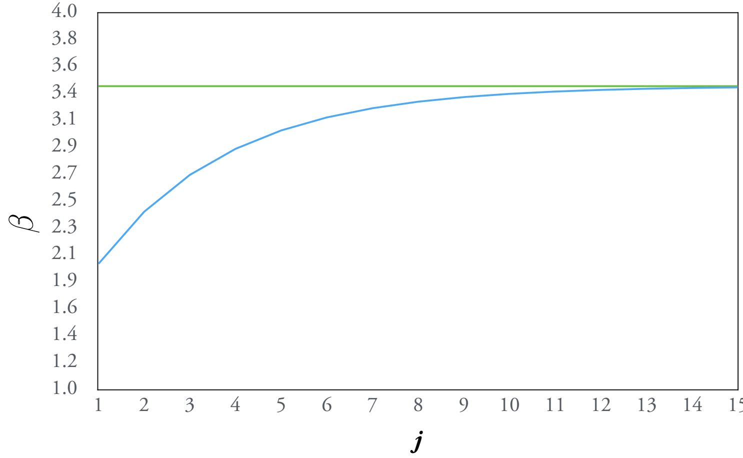

Figure 13: β function and its horizontal asymptote.

Now we are finally ready to show:

Theorem 5.2.2. $\mathcal{E}_2(n,u) = o(n)$ bits.

Proof. Following the very same approach of the proof of theorem 5.1.1, we can solve

$$\mathcal{E}_2(n,u) < 2\frac{\sqrt{2}}{\sqrt{2}-1}B\lg u < cn$$

with respect to n and concluding that $\forall\ c > 0\ \exists\ \eta = \frac{64\lg^2 u}{(\sqrt{2}-1)^2c^2} > 0$ such that previous inequality holds $\forall\ n \geq \eta$, i.e., our claim. ∎

Of course, as already shown for the append-only structure, this theorem automatically implies the *succinct bound satisfaction*.

Now we deal with the case $n \leq n_0$. As already noticed when we introduced the append-only Elias-Fano data structure, for small sequences the needed extra memory will not be negligible with respect to the over-

all memory of the data structure. However, when s keeps growing, the percentage of extra memory decreases *quickly* and we approach the limit situation in which $\mathcal{E}_1(n,u) = o(S^*(n,u))$. In conclusion we can say that *in practical situations*

$$S(n,u) \leq \begin{cases} S^*(n,u) + \mathcal{E}_1(n,u) & n \leq n_0 \\ S^*(n,u) + o(S^*(n,u)) & n > n_0 \end{cases} \quad \text{bits,} \tag{37}$$

where $\mathcal{E}_1(n,u)$ is such that

$$\lim_{n \to n_0} \mathcal{E}_1(n,u) = o(S^*(n,u)).$$

5.2.4 Time complexity

In the following we derive upper bounds on worst case performances for our two fundamental operations. The analysis naturally exploits previously found results for the first introduced data structure.

5.2.4.1 Append

We have already evaluated the number of operations due to the reconstructions that happen in the first segment of ϕ appended integers. Now, in order to evaluate the time of n consecutive `append()` operations, we distinguish two cases. The first one is when $n \leq \phi$: in this case the amortized constant time can be obtained with formula 24 and averaging with the number of added integers n.

More interesting is the case when $n > \phi$, represented below.

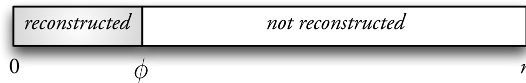

Figure 14: Reconstruction process occurs for the first ϕ added integers.

Using the result found in Subsection 5.2.1.2, is enough to do a weighted mean, as follows:

$$\frac{9.33\phi + 4(n-\phi)}{n} = 4 + 5.33\frac{\phi}{n} \text{ memory accesses, } n > \phi. \tag{38}$$

We clearly see that the more integers we add the less our amortized cost will be. However, 9.33 represents a *constant amortized upper bound* cost, $O(1)$.

5.2.4.2 Access

Before accessing an integer at a specified position, we need to identify its chunks, i.e., the append-only Elias-Fano data structure where it is stored. This preliminary operation is performed in *constant time worst case* in our implementation, resorting, again, on bitwise manipulations with no

tests nor branching, as we are going to see next. Then we just access the required integer in the identified chunk using its `get()` function. In conclusion, as similarly noticed in Section 5.1.4.4, we expect to perform an access almost as fast as an append-only Elias-Fano structure because of the extra, constant-time, contribution. In fact, we will see in Chapter 7 that this extra time is practically *negligible*.

5.2.4.3 Next Greater Or Equal

As done for the append-only structure, we derive that a $nextGEQ_s(x)$ operation costs $O\left(\lg c + \lg \frac{n_j}{B_j} + B_j\right) = O(B_j)$, where c is the total number of created chunks and j is the index of the chunk storing x. The contribution of $O(\lg c)$ is *negligible*, since c is small even for huge sequences.

5.3 Implementation key-points and APIs

In this section we overview general implementation considerations along with a detailed discussion of some selected, yet meaningful, implementation choices.

For an append-only Elias-Fano data structure, the buffer of integers to be compressed is a simple Java array of `longs` (64-bit integers) of size B. We keep a resizing-array of references to objects `SimpleSelect` as implemented in Sux4J, as well as a resizing-array of `long[]` representing the lower bits resulting from the compression step of Elias-Fano encoding of buffer integers. Finally, for each bucket b_i, we keep track of its maximum element u_i and the number of lower bits ℓ_i we need to access its compressed elements. These two values could be trivially stored in two separate arrays. However while u_i could be very large, ℓ_i is at most $\lg 64 = 6$ bits. Thus, we store them in an *interleaved fashion* as follows. We take a 64-bit integer and split its representation in two parts: the first 6 bits starting from the right are dedicated to the storage of ℓ_i and the remaining 58 bits for the representation of u_i. Notice that this choice actually restricts the dynamics of the integers we can store in our structure[15]. In this way we need only an array access and a couple of easy bitwise manipulations to retrieve both u_i and ℓ_i. Cache misses are therefore minimized [30]. The following Java code snippet[16] illustrates the needed manipulation to retrieve u_i and ℓ_i.

15 Integers range from 0 to a maximum of $2^{58}-1$, probably enough for most practical applications.

16 `info` is the array storing u_i and ℓ_i in such interleaved form.

```
1  final long LOWER_BITS_MASK = (1L << 6) - 1;
2  final long UPPER_BITS_MASK = ~ LOWER_BITS_MASK;
3  final long li_ui = info[i];
4  final long li = li_ui & LOWER_BITS_MASK;
5  final long ui = (li_ui & UPPER_BITS_MASK) >> 6;
```

As we can see only few, constant time, operations suffice. Clearly, the two masks are constants[17] and need not to be computed each time.

17 Use modifiers `static` and `final`.

Regarding the adaptive data structure, we basically need to maintain a resizing-array of append-only structures. Therefore, the append-only data structure is the *backbone* of our adaptive structure but it can be used in a *standalone* way too. The meaningful point we would like to illustrate is the code of the `get()` function.

```
1  public long get(final int index)
2  {
3    final int d = mostSignificantBit(index) - msbn0;
4    final int MASK = d >> 31;
5    final int x = d + ((d + MASK) ^ MASK) >> 1;
6    final int id = x + ((n0 << x) - index >>> 31);
7    AppendOnlyEliasFano s = chunks.get(id);
8    return s.get(index - (((id - 1) >>> 31) ^ 1) *
9    ((n0 << id - 1) + 1)) + s.prevUpper;
10 }
```

Names are self-explanatory. `msbn0` tells us which is the most significant bit of n_0, while the static method `mostSignificantBit(index)` calculates it for the input `index`. This method can be easily implemented in Java using the built-in static method `numberOfLeadingZeros()` of `Integer` and/or `Long` classes[18] that returns the number of zero bits to the left of the most significant bit of an integer.
We can use this method to implement `mostSignificantBit()` in the following way[19]

```
public static int mostSignificantBit(final int x)
{
  return 31 - Integer.numberOfLeadingZeros(x);
}
```

18 For the class `Integer` API: `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html`.

19 This is trivially generalized for the `long` case.

When `index` is greater than n_0 the computed difference `d` tells us the power-of-2 that multiplied by n_0 is closer to `index`, i.e., $2^{\mathtt{d}} n_0$ individuates one of the thresholds in Figure 12. The problem, here, is when the computed difference is negative, i.e. for those values smaller than $2^{\mathtt{msbn0}}$. This case is handled in lines 4, 5 and 6. What we want is that whenever the difference is positive then $2^{\mathtt{d}} n_0$ is the threshold we are looking for, but when the difference is negative just returns us `0`, since `index` will be surely located in chunk c_0. To implement this, we can compute the *absolute value* of the difference and sum it to the difference itself. This will give us exactly `0` when the difference is negative or twice the absolute value of the difference when it is positive. Then just divide it by two to obtain exactly what we want and save it in `x`.

Then we just have to decide if `index` is located to the left (`index` is stored in $c_{\mathtt{x}}$) or to the right (`index` is stored in $c_{\mathtt{x+1}}$) of such threshold. This can be trivially done with a comparison with $2^{\mathtt{x}} n_0$ but this is exactly what we want to avoid: no tests nor branching should be involved. To avoid performing a test, we compute again the difference between the threshold $2^{\mathtt{x}} n_0$ and `index`. If this difference is positive then `index` is stored in the (`id=x`)-th chunk, otherwise in the (`id=x+1`)-th. In other words we have to sum a 1 if the difference is negative or a `0` if it is positive. But this information is already encoded in the difference itself, which, left shifted by 31 positions[20] gives us its sign bit, i.e., the proper value to sum.

20 Signed integers are 32-bit integers in Java.

Last but not least, now that we have the chunk's identifier we have to `get` the proper element, i.e. pick the element of index `index-(n0«id-1)+1` from chunk $c_{\mathtt{id}}$. This is true except for the chuck c_0. In such case, we should just return the integer in position `index` from it. We can therefore multiply the term `(n0«id-1)+1` by `0` when `id=0` or by 1 when `id=1`. Again another difference! We can compute the sign bit of `id-1` and complement it using a bitwise XOR with a 1.

Remember we have also to sum previous chunk's upper bound. This value is stored inside the *same chunk* we are accessing, so that we do not incur in any further cache miss.

Iterating over the structure

Another meaningful point to describe is how it is possible to iterate over the compressed integers in the sequence *without* using any *select* operation. A *select* is, in fact, the most expensive operation in the *access* procedure and avoiding to perform it reveals in a great optimization.

We recall from Section 4.1.2 that to retrieve the i-th integer of an Elias-Fano-compressed sequence, we need to concatenate the i-th upper bits with the i-th lower bits. Reading the i-th lower bits is straightforward: we just need to access a proper word of the lower-bits array. Retrieving the i-th upper bits require to know how many zeros are present up to the position of the i-th one. This information can be computed by making a simple difference as follows

```
nextOne = selector.bitVector().nextOne(nextOne + 1);
upperBits = nextOne - ones++;
```

by just keeping track of two variables: `nextOne` which is initialized to -1 and `ones` which is initialized to `0`. `selector` is clearly the `SimpleSelect` structure from whose `bitVector` we retrieve the i-th upper bits.

The `nextOne(p)` operation of a `bitVector` returns the position of the first bit set after position `p`.

In conclusion, we are just sequentially accessing a bitvector an making a difference with a counter. As a net result we manage to *almost halve* the accessing time, as we are going to see in Chapter 7.

The discussed optimized iteration implies that our structures *should not* implement the so-called Java interface `RandomAccess` (as `ArrayLists` do), even if they effectively support random access in constant time. This may *apparently* seem a contradiction since constant-time random access is, indeed, the most important feature of Elias-Fano. This design choice is actually dictated by the Java language which encourages programmers to check whether the structure is an `instanceof` such interface[21] *before* iterating over it. Doing so, experienced programmers should understand to prefer using an iterator for sequential accessing rather than a simple `for` loop.

21 http://docs.oracle.com/javase/8/docs/api/java/util/RandomAccess.html

In other words, the loop[22]

22 s is an instance of one of our append-only Elias-Fano structures.

```
for (Long integer : s)
{
  // do something on integer...
}
```

runs *twice faster* than this loop

```
for (int i = 0, length = s.size(); i < length; i++)
{
  Long integer = s.get(i);
  // do something on integer...
}
```

We finally show the Java APIs of the implemented succinct data structures. These implemented structures form a library called Ef4J - *Elias-Fano Succinct Data Structures for Java* and it is freely available under proper license.

Both structures extend the operations defined in `AbstractAppendOnlyMonotoneLongSequence`. This is an `abstract` Java class, meaning that it *cannot be* instantiated but serves to *define a behaviour common to all extending classes*.

The class, in turn, extends the `AbstractMonotoneLongSequence`, which implements the Java interface `List<Long>`, therefore offering to implement all methods in the interface (`.clear()`, `.isEmpty()`, `.size()`, `.iterator()`, just to name a few[23]). The role of the append-only variant of the class is to provide *proper restrictions*, since it represents an append-only monotone sequence (as an example, set a specific element to a given value or inserting at arbitrary positions are not permitted operations).

Picture 15 shows the implemented hierarchy of classes.

23 http://docs.oracle.com/javase/7/docs/api/java/util/List.html

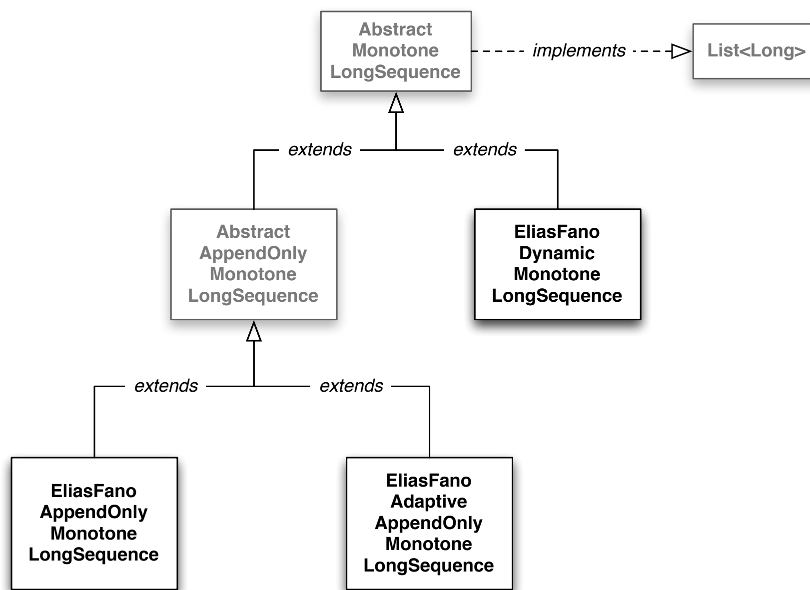

Figure 15: Simple tree hierarchy of the implemented Java classes. Gray classes are meant to depict abstract classes. The *dynamic* Elias-Fano structure will be introduced in Chapter 6.

Constructors

```
public class EliasFanoAppendOnlyMonotoneLongSequence
extends AbstractAppendOnlyMonotoneLongSequence implements
Cloneable, Serializable
```

Create a new, empty, append-only Elias-Fano data structure with a specified bucket size B.

```
EliasFanoAppendOnlyMonotoneLongSequence(int B)
```

Create a new append-only Elias-Fano data structure with a specified bucket size and a specified initial capacity.

```
EliasFanoAppendOnlyMonotoneLongSequence(int B, int
capacity)
```

```
public class EliasFanoAdaptiveAppendOnlyMonotoneLongSequence
extends AbstractAppendOnlyMonotoneLongSequence implements
Cloneable, Serializable
```

Create a new, empty, adaptive append-only Elias-Fano data structure.

```
EliasFanoAdaptiveAppendOnlyMonotoneLongSequence()
```

Create a new, empty, adaptive append-only Elias-Fano data structure with a specified initial bucket choice.

```
EliasFanoAdaptiveAppendOnlyMonotoneLongSequence(int B)
```

The next page reports the most important methods of the structures (trivial ones have been omitted). The are clearly the same for both structures.

Methods

Appends the specified integer to the sequence.

```
boolean add(long integer)
```

Append all integers in the specified collection.

```
boolean addAll(Collection<? extends Long>)
```

Returns the integer at the specified index.

```
Long get(int index)
```

Returns the smallest integer of the sequence that is greater than or equal to the one specified or just -1 if such value does not exist.

```
Long nextGEQ(long integer)
```

Return true if the sequence contains the specified integer.

```
boolean contains(Object o)
```

Return true if the sequence contains all the integers in the specified collection.

```
boolean containsAll(Collection<? extends Long>)
```

Returns the sub list specified by the given range.

```
List<Long> subList(int from, int to)
```

Returns the number of bits used by the structure.

```
int bits()
```

Returns an array containing all of the elements in the sequence in proper sequence (from first to last element).

```
Long[] toArray()
```

Trims the capacity of this sequence instance to be the sequence's current size. This operation affects all inner stored data structures that can possibly be trimmed.

```
void trimToSize()
```

Returns an iterator over the elements in this list in proper sequence.

```
Iterator<Long> iterator()
```

Returns an iterator over the elements in this list in proper sequence in the specified range, extremes included.

```
Iterator<Long> iterator(int from, int to)
```

Returns a copy of the object.

```
EliasFano(Adaptive)AppendOnlyMonotoneLongSequence clone()
```

5.4 Brief summary

We have introduced two new append-only succinct data structures, that use Elias-Fano integer encoding to store a monotone sequence of increasing integers, while growing in dimension. Our data structures and algorithms are relatively *simple* and *elegant*. This is a point of strength since they can be implemented and used in practical applications as we will see in Chapter 8.

"Simplicity is prerequisite for reliability." E. W. Dijkstra

With respect to the static case, we are adding some additional space, that, with the help of some theorems, we have proven to be a o-term of the number of stored integers. We have also remarked that this does *not* always imply practical usage of succinct data structures, because of the potentially large constants involved. On the other hand, we will see by some experiments (Chapter 7 and 8) that our structures *really use a negligible extra space* for relatively large sequences.

Summing up, we can state the following concluding theorem, solving the problem we pose at the beginning of the chapter.

Theorem 5.4.1. There exists an encoding strategy for an *append-only* sequence of monotonically increasing integers which takes

$$EF(s[0,n)) + o(n) \text{ bits}, \forall n = \omega(\lg^2 u),$$

while supporting *append* and *get* operations in constant time worst case. $EF(s[0,n))$ is equal to $n\lceil \lg(u/n) \rceil + 2n + o(n)$ bits and it represents the space needed by the equivalent, static, Elias-Fano succinct data structure storing the sequence, where n is its current length and u the current maximum stored integer.

Proof. Immediate from previous analysis. ∎

6 Dynamic

The next design step, which naturally arises from previous chapter's material, is taking into account a fully dynamic strategy, able of supporting additions and deletions in random positions of the sequence. The new problem that we would like to tackle is formalized as follows.

Problem. Starting from a *non-empty* append-only monotone sequence s, encoded with the Elias-Fano strategy, dynamize it so that we can *insert* and *delete* its integers, keeping the sequence as much compressed as possible.

Being a more general problem it offers a wider number of practical implications, but it is also a *much harder* problem to solve. The third structure we propose is, therefore, the *most powerful* and *flexible* of those presented, since we are not obliged to append monotonically increasing integers and we can change our mind deleting what we have previously add. This power, of course, *does not come for free*: while the *usability is greatly improved*, we use additional space and we are not performing constant-time operations any more (except for few exceptions).

Trade-off

This is again a very good example of a typical software trade-off: flexibility if often sacrified for efficiency and viceversa.

Fundamental assumption

Since we are dynamizing a compressed sequence, a first yet fundamental assumption is made: *additions and deletions are not as frequent as append and access operations.* This is reasonable since we want to keep the largest amount of data compressed and use the sequence in an append-only fashion mostly.

In the rest of the chapter we present the implemented strategy along with our design choices.

6.1 Algorithmic description

Suppose we are given an append-only monotone sequence s as described in previous chapter. Until the moment we decide to dynamize it, we use it in an append-only way as usual. We now want to delete some integers and make the sequence grow not necessarily from the end: we make it a dynamic one. This choice implies "attaching" a *dynamic index* to the already-formed s that will contain the integers we would like to add/remove. This index is, in turn, broken into a collection of smaller indices, one for each compressed bucket of the sequence. The core implemented strategy is to accumulate to-be-added/removed integers in the proper bucket index until it becomes full. At that point, we just empty the index integrating its

content with the compressed block. This latter process means *reconstructing* the block.

In just few lines we have explained a high-level overview of the strategy. This is simple and has potential to work well in practice. Let us now dig into the details of its design.

The backbone underlying structure is an Elias-Fano append-only sequence whose bucket size is fixed to B. Being n the length of the sequence, we have n/B compressed buckets. Till now, nothing new.

The attached dynamic index is made up of n/B bucket-indices, say $I_0, \ldots, I_{n/B-1}$. Each of them is made up of two *ordered* arrays: one for the integers to add, the other for the ones to be removed. Let us use in the following the names of `additions` and `deletions` to indicate such arrays, as in our code implementation. They are kept ordered since this avoids to sort them every time we need to access an integer, as we will see next. The arrays are resizing ones but their size has been bounded by $B/(4 \lg n) = \sqrt{2n}/\lg n = o(n)$ using best B.

As clear, now each bucket size can change and differ from B. Therefore, when resolving an access query, we need to first identify the bucket containing the desired integer. For this preliminary operation we need to keep track of buckets' sizes. These sizes are stored in a resizing array too, in a *prefix-sum fashion*: if `sizes` is the name of the array, its i-th element is

$$\texttt{sizes}[i] = \sum_{j=0}^{i} \texttt{sizes}[j], \; i = 0, \ldots, n/B - 1.$$

This array takes n/B words. Apart from buckets' sizes we do not need to hold any other extra information.

Dynamic index space

Given the described structure, the space required by the whole dynamic index is *at most* (when all bucket-indices are full)

$$\left(\frac{n}{B}\frac{B}{2\lg n} + \frac{n}{B}\right) w = \left(\frac{n}{2\lg n} + \sqrt{n/8}\right) \lg u \text{ bits} = o(n) \text{ bits},$$

so that the whole dynamic sequence will, theoretically, take the same space as an-append-only one plus $o(n)$ bits. We say *theoretically* for the same reasons explained in Subsection 5.1.3.2. The initial dimension of each bucket-index is just 2, so that the whole dynamic index takes *at least* (when all bucket-indices are empty) $3n/B$ words $\approx \sqrt{n}$ words $= o(n)$ bits.

Updating the structure

When a bucket needs *reconstruction* we update (obviously) lower bits, upper bits (selection structure) and bucket size. What about the `info` array? When we add an integer to a bucket, u cannot change because integers greater than u we will be added to a proper subsequent bucket. So nothing to worry about an addition. But u can still be deleted from its

bucket. If this is the case, we delete it but we do not update it in `info`. This choice is *safe*, in the sense that integers in the next bucket will be encoded with respect to the old value of u which is larger than the new one. This also reveals in a better compression gain. In conclusion, the upper bound values are *never* updated. The number of lower bits needs, instead, to be updated because the bucket size can change.

Merging and splitting

When a bucket will contain more than $2B$ integers it will be split in two buckets: the first counting B integers; the second counting the remaining ones. When a bucket becomes too small, i.e. less than $B/2$ integers, it is merged with the next bucket if and only if the sum of the dimensions of these two blocks is less than $2B$; otherwise it is just re-compressed. In this way, we are sure that the newly merged bucket will not cause any further splitting because of its size. These are general and largely used techniques in compressed dynamic data structures and have been inspired to the ones described in [15].

The following picture shows a graphical representation of the data structure.

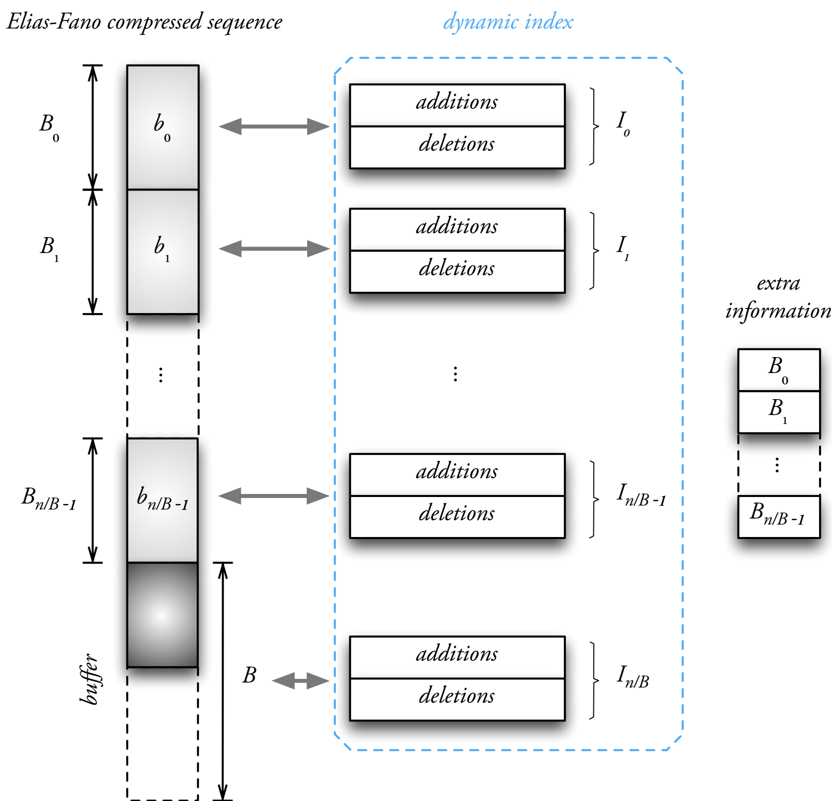

Figure 16: Dynamic Elias-Fano compressed structure. The two main components are shown: the dynamic index and the Elias-Fano compressed sequence. For ease of representation, `info`, `selectors` and `lowerBits` arrays have been omitted. B_i represents the i-th bucket's size, $i = 0, \dots, n/B-1$.

The reconstruction strategy just presented is quite general and describes a typical reconstruction case. Almost all programming effort is devoted to handling *corner cases*, such as the one of the last bucket under construction, i.e., the buffer. As simple examples: the buffer is never split since it accumulates uncompressed integers until it becomes full and, therefore, compressed; if it needs to be merged with the previous bucket, it is just emptied and its index cleared but not truly *deleted* as it occurs for any

other bucket. Also the reconstruction of the buffer differs from all the others: if the size B is not reached, we merge the index content with the buffer without any compression.

These examples are meant to provide just few clues about the complexity of programming this data structure. The handling of such corner cases implies more lines of code; more logic and in general, more complexity.

6.2 Theoretical improvement

Binary Search Trees

In this section we present an advanced discussion concerning Binary Search Trees [28, 6] (BSTs) and their application to our data structure.

Another possibility for implementing each bucket index could be using two binary search trees, since they support basic dictionary operations (*search/insert/delete*) in a time proportional to their own height.

Suppose, in the following, we are given a BST of height h storing n items. Its power lies in this key observation: *if complete*[1] *then* $h = \lfloor \lg n \rfloor$ *and each operation takes* $O(\lg n)$. Therefore, working with BSTs allow us to insert/search each integer in logarithmic time and we do not need to touch any other items as we do for resizing arrays. This would be a *clear advantage*. However, there are some drawbacks we point out below and are nicely illustrated by the following picture.

1 Perfectly balanced, except for bottom level.

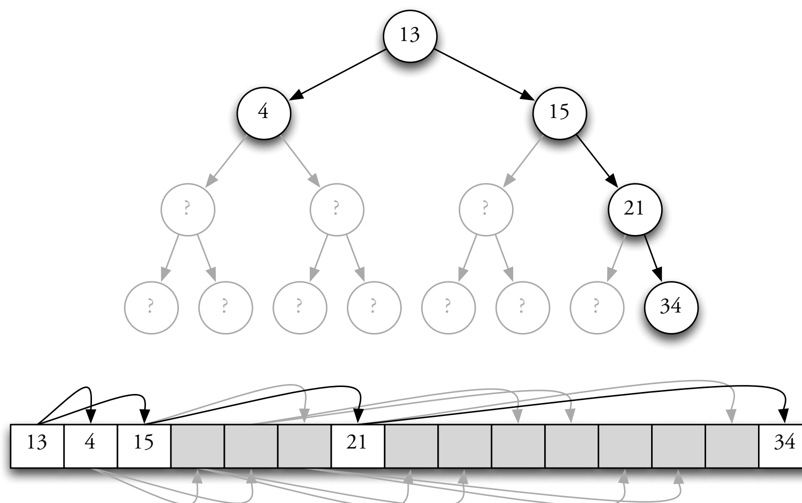

Figure 17: Not balanced BST stored in a pointer-based manner and in an array.

- As said, in order to have a worst-case performance guarantee of $O(h)$, we need to maintain it *balanced* so that $h = \lfloor \lg n \rfloor$. If not, each operation could cost $\Theta(n)$ (linear chain). What we would need is a *self-balancing* implementation, such as *red-black* or *AVL trees*.
- If they are implemented in a pointer-based manner, then we need *at least* $2\lceil \lg n \rceil$ bits more for each node, ending up with $2n\lceil \lg n \rceil$ additional bits.
- A pointer-based implementation could be *dangerous for the cache*: nodes could be spread through out all memory hierarchy.

- Even if they are stored in resizing arrays[2], thus eliminating the need of pointers, then they can lead to a *great waste of space*. Inserting items in a breadth-first search order wastes *no space only if the tree is complete*. The problem in our case is that we do not know the tree (no search is possible) and we do not know how many integers we will insert. Figure 17 shows this behaviour.

  The array representation of a binary tree is a possible method used for storing *binary heaps* [28, 6]. If we use min-heaps, we do not incur in any space waste and we support insertions in logarithmic time as well. The problem arises when we want to retrieve all integers in ascending order: this operation is performed in $O(n \lg n)$ time, the same as for a *sorting* operation (which we want to avoid).

2 Node at position i keeps its children at positions $2i+1$ (the left one) and $2i+2$ (the right one) respectively. Assume the root be stored at position 0. The parent of node i is kept at position $\lfloor (i-1)/2 \rfloor$.

Binary Heaps

These are the considerations that have driven our design. Furthermore, a simple ordered array permits *the fastest sequential scan as possible*, which is fundamental during a random access operation (see next). A BST allows to retrieve all keys in order too using an *in-order visit* (linear time) but this is not performed as fast as for arrays (we need to allocate an `Iterator` object during an access operation, which is costly and uses wrapper methods to access collection's elements).

In general, whether a data structure is better than another is often determined by the *frequency of operations* to be performed [28, 6]. Our case is an excellent example: if insertions/deletions are very frequent, say more frequent than accesses, than we would use tree data structures as previously discussed for their great dynamism[3]; if accesses are predominant (as it should be in a compressed data structure) than it could be almost useless using this kind of data structures.

Frequency of operations

3 However, remember we are using a succinct data structure, so: if we need much more dynamism, *are succinct data structures the best choice in this case?*

We need to keep in mind our fundamental assumption. Since insertions/deletions should not be as frequent as access operations, we require these latter being implemented as fast as possible. Therefore, we use an *eager* approach to the problem: we do as much work as possible up front in order to go as fast as possible later on [28]. In conclusion, notice that the dimension of such arrays will be *very small*[4] compared to bucket size, so that the overhead introduced by a BST does not worth the benefit for the above considerations.

4 Just for illustrative purposes, if n is 2 million integers, than best B will be 4000 and each array will be *at most* of length 50.

For the storage of buckets' sizes we could resort on a self-balancing BST too. In particular we could store in each internal node the number of integers present in its left subtree. In this way, binary searching over the array or the tree uses the same algorithmic idea. Picture 18 offers an example.

The clear advantage over a simple array is that, again, we update a bucket's size in $O(\lg(n/B))$ keeping the tree balanced, against $O(n/B)$ in the worst case. The disadvantages of this choice is that also retrieving a bucket's size is done in logarithmic time and this is a problem for a random access operation (see next). Same previous considerations about BSTs apply to this case. Moreover, we would need to maintain not just the n/B buckets' sizes (that would become the leaves of the tree), but also the

internal nodes which would be *at least* $n/B-1$ (if n/B is a power of 2, otherwise even more than that). Overall, we would use the *double* of the space.

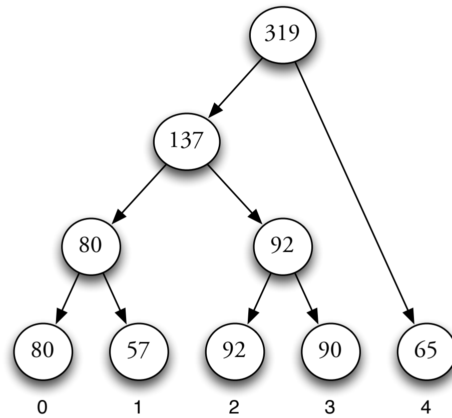

Figure 18: An example of how we can use a BST to maintain buckets' sizes. Each internal node stores the number of items in its left subtree. In this case we have 5 buckets of sizes, respectively, 80, 57, 92, 90, 65.

In conclusion, the advantage of using BSTs instead of ordered arrays (our choice) will just be *theoretical* and *not practical*. This advantage is summarized in what follows.

If we opt for BSTs and use best B we can support insertions/deletions in $O(\lg\sqrt{n})$ amortized instead of in $O(\sqrt{n})$ amortized as with ordered arrays. Now we will have a dynamic index space of

$$\left(\frac{n}{2\lg n}+\frac{\sqrt{2n}}{\lg n}\lg\frac{\sqrt{n/2}}{\lg n}+\sqrt{n/2}\lg\sqrt{n/8}+\sqrt{n/8}\right)\lg u \text{ bits,}$$

which is anyway $o(n)$ bits as before, even though significantly greater.

This is the only theoretical advantage of BSTs and has *no impact* on the other operations we are going to see. Therefore, from now on, we assume the design and structure of Section 6.1.

6.3 Operations

This section offers a description of the *core operations* of the structure with their time complexities. The supported operations are the same for the other implemented structures except for the *remove* operation.

6.3.1 Add/Remove

In the general case, an addition (removal) of an integer from a block consists in the following steps:

1. binary search over `info` array in order to identify the bucket the integer has to be added (removed);
2. insertion in the proper ordered-array;
3. increment (decrement) the length of the sequence and bucket's size.

The complexity of the operation is clearly $O(\lg(n/B)) + O(B/(4 \lg n)) + O(n/B)$. If we are using best B, we obtain $O(\sqrt{n})$ as already mentioned in previous section.

Appending an integer in last position in the buffer or deleting the last item from it represent two trivially-handled, constant-timed, corner cases.

As said, adding/removing some integers may cause a reconstruction of a bucket, a merging or a splitting. In the general case, all these operations count two main subroutines: iterating over the bucket to reconstruct (to properly integrate the index content with the bucket) and re-compressing it. Therefore, the complexity of an add/remove operation needs to be averaged by the complexity of such occasional[5] routines.

The aforementioned times are therefore *amortized*, not worst-case running times.

5 They should not be as frequent as the general case behaviour for the assumption made at the beginning of the chapter.

Furthermore, before performing a removal we have to be sure we are deleting something which *truly belongs* to our sequence. We have first to check if the sequence contains the to-be-removed integer. For that preliminary task, just a *nextGEQ* operation suffices. The problem is that this preliminary *nextGEQ* practically constitutes the *whole* running-time of a remove operation which costs *one order of magnitude more* than an addition/removal. Therefore we have introduced the following optimization: we do *not* check for the inclusion of the to-be-removed integer but it is always inserted in its proper bucket-index. It will be the reconstruction process in charge of discarding an integer to remove if it is not present in the sequence. This *greatly* reduces the time for a removal.

Malicious usage

However, there is an important trade-off to mention. Since we accumulate integers in the index even if they will not be removed because not present at all, there is the potential risk of a *malicious user* trying to remove such integers and, therefore, inducing a lot of reconstruction processes. Moreover, this will cause problems in a random access operation since the structure will be misled about the number of deletions to

perform and may return the wrong item. In conclusion, it will be *responsibility of users* to be sure they are deleting something truly belonging to the sequence.

In the following we show some skeleton pseudo code for *add* and *remove* operations.

Algorithm 10: `buckets` is a variable keeping track of the number of created buckets. All other routines are the ones already met in previous chapters.

```
1: procedure add(x)
2:     if x ≥ last then
3:         buffer.add(x);
4:         bucket = buckets;
5:     else
6:         bucket = binarySearchOverU(x, 0, u.size());
7:         insert(I_bucket.additions, x);
8:     B_bucket++;
9:     if I_bucket.additions.isFull() or buffer.isFull() then
10:        if bucket ≠ buckets then
11:            newB = B_bucket;
12:            if newB ≥ 2B then
13:                split in 2 buckets;
14:            else
15:                rebuild with newB;
16:        else
17:            newB = buffer.size() + indexSize(bucket);
18:            if newB < B then
19:                merge I_bucket.additions' content with buffer;
20:            else
21:                compress(buffer);
22:                buckets++;
23:                buffer.clear();
24:                create new bucket index;
25:        I_bucket.clear();
26:    length++;
27: procedure indexSize(bucket)
28:    return I_bucket.additions.size() − I_bucket.deletions.size();
```

Algorithm 11: *remove* algorithm.

```
1:  procedure remove(x)
2:      if x = last and !buffer.isEmpty() then
3:          buffer.remove(x);
4:      else
5:          bucket = binarySearchOverU(x, 0, u.size());
6:          insert(I_bucket.deletions, x);
7:      B_bucket− −;
8:      if I_bucket.deletions.isFull() then
9:          if bucket ≠ buckets then
10:             newB = B_bucket;
11:             if newB ≤ B/2 then
12:                 nextBlockDim = B_bucket+1;
13:                 finalBlockDim = newB + nextBlockDim;
14:                 if finalBlockDim < 2B and nextBlockDim > 0 then
15:                     merge bucket bucket with bucket bucket+1;
16:                     buckets− −;
17:                 else
18:                     rebuild with newB;
19:             else
20:                 rebuild with newB;
21:         else
22:             newB = buffer.size() + indexSize(bucket);
23:             merge I_bucket.additions' content with buffer;
24:     length− −;
```

6.3.2 Access

This is, by far, the most delicate operation. Most of the data structure's complexity is due to this operation. Let us briefly explain why. Suppose we want to retrieve the integer of index i from our sequence. Adding and deleting some integers will alter the position of *potentially many* integers, so that the i-th integer that we access *may not be* the i-th one any more. More precisely, in the general case, having some additions to perform will shift the i-th integer to the left; viceversa if we have some deletions the i-th integer will be shifted to the right. This is to say that we have to *recompute* the correct position of the accessed integer.

This time, instead of showing some boring pseudo code, we try a different approach with the help of picture 19. First of all, we need to understand which is the bucket storing such integer. This preliminary operation is performed by binary searching the array storing buckets' sizes. Now that we have our bucket index, say ℓ, we start accessing the inte-

ger corresponding to the i-th integer of the compressed sequence. Let us call it v and let i' be its position relatively to bucket ℓ. Now we determine $j_v = |\{n : \texttt{additions}[n] < v\}|$ and $k_v = |\{n : \texttt{deletions}[n] \leq v\}|$ as the number of integers in `additions` and `deletions` respectively that are less or equal than v. We finally access ℓ-bucket's element p at position $i'' = i' - j_v + k_v$. Picture 19 illustrates what may happen next. Let us analyse case by case.

Four cases can happen. There are obviously another two, dummy, corner cases. The first is when $j_v = k_v = 0$ and can just return v in constant time $O(1)$; the other is when $i' < 0 \vee i' \geq B_\ell$ and we resort on a merge-like iteration (see below).

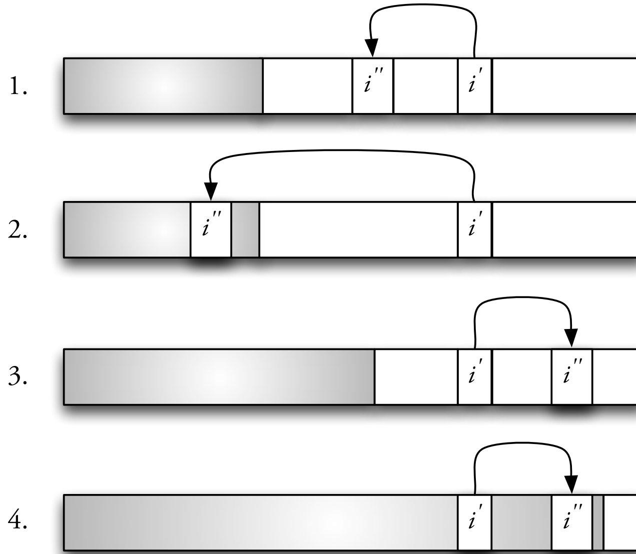

Figure 19: The different cases can happen during a random access operation inside a bucket. Light gray blended part represents the portion of the compressed bucket which is subject to updates, i.e., some additions and/or deletions occur in it.

1. The first case we take into account is when $j_v \geq k_v$ and therefore $i'' \leq i'$. In this case if $p > \texttt{additions}[j_v] \wedge p > \texttt{deletions}[k_v]$ is true, i.e., i'' falls *outside* the portion of the sequence which is subject to updates, than we are sure p is the correct integer to return. This case is handled in $j_v + k_v + O(1)$ and has a worst case guarantee of $O(B/(2 \lg n))$.

2. The second case happens again when $j_v \geq k_v$ but unfortunately $p \leq \texttt{additions}[j_v] \vee p \leq \texttt{deletions}[k_v]$ is true, meaning that p falls inside the gray portion. If this happens, than we use a merge-like iteration to return the correct integer. This operation is very similar to the subroutine *merge* of Merge Sort which takes $x + y$ comparisons if x and y are the length of the two sorted sequences to merge. Our algorithm has, however, three important differences.

   a) We have to merge integers from *three* ordered sources, i.e., the compressed integers from the bucket; the integers in `additions` and in `deletions`.

b) We do not need to merge *until the end* of the sequences but only up to the i'-th integer (the one that we will eventually return).

c) We are *not truly merging* the three sequences: we do not want to return the whole ordered, resulting, sequence but just understand which is its i'-th element (therefore no extra space is allocated as it occurs in Merge Sort).

This is the most expensive operation and in the worst case it runs in $O(B + B/(2\lg n)))$.

3. The third case happens when $k_v > j_v$ and i' will be shifted forward: $i'' > i'$. This case is a little bit more *subtle* because, since we are moving to the right, we do not know if the skipped part (from i' to i'') is subject to modifications. Therefore we have to update both j_v and k_v in j_p and k_p. At this point, however, we must ensure anyway whether the candidate integer, say p', at position $i' - j_p + k_p$, is not placed in `additions` or `deletions` by checking the usual condition of case 1. If it evaluates to true, than we can return the integer. Obviously the complexity is the same as the one of case 1.

4. The fourth case is the counter-part of previous one, i.e., when the condition in case 2. yields true and, therefore, we have to resort on a merge-like iteration.

Summing up, apart from the preliminary binary search in $O(\lg(n/B))$, in half of the cases we retrieve an integer spending additional $O(1)$ while in the other half of the cases the retrieval costs more. In conclusion, the accessing time really depends on which integers we decide to add/delete but has a worst case performance guarantee of

$$O(\lg(n/B) + B + B/(2\lg n)) = O(\sqrt{8n}). \tag{39}$$

6.3.3 Next Greater Or Equal

We do not have to change a single line of code for the *nextGEQ* operation once we have properly implemented the merge-like iterator. We proceed as explained in Subsection 5.1.4.5. Its complexity is the one of formula 39.

6.4 API

The implemented operations are exactly the same as for append-only structures, but we also support *remove* and *removeAll* operations.

We point the reader back to Section 5.3 for the other operations contained in the API. Moreover, Figure 15 illustrates the implemented, minimal, Java class hierarchy.

Additional Methods

Delete the specified integer from the sequence if it is present.
`boolean` **`remove`**`(long integer)`
Remove all integers in the specified collection.
`boolean` **`removeAll`**`(Collection<? extends Long>)`
Dynamize the sequence.
`void` **`dyamize`**`()`
Tells us if the dynamic behaviour is ON or OFF.
`boolean` **`isDynamic`**`()`

Constructors

`public class` **`EliasFanoDynamicMonotoneLongSequence`** `extends` **`AbstractMonotoneLongSequence`** `implements` **`Cloneable`**`,` **`Serializable`**

Create a new, empty, append-only Elias-Fano data structure with a specified bucket size B.
`EliasFanoDynamicMonotoneLongSequence``(int B)`
Create a new append-only Elias-Fano data structure with a specified bucket size and a specified initial capacity.
`EliasFanoDynamicMonotoneLongSequence``(int B, int capacity)`

6.5 Brief summary

We have introduced a new *dynamic* succinct data structure that makes use of the Elias-Fano integer encoding to store a monotone sequence of non-decreasing integers that grows in dimension.

As already done for the append-only data structures, we can state the following concluding theorem, solving the problem we pose at the beginning of the chapter.

Theorem 6.5.1. There exists an encoding strategy for a *dynamic* sequence of monotonically increasing integers which takes

$$EF(s[0,n)) + o(n) \text{ bits}, \forall n = \omega(\lg^2 u),$$

while supporting *insert/delete* in $O(\lg \sqrt{n})$ amortized time and *get* in $O(\sqrt{8n})$ worst case. $EF(s[0,n))$ is equal to $n\lceil \lg(u/n) \rceil + 2n + o(n)$ bits and it represents the space needed by the equivalent, static, Elias-Fano succinct data structure storing the sequence, where n is its current length and u the current maximum stored integer.

Proof. Immediate from previous analysis. ∎

Part III

Practical Impact

7 Experimental results

In this chapter we validate our Java implementation of the previously introduced succinct data structures, showing the results of the performed experiments. In all experiments we will confirm and validate our theoretical results.

Test machine

All experiments were run on a 2.4 GHz Intel Core 2 Duo with 3 MB of L2 cache, 4 GB 1067 MHz DDR3 internal memory (RAM) and 7200 RPM SATA hard drive, running Mac OS X 10.10-64 bits. As said in Section 2.8 the java compiler has been instructed with optimization `-O`.

We have run lots of experiments varying dimension of sequences and confirming the theoretical models of previous chapters. While being impractical reporting all those numbers, we will concentrate on a representative sample sequence. Let s be its name in the following.

This sequence[1] consists of $n = 2\,348\,411$ increasing integers, ranging from 1106 to $u = 1\,759\,782\,123$ ($\lceil \lg u \rceil = 31$). The dimension of such sequence is ≈ 18.79 MB.

1 It has been generated using a simple Java program, called `MonotoneSequenceGenerator.java`, whose example output was:

```
monotoneSeq1500-2348411.txt
  1106
  2095
  2622
  3802
  5029
  6160
  6830
  6877
  7189
  ... (2 348 402 lines missing)
```

The maximum gap between an integer and the following one has been fixed to 1500. Choosing another value for the maximum gap, e.g. 150 or 15 000, will *proportionally scale* all curves. We can, therefore, focus on and show results for that fixed choice of maximum gap.

During experiments, both sequence and query pattern have been read with a linear scan of a simple array, therefore minimizing interference (creating query positions during tests as well as the sequence itself produces, instead, a great perturbation on results, mainly because of the generation of pseudo-randomly generated numbers). Before each run, cache was purged to *enforce fairness* of temporal results among different tests.

Before every timing, we call `System.gc()` to *minimize inconsistent results* due to garbage collection (even if such system call cannot force a garbage collection algorithm execution).

All tests have been repeated 10 times and averaged after removing the lowest and highest value.

We measured time with `System.currentTimeMillis()` and, for finest grain computations, with `System.nanoTime()`.

7.1 Memory footprint

We start with the *space occupancy* of the implemented data structures, offering a comparison between static, append-only and adaptive append-only Elias-Fano. We will than consider the dynamic version.

Figure 20 shows the number of bits required by static and append-only Elias-Fano for the storage of s.

The number of bits required by the static version is 27 581 838 bits ≈ 3.447 MB. The *empirical minimum* of the blue curve is 27 975 853 bits ≈ 3.496 MB. This minimum has been found for a choice of $B^* = 4600$. Therefore we derive an extra storage of approximately 49.25 KB.

Now in order to validate our theoretical upper bound, we apply it to such a concrete example. For $n = 2\,348\,411$ the model suggests $B^* = 4335$, a very close value to what experimentally found. In correspondence of the empirical $B^* = 4600$ the model is predicting a maximum extra of 73.6 KB. What we experimentally found was 49.25 KB, thus the model correctly guarantees an upper bound on extra storage.

Moreover, using our tool in 20 we derive an extra percentage of approximately 2.15%. Therefore our theoretical model should have predicted a bit less[2]. In fact, 73.6 KB is 2.13% of 3.447 MB, exactly a bit less than 2.15%. Finally, our experimental extra storage should be less than these percentages: 49.25 KB corresponds, in fact, to an extra percentage of ≈ 1.428%. This is a *negligible* extra space, as claimed.

2 Remember that in equation 20 we are lower bounding $S^*(n,u)$ neglecting the $o(n)$ term.

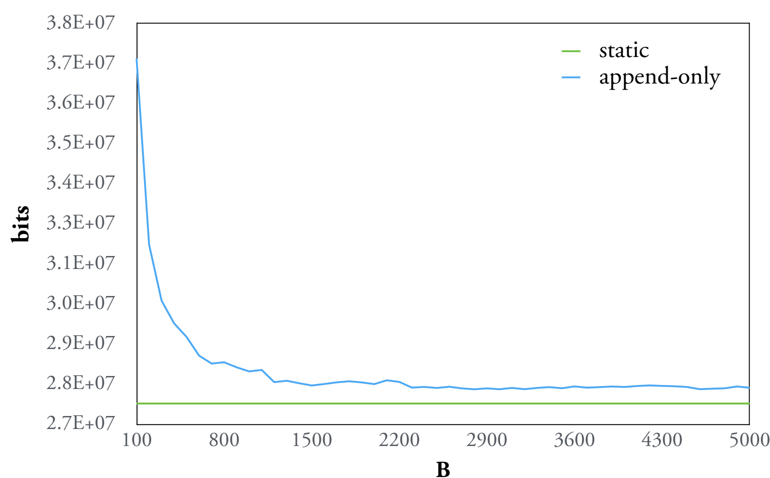

Figure 20: Bucket size against number of bits.

Flat region

Notice that there is an *almost flat* area once we passed a certain threshold (say the first 1000 values), so that users should not worry for suboptimal choices of B: they will suffer from a *negligible space degradation*. In what follows, we will use the best experimentally found value but very similar results would have been obtained for $B = 3000$ or $B = 4000$. This flat region plays a *key role* in the dynamic Elias-Fano data structure as we are going to see in Subsection 7.2.2.

Next, we show a comparison between the theoretical extra memory predicted by the model, and the one measured in practice.

As we can see form Figure 21, when B tends to the optimal experimental value, the upper bound is *very close* to, or *even matches*, real extra memory space. Being rigorous we have plotted also another curve (the green one). This is due to the *lack of implementation-specific details* in formula 15, as it should be because it is a theoretical upper bound. In particular it neglects the effect to resizing-arrays implementation which causes arrays'

dimensions grow as powers of 2. For this reason, we need to introduce a $\mathrm{cpo2}(x)$ function which returns the closest power of 2 of the argument x. This will slightly modify formula 15 in

cpo2(·) *function*

$$\mathcal{E}(n,u,B) = B \lg u + 8\,\mathrm{cpo2}(n/B) \lg u \text{ bits},$$

which is plotted in green in the above picture. This plot is *very interesting*, indeed. Firstly because, as already noticed, it clearly shows how close our model is to reality; secondly because it demonstrates that resizing-arrays' capacities have *no impact* on extra space occupancy of an append-only Elias-Fano data structure and, therefore, we can rely on formula 15.

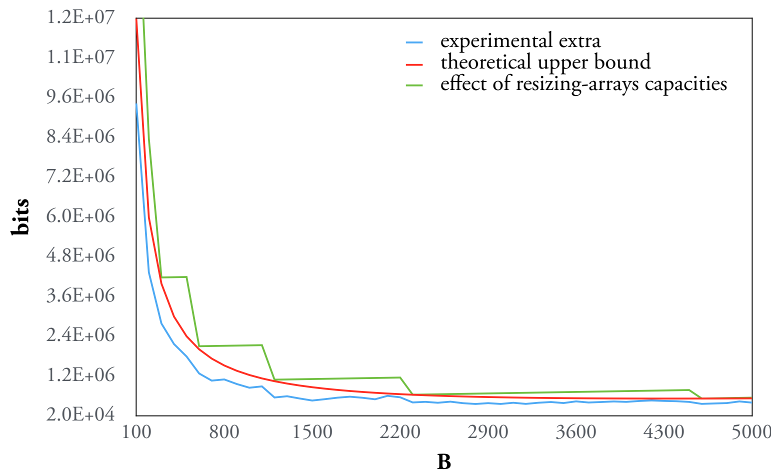

Figure 21: Extra space function compared to the experimental one.

Now we take into account our second data structure, i.e., adaptive append-only Elias-Fano. We report a comparison between the three different strategies of storing s, when the number of stored integers grows more and more.

Figure 22 shows the comparison for the first 30 000 appended integers.

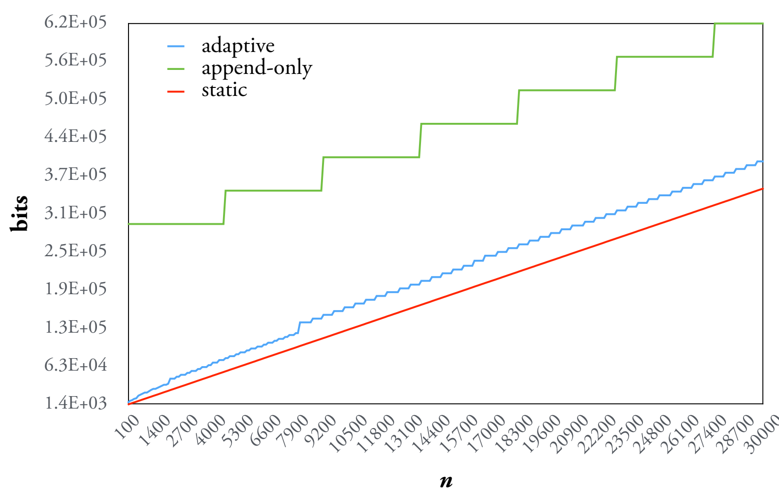

Figure 22: Appended integers against number of bits for n ranging from 100 up to 30 000 integers.

As immediate from the picture, in this case our adaptive strategy approximates much better than `append-only` the space required by a static Elias-Fano data structure. This is so since for all $n \leq B$ integers will be stored uncompressed, thus revealing in a big wasted space compared to a

static Elias-Fano representation. The bucket size B for `append-only` has been fixed to the optimal experimental value, i.e., $B = 4600$. As explained in Chapter 5, the initial choice of bucket size for the adaptive data structure has been fixed to 32.

As an example, consider $n = 10\,000$ added integers. In this case, we have already seen that `append-only` will theoretically take 73.6 KB of extra space, while experimentally we get 35.68 KB. For `adaptive`, using formula 27, we derive an upper bound of 5.346 KB of extra space. Actually we have 4.453 KB of extra, thus validating the efficacy of our theoretical upper bound. However, it worth noting that even 4.453 KB is *not negligible* in this case because the overall space of the static Elias-Fano representation takes approximately 14.7 KB.

Figure 23 shows the asymptotic behaviour of the space required by an adaptive structure. In this case, `append-only` and `adaptive` almost[3] coincide. For $\phi \approx 525\,000$ added integers, the static representation takes 0.77 MB of space. The adaptive representation takes ≈ 0.8 MB, and we derive an extra of 36.8 KB. This corresponds to approximately 4.78% extra memory, thus a negligible factor as claimed. Notice that this value is *much less* than the one predicted by the model (25%), since that value was derived assuming the worst ever possible u, i.e., $u = n$. Clearly, the more integers we add the smaller the percentage of extra memory will become, since $\lim_{n \to n_0} \mathcal{E}_1(n, u) = o(S^*(n, u))$.

3 `adaptive` is again slightly better than `append-only`.

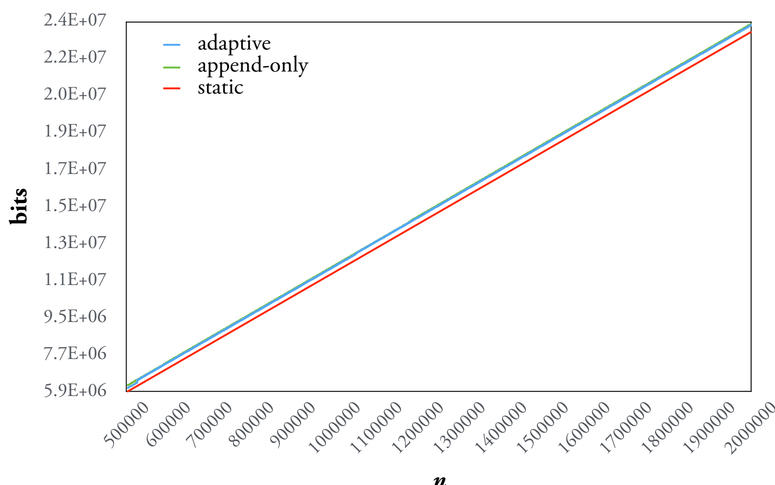

Figure 23: Appended integers against number of bits for n ranging from 500 000 up to 2 000 000 integers.

The last plot shows the behaviour of `adaptive` when we surpass the threshold n_0. Remember that from this value on, we create a new chunk, i.e., a new append-only data structure. Therefore, since its space will be added to the previous chunk space, we expect an increase in space occupancy for $n > n_0$.

This increase, in fact, clearly shows up in Figure 24 (because of the large scale). Now, by using our $\beta(\cdot)$ function defined in 36 we can evaluate the maximum theoretical extra space. In this case $j = 1$ since we have only two created chunks. We then derive an extra of 128 KB. Experimentally, for storing the whole s, `adaptive` takes 28 232 891 bits ≈ 3.529 MB. We derive an extra space of 81.38 KB, thus correctly upper bounded by the

model. The difference between `append-only` and `adaptive` in this case is only 32.13 KB, a *negligible fraction* compared to the overall space of the structure.

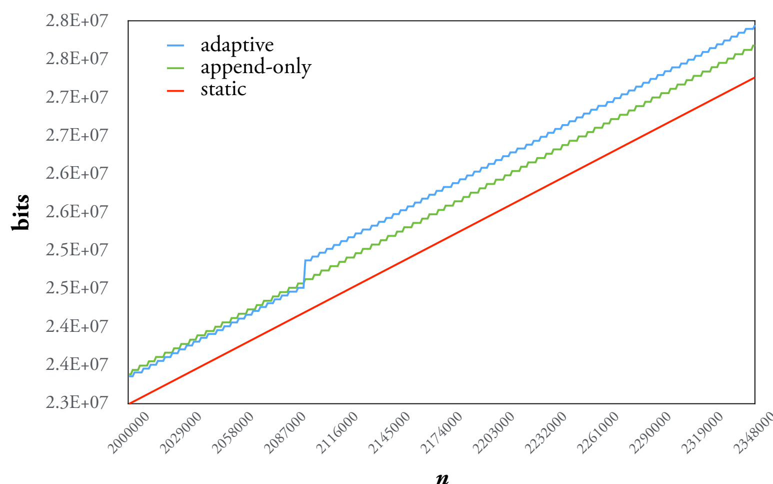

Figure 24: Appended integers against number of bits for n ranging from 2 000 000 up to 2 348 411 integers.

In conclusion `adaptive` is *always better* than `append-only` for $n \leq n_0$ and behaves *almost the same* (with negligible difference) for $n > n_0$.

We finally consider the dynamic Elias-Fano data structure. Storing s without performing any further additions (empty dynamic index) will cost 3.515 MB corresponding to a space overhead of just 0.543% and therefore negligible. On the other hand when the dynamic index is completely full, we will obtain 3.965 MB, for 13.415% overhead. These are the two extreme situations.

A typical use case will exhibit the following numbers. The experiment we propose is to perform some additions to a dynamic sequence and compare its space occupancy to the one of an append-only sequence storing the same integers. We do so because we want to validate our analysis on a loaded sequence, as it could be in a typical use case (working with no further additions will yield the very same space/time performance of `append-only`). Therefore, we add 10% integers more to s. An append-only structure will now take 3.804 MB. Performing these additions on s will cost 3.931 MB revealing approximately 3.33% space overhead. This is actually *quite small* while having a dynamically-compressed sequence.

The following table, finally, offers a schematic re-cap of the discussed quantities.

DATA STRUCTURE	MB	bpi	
`static`	3.447	11.75	
`append-only`	3.496	11.91	+ 1.428%
`adaptive`	3.529	12.02	+ 2.37%

Table 2: Space occupancies in MBs and *bpi* (bit per integer) for the storage of s.

Now adding 10% more integers, will yield the following numbers.

DATA STRUCTURE	MB	bpi	
`append-only`	3.804	11.8	
`dynamic`	3.931	12.17	+ 3.33%

Table 3: Space occupancy after 10% more additions.

7.2 TIME MEASURES

In this section we report the experimental time measurements done to evaluate the performance of the implemented `append()` and `get()` operations. As usual, we have used for the append-only structure the best experimental $B = 4600$.

7.2.1 APPEND

Mean Append Time per integer (MATpi)

Now we show a comparison between Mean Append Time per integer per element of the first two implemented structures and a simple Java `ArrayList`, for the creation of the whole sequence s (2 348 411 consecutive `append()` operations). The results are summarized by the following table.

DATA STRUCTURE	MATpi
`append-only`	161
`adaptive`	206
`ArrayList`	362

Table 4: MATpi. Shown times are in nano seconds (1 ns $= 10^{-9}$s).

As expected, append-only is the fastest, while a Java `ArrayList` should take, according to our model of Section 5.1.4.2, $161 \times 1.25 = 201.25$ ns, which is, in fact, less then the one measured.

Concerning the adaptive structure, we know that the first ϕ operations will cost something like 1.33 times more. This means that is like we are performing $2\,348\,411 + 1.33 \times 524\,288 = 3\,045\,715$ operations. In fact if we do: $3\,045\,715 \times 161$ ns / $2\,348\,411$ we approximately get 209 ns, a very close value to the one experimentally found.

7.2.2 ACCESS

Mean Query Time per integer (MQTpi)

We report here the experimental results concerning the Mean Query Time per integer for the sequence s. The experiments have been made for query patterns of 1 500 000 queries[4], generated in both random and sequential way.

The analysis is based on a comparison of mean query times for four different ways of storing s, namely:

1. static Elias-Fano;
2. append-only Elias-Fano;

4 Patterns has been generated using a simple Java program, called `QueryPatternGenerator.java`, whose example output was (for a random pattern):

`queryPatternRnd.txt`

```
1500000 ⟵ number of queries
1767221
724002
... (1 499 998 lines missing)
```

3. adaptive append-only Elias-Fano;
4. simple Java `ArrayList`.

The found results are summarized by the following two tables.

DATA STRUCTURE	sMQTpi	rMQTpi
`static`	70	100
`append-only`	87	133
`adaptive`	97	147
`ArrayList`	11	17

Table 5: Mean Query Times per integer for sequential (sMQTpi) and random (rMQTpi) accesses. Shown times are in nano seconds.

Prefetching

As usual, a sequential pattern of accesses is performed *faster* than a random one, thank to *prefetching mechanisms* of modern processors. Therefore, the analysis will focus on random accesses.

First of all, concerning the adaptive structure, we see that the difference in time from `append-only` is minimum and, therefore, negligible. *They perform practically the same.*

Time loss of append-only Elias-Fano with respect to the static version is something like 33 ns per query on average, which means `append-only` needs just 30% more than the static version. The difference is actually *very low*. We claim this is due to the fact that the compressed sequence can *almost completely* fit in the cache hierarchy (L2 cache is 3MB). Only a small portion is accessed in main memory and this results, on average, in a very small difference. For this reason we also report the behaviour of a bigger monotone sequence containing $n = 10\,445\,688$ integers whose maximum gap is again 1500.

Since now we want to focus on the analysis of cache misses, we only say this sequence takes 83.56 MB when uncompressed and 15.34 MB once compressed with a *static* Elias-Fano scheme. The redundancy factors added by append-only and adaptive structures are, respectively, 0.86 % and 1.6%. They are *negligible* as we expect.

The following table shows the time performance of *get* on such sequence.

DATA STRUCTURE	sMQTpi	rMQTpi
`static`	70	150
`append-only`	87	240
`adaptive`	97	240
`ArrayList`	11	17

Table 6: Mean Query Times per integer for sequential (sMQTpi) and random (rMQTpi) accesses on a sequence counting 10 445 688 integers. Shown times are in nano seconds.

Obviously, the sequence performs as the smaller one when accessed in a sequential manner. Also a Java `ArrayList` performs the same.

We see `append-only` and `adaptive` perform exactly the same while there is, roughly speaking, one cache miss of difference between the static version and `append-only`. We now use our model from Section 5.1.4.4 to understand this behaviour. Recall that we modelled the time complexity of a *get* operation as

$$T_{get_s} = 2\,\text{cf} + T^{*}_{get_s} + T_{select_B} - T_{select_s}. \tag{40}$$

Therefore, we need to know T_{select_B} and T_{select_s}. Measurements show that[5] $T_{select_B} \approx 55$ ns and $T_{select_s} \approx 145$ ns (those numbers are consistent with the ones reported by Vigna in [30]). Finally, plugging these values in our model, we can expect a T_{get_s} of $200 + 150 + 55 - 145$ ns $= 260$ ns. Therefore, we can immediately see how *precise* is the developed model.

5 Notice there is almost one cache miss of difference as we claimed in Section 5.1.4.4.

We consider, now, the third implemented structure. Clearly, the access time for the dynamic sequence *does not compete* with the one of the append-only structures, since we have to pay at least the binary search over buckets' sizes. As previously said, we add 10% more integers to s and we evaluate the performance on the same query pattern.

We now resort on the *powerful shape* of Figure 20. Since the access time for a dynamic Elias-Fano structure depends on bucket size B, the flat region of the curve permits us to find a reasonable trade-off between time and space performance of the data structure. Intuitively, raising B will yield a lower space, while slowing down access time. This is exactly the behaviour shown in the following two plots. We vary B between 1000 and 5000 and evaluate both space and time performance.

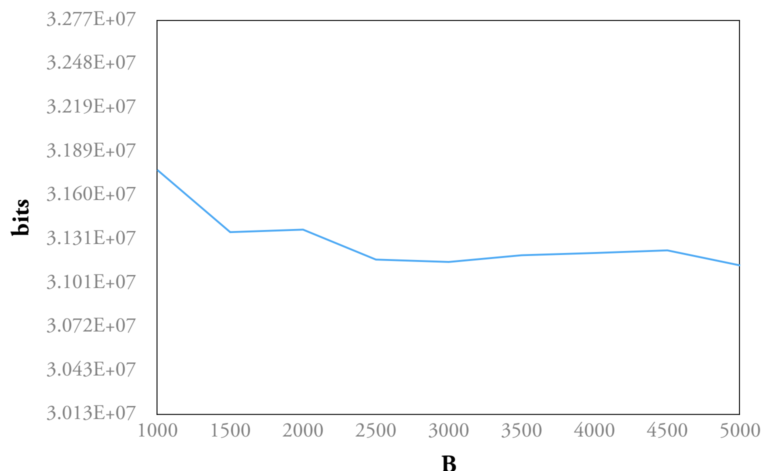

Figure 25: Bucket size against number of bits for the dynamic Elias-Fano data structure storing s. This is basically an enlargement of Figure 20.

As claimed, varying B towards the optimal value of, approximately, 5000 will decrease the number of bits, but *not so much since we are operating in the flat region*. In particular, the difference between the two extreme points is of about 0.03 MB corresponding to 1% more bits if we choose lowest $B = 1000$. This is a *very small* additional redundancy.

Now, by looking at the next plot we definitively see this small redundancy *is really worth* the trade-off.

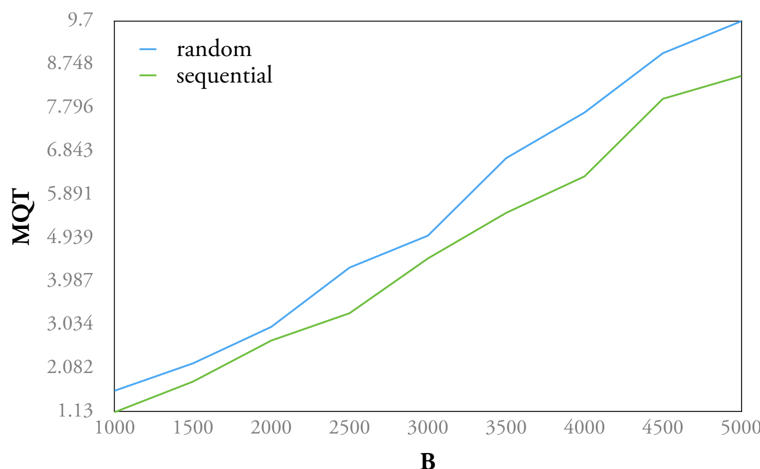

Figure 26: Bucket size against MQT in μs.

As a rule of thumb, we wish to use the smallest value of B as possible for a dynamic Elias-Fano structure. Picture 20 is clearly suggesting us that choosing B as being too small (less than 1000 as an example) will unacceptably enlarge the space required by our structure. Therefore we *should operate in the flat region*. This region will provide us the proper value of B: in this experimental example we are able to achieve an almost $\times 9$ improvement with only 1% of additional space.

In conclusion the two plots 20 and 26 should be used together to let us understand which is the value of B that achieves the right trade-off between time and space complexity of the structure.

7.2.3 Next Greater Or Equal

As claimed in Subsection 5.2.4.3, the difference in mean *nextGEQ* times of append-only and adaptive Elias-Fano is minimal or non-existent. The little overhead introduced by the dynamic structure is, instead, due to the comparisons performed during the merge-like iterating process. The shown times are, however, competitive.

The complexity of a *nextGEQ* operation is a function of bucket size too, as already noticed for the dynamic random access time. For this reason we show how its mean time varies choosing different values on B in Figure 27.

7.2.4 Iterating over the sequence

We report here the result of the optimization discussed in 5.3 concerning the way we iterate over the sequence. The following table offers a comparison between iterating using sequential *get* operations and the optimized iterating method.

DATA STRUCTURE	loop of gets	iterator
append-only	87	50
adaptive	97	50

Table 7: MQTpi when iterating over the whole sequence. Shown times are in nanoseconds.

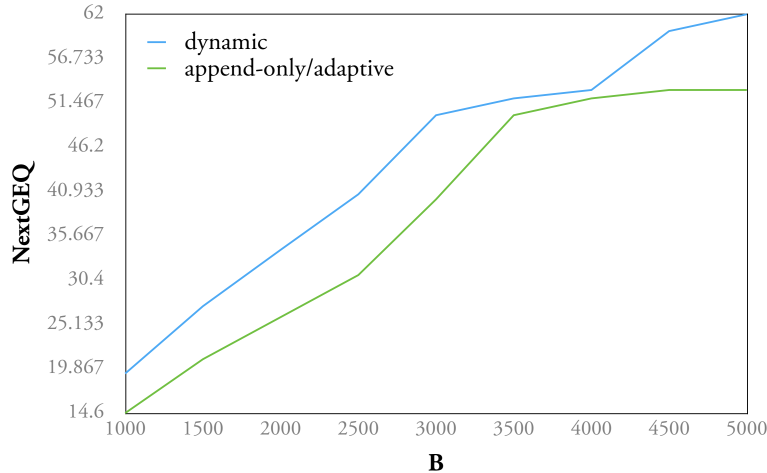

Figure 27: Bucket size against mean *nextGEQ* times in micro seconds.

As we can see, we have *almost halved* the retrieval time and the there is *no difference* between iterating over the two data structures.

The *dynamic* sequence exhibits the *same* iterating time, since integers are accessed sequentially and the costs of tests due to the merge-like iterating process introduce a negligible overhead.

7.2.5 ADDING/REMOVING

Concerning the dynamic Elias-Fano structure, we add 10% more integers to it and made sure that *all* merging/splitting cases are executed, as to obtain a general analysis about add/remove time.

The obtained results strictly confirm the theoretical complexities presented in the previous chapter. Adding/removing one element takes on average less than 10μs.

8 APPLICATIONS

We can proudly claim our implemented library will be useful for a lot and different engineering tasks. In this chapter we present three selected application benchmarks to show the practicality of our data structures.

1. For an append-only Elias-Fano data structure we show how it can be useful to succinctly represent *large static graphs* in memory, applying compression on-the-fly while reading them from disk and supporting fast access to their link structure.

2. An adaptive append-only Elias-Fano structure can be useful, instead, when building an *index for a web crawler*. A common web crawler simply downloads web pages and write them down to a large file on disk representing the crawling dataset. As an example, we may need to support exceptionally fast access to the position of the crawling collection where a specific web page begins.

3. The dynamic structure can be used in large *dynamic retrieval systems* that need to maintain huge inverted indexes. Since the indexed documents can be edited by users, than we need to support random insertions and deletions in large inverted lists. Our dynamic Elias-Fano structure is exactly designed for that purpose.

In what follows we treat these selected applications in order.

8.1 COMPRESSED IN-MEMORY GRAPHS

We start reviewing some preliminary background points. Then we explain our succinct graph representation in details.

8.1.1 PRELIMINARIES

Let $G\langle V, E\rangle$ be a graph, where V is the set of vertices and E is the set of edges. G can be directed or not. For ease of notation, let us call $n = |V|$ and $m = |E|$. Without loss of generality, we can associate to each vertex $v \in V$ an integer in $[1, \ldots, n]$, so that it is uniquely identified by that number. Basically, three ways of representing a graph link structure have been proposed in the literature. We recall them briefly because they turn out to be fundamental to properly understand our design choices. We point the reader to [28, 6, 24] for more details.

- Perhaps the most intuitive way of storing a graph link structure, is to use the *adjacency lists representation*. For each vertex $v \in V$ we store its adjacency list adj_v, i.e., the list of all the vertices pointed

out by v: $adj_v = \{u \in V : \exists(v,u) \in E\}$. The space need for the representation is $\Theta(n+m)$ or $\Theta(n+2m)$ for an undirected (symmetric) graph. Indicating with $d(v)$ the out-degree of vertex v, i.e., how many arcs depart from v, we clearly have that $d(v) = |adj_v|$. Then the time needed to retrieve all *neighbours* of v is $\Theta(1+d(v))$.

- Another way of representing a graph is by its *adjacency matrix*. This matrix $M\langle m_{ij}\rangle \in \mathbb{M}_{n\times n}$ is such that $m_{ij} = 1$ if $(i,j) \in E$ and $m_{ij} = 0$ if $(i,j) \notin E$. Storing M requires $\Theta(n^2)$ independently of whether the graph is sparse[1] or not. Retrieving all neighbours of a given node is performed in $\Theta(1+n)$. We notice a strong similarity between this representation and the one with adjacency lists. In fact, the latter representation can be seen as a *compact way* of storing the adjacency matrix M.

- There is also a third representation which achieves its best usage when the link structure is *static*. In such cases we can store all adjacency lists in an array, one after the other, and keep track of the positions from which each list begins in the array. The resulting structure is named an *adjacency array representation* of a graph. Time and space complexities are identical to the ones of the adjacency list representation.

1 Informally, a graph is said to be *sparse* if $m = O(n)$ and *dense* if $m = \Theta(n^2)$.

The adjacency list representation works well both with static and dynamic graphs, since adding a new arc is as simple as a single append operation is a proper list. Therefore we can actually use our adaptive append-only Elias-Fano data structure (or the dynamic one) as an adjacency list. In this way, we will end up with n append-only data structures. The problem here is that, when the graph is sparse (almost all real-world useful graph enjoy this property), almost every adjacency list will contain *only few integers* and there could be the risk of storing them uncompressed (large bucket size) or introducing a non-negligible space overhead. Even if the graph were dense, poor advantage will come by this solution because a smarter way of representing it is by the "complementary" graph, i.e., the structure storing the missing arcs for each node, which is expected to be small because of the density of the graph.

What we would like to have is *only one long sequence* of monotone integers. And this is possible, indeed, with the adjacency array representation for which our append-only data structure is a natural implementing candidate. The only drawback is that, in doing so, inserting a new arc in arbitrary position is a very costly operation, since we need to make room for it and theoretically rebuild the whole compressed array representation. This could be solved with our dynamic structure but we encounter another problem: how to efficiently update the whole, compressed, list of positions to adjacency lists. A single update to this list needs to be reflected to all subsequent elements, inducting a reconstruction of the whole sequence. This is too much costly. Therefore, we require the graph to be a *static* one.

So the scenario is: we already have our (large) graph structure stored in a file on disk and we read it to *incrementally* acquire its link structure in a *compressed fashion* in main memory, for querying, mining, applying further compression and so forth.

DATASETS. We have used the following three graphs of increasing size.

- `web-Stanford` is the web graph of the Stanford University, in which a node represents a page in `stanford.edu` domain and directed arcs represents hyperlinks between them. The data was collected in 2002 and is available at[2]
  http://www.cise.ufl.edu/research/sparse/matrices/SNAP/web-Stanford.html.

- `dblp` is the co-authorship network built in October 2014 from a data downloadable at http://dblp.uni-trier.de. Nodes are represented by people and an edge (u, v) exists if u and v have co-authored at least one publication.

- `ljournal-2008` is a snapshot of the LiveJournal blogging community crawled in 2008 and available at
  http://www.cise.ufl.edu/research/sparse/matrices/LAW/ljournal-2008.html.

  Nodes are people and a direct arc (u, v) exists if v is a friend of u.

2 At the same address the interested reader can find a large collection of (sparse) matrices useful for scientific analysis and described in [8]

Every graph was first preprocessed[3] to eliminate potential "holes" in vertices' identifiers so that, after remapping, every vertex id falls in $[1, \ldots, n]$. Basic statistics of the used graphs are summarized by the below table.

3 Using a Java program called `GraphMapper.java`.

GRAPH	n	m	structure
`web-Stanford`	281 903	2 312 497	unsymmetric
`dblp`	1 420 765	12 005 120	symmetric
`ljournal-2008`	5 363 201	79 023 142	unsymmetric

Table 8: Basic statistics of our datasets.

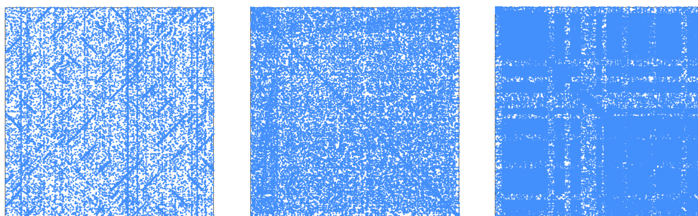

Figure 28: If we interpret an arc (u, v) as a 2D point, we can generate the following three plots. They represent the top-left corners (first 10% of vertices is shown) of `web-Stanford`, `dblp` and `ljournal-2008` respectively. These plots have been generated by a Java program called `MatrixPlotter.java` using the `JMathPlot` library available at https://code.google.com/p/jmathplot/

8.1.2 THE STRUCTURE

As already said, the graph we have to acquire in main memory is stored in a file on disk. This should have been stored in an easy-to-parse format so that any algorithmic infra structure can successfully work with it. A natural way of storing graphs is just by a sequence of arcs (u, v) such that $u, v \in [1, \ldots, n]$. This is also known as the Matrix Market (MM) Coordinate Format[4]. If vertex v does not point to any node we just read $(v, 0)$[5]. The first line of the file reports the dimension of the graph, namely the number of vertices (n) and of arcs (m). All graph datasets used in our analysis have been stored in this way. As an example, a MM graph file will look like the following one:

4 http://math.nist.gov/MatrixMarket/formats.html

5 0 cannot be interpreted as a node identifier since they go from 1 to n.

graph.txt

```
265000 1000000 ⟵ number of vertices and of arcs respectively
1 16544
1 23397
1 49821
... (27 lines skipped)
1 260675
2 69149
3 6115
3 13382
4 0
5 673
... (999 964 lines missing)
```

We maintain two append-only Elias-Fano data structures. Let us call them `positions` and `arcs` (as in our Java implementation). Now, by reading the first line of the file, we know how many elements our sequences will contain and we can compute best bucket sizes, B^*, for both of them. Then we just need to keep on reading sequentially[6] this file from disk and appending its arcs (i, j) in `arcs`. Whenever i changes to $i + 1$, we store an integer in `positions`, representing the position in `arcs` where the i-th adjacency list begins. For this task a simple counter suffices, properly initialized to 0 when we start reading the file.

6 So the reading pattern is predictable and we do not incur in any random I/O (disk seek).

The point here is that we are reading arcs (couples of integers), not a single integer to append. Therefore we have to find a way of transforming a sequence of arcs in a sequence of monotone non-decreasing integers. This is achieved considering just the "second column" of the integers in the file (the one formed by all j such that $(i, j) \in E$). Still this could not be a monotone sequence, as immediate from the above example. So we do something similar to a prefix sum. When storing the $(i + 1)$-th adjacency list, we sum to all j in adj_{i+1} the value stored at the last position of the i-th adjacency list. Clearly, at the beginning, when we have to store adj_1, we just sum 0. If the adjacency list of a vertex is empty, we just store the previous stored value, since j will be 0. Now that we have our monotone

sequence we can apply the strategy described in Chapter 5 to compress it.

Figure 29 illustrates how our data structure looks like if applied to the `graph.txt` MM file. The resulting structure have been implemented in Java and goes under the name `EliasFanoAppendOnlyGraph`.

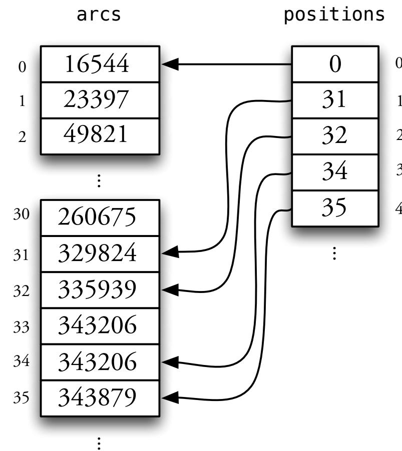

Figure 29: An example showing the adjacency array representation applied to `graph.txt`.

The query *neighbours*(v) which returns adj_v is illustrated in the following pseudo code and it runs in $O(2+d(v))$ thanks to the constant time random access of Elias-Fano encoding. Pseudo code follows.

Algorithm 12: *neighbours* pseudo code.

```
1:  procedure neighbours(v)
2:      neighbours = [];
3:      from = positions.get(v − 1);
4:      to = v == n ? arcs.size() : positions.get(v);
5:      sum = v == 1 ? 0 : arcs.get(from − 1);
6:      first = arcs.get(from) − sum;
7:      if first = 0 then
8:          return neighbours;
9:      neighbours.add(first);
10:     if to − from = 1 then
11:         return neighbours;
12:     for i in arcs from from + 1 to to − 1 do
13:         neighbours.add(i − sum);
14:     return neighbours;
```

Querying the link structure

Retrieving all the adjacency lists in *random order* is performed, on average, in 182 ns × extracted neighbour (we measure it with the very same methodology explained at the beginning of Chapter 7).

A static Elias-Fano encoding performs a bit better as expected from previously discussed results: 170 ns × extracted neighbour.

Finally, implementing the adjacency array representation with plain Java `ArrayLists` will prove to be, as usual, the fastest implementation. This

is true as long as we stay in memory, as it holds for `web-Stanford` and `dblp` graphs. In these cases, we access on average each neighbour in 97 ns. For the largest graph, i.e., `ljournal-2008`, a portion of the array is not accessed in main memory and query time $\times$ extracted neighbour is degraded to 1.52 μs.

8.1.3 Reordering of identifiers

As we notice from the previous toy example, the obtained monotone sequence of integers can grow very fast and we could end up very soon in adding huge integers. Think of the (pathological) case in which the very first vertex points to the last one, i.e., we have an arc $(1, n) \in E$. Then we will add at least n to *all* node identifiers in the adjacency list of node 2 and so on. This is dangerous for the Elias-Fano encoding scheme since it will greatly enlarge u, the maximum integer of the sequence (Chapter 4).

So we wonder if there exists a method able of avoiding this situation. Our solution consists in performing a *reordering of vertices identifiers* so that we are able to save even more space with respect to the Elias-Fano strategy we have outlined in previous section. However, remapping vertices of a graph is a well-known NP-hard problem and we can only provide *heuristic methods* trying to dominate such complexity [2].

We show that visiting the graph with a BFS (Breadth First Search) and sequentially assigning identifiers to vertices can work well with our Elias-Fano encoding strategy. A simple id-counter and a direct-address table suffice for this task. We start from the "first" vertex, namely the one of id 1 which is not remapped[7]. Then we assign id 2 to the first node in the adjacency list of 1, id 3 to the second node...up to $d(1)$. This is the best we can do for Elias-Fano, since we will produce a list of arcs like $(1,2)(1,3)(1,4)\cdots(1,d(1))$, therefore minimizing the growth of u. Here the BF order comes into play: we then consider the just remapped neighbours and we repeat the same procedure of sequentially assigning (if not already) vertices identifiers to their neighbours. This is a BFS-based reordering. Its action can be imagined as if we were "pushing" towards the bottom of the adjacency array the largest vertex ids. This can be graphically visualized as in Figure 30.

7 Or equivalently, it is always remapped in itself.

Moreover, this reordering will cause the presence of *clusters of contiguous integers*, i.e., $x, x+1, x+2, \ldots, x+k$, that are *perfect to be exploited for further compression* (compressed intervals, for example).

It is worth noting that a DFS (Depth First Search) approach would be of *no* great help instead. This is because we will assign identifiers in-depth order starting from vertex 2 and when we consider vertex 3, the id-counter could have grown *indefinitely* (at least for large graph). This will cause a potentially big gap between the last vertex id of adj_2 and the first id of adj_3. This means a greater average gap between our integers.

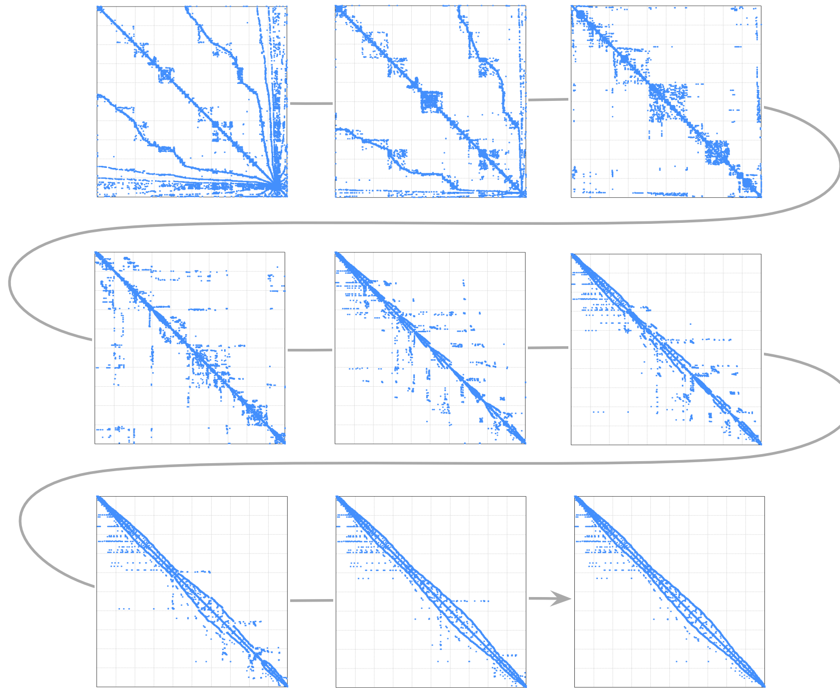

Figure 30: Graphical example of BFS reordering on a sample matrix called `hvdc1`, available at http://www.cise.ufl.edu/research/sparse/matrices/HVDC/hvdc1.html.

The following table sums up the effect of a BFS ordering on our test graphs.

	`ArrayList`		`append-only`		`static`	
	BFS ✗	BFS ✓	BFS ✗	BFS ✓	BFS ✗	BFS ✓
(a)	35.89	35.89	17.63	**15.74**	17.46	16.67
(b)	35.79	35.79	19.66	**18.21**	19.67	19.27
(c)	34.17	34.17	19.73	**19.04**	20.37	20.08

Table 9: Space occupancy expressed in *bpe* (bits per edge). (a) refers to `web-Stanford`, (b) to `dblp` and (c) to `ljournal-2008`.

We clearly see that our append-only data structure offers better compression with respect to a static Elias-Fano encoding. These are concrete examples of what we said at the end of Subsection 5.1.3.

The mean query time per extracted neighbour is improved too, since our iterator implementation of Section 5.3 will benefit from the presence of clusters of contiguous values. All adjacency lists are now retrieved, in random order and on average, in 162 ns achieving an improvement of 11% with respect to the unordered case. This result shows that, in case of reordering and thanks to our optimized iterator, append-only is competitive or even surpass the speed of a static encoding.

8.2 Building a crawling index

Web crawlers (also known as *web spiders* or *Internet bots*) are programs that systematically browse the World Wide Web (WWW) with the purpose of (typically) downloading as mush as possible web pages for later analysis. As an example, they are a core component of all modern web Search Engines (SEs).

Usually, downloaded pages are stored on large files on disks in the WARC format[8]. These files constitute the crawling data sets that are periodically inspected by web search engines for indexing the WWW.

8 http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml

SEs need fast access to web pages stored in these crawling data sets while using little space. Our adaptive append-only Elias-Fano structure can be useful for such purpose, since we do *not* know how many pages the crawler will download in a single file. The process is completely *general* toos: we parse the crawling datasets and we store in our succinct structure the position in the file at which a specific information is found. Generated positions form, of course, a monotone sequence of increasing integers.

The searched information can be *whatever we want*. As an example, the most intuitive extracted information could be storing where each HTML page begins. For these purpose we only need to parse the beginning of a web page (`<!DOCTYPE html...`) and store the position at which this string has been recognized by the parser.

We have tested this example on datasets available at

http://commoncrawl.org/the-data/

Each data segment, once unzipped, is about 3.3 GB of HTML pages. On average, each segment contains 8000 web-pages, meaning that with a sequence of 8 million integers we are able of indexing up to 330 GBs.

In this case, an interesting analysis could study how many `.get()` operation per second (reads/sec) the data structure is able to support while having a fixed number of `.add()` operation per second (writes/sec).

To this end, we have fixed 1 second as our unit of measure and count the number of read operations performed within this time. We expect a *clear trade-off*: augmenting the number of writes/sec we decrease the number of reads/sec. The following plot gives a nice idea of what explained.

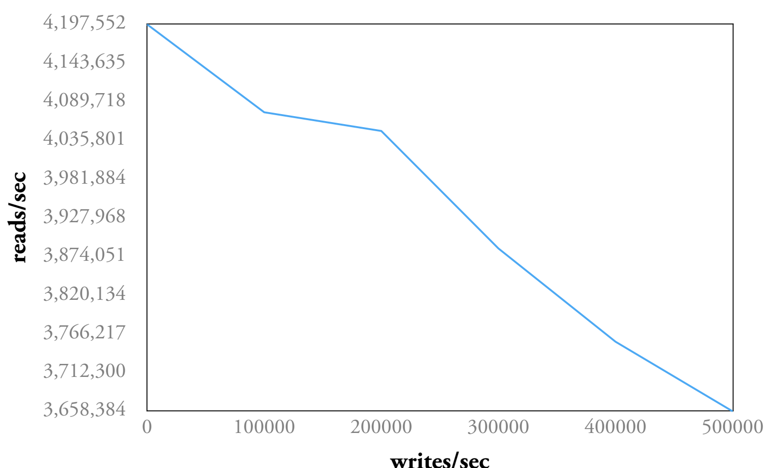

Figure 31: Number of reads per second against number of writes per second.

With 0 writes/sec we are able to achieve nearly 4.2 million reads/sec on average. Progressively increasing the rate of writes/sec of 20% will only decrease, on average, the number of reads/sec of about 2.14%.

8.3 Dynamic Inverted Lists

Inverted Indexes are the core data structure that modern search engines use to search for all documents in their corpus containing the specified user query [23]. At a high level, it basically maintains the collection of all terms appearing in the indexed corpus and for each term t the list of document identifiers (docIDs) that contain t. Such lists are called *inverted lists* or *postings-lists*.

Inverted lists

Indexing a new document in the collection means appending its docID to all postings-lists of the terms appearing in the document. The compression of these integer lists has been largely studied since the 1950s and each strategy exhibits its own space/time trade-off: usually a greater compression ratio implies a minor decoding speed and viceversa [23, 4, 1, 21, 22].

Since docIDs are assigned consecutively, postings lists are exactly monotone integer sequences as the ones described in this dissertation. The applicability of the Elias-Fano strategy to inverted indexes has been already proposed in the literature few years ago. However, achieved results concern *static* inverted indexes that are compressed to minimize their space [31, 26]. The Elias-Fano dynamic encoding we have proposed would be useful to apply *compression on-the-fly already at indexing time*. This would *highly reduce running-time memory requirements* of an inverted index.

In particular, if the collection is *not static* but documents can change over time then updating a document means adding (deleting) some of the terms appearing in it. This is naturally resolved by adding (deleting) the docID to (from) the lists of such modified terms. The dynamic Elias-Fano structure we design can, therefore, take into account these modifications.

In what follows we present a similar analysis to the one of Chapter 7 on a real-world data set in order to test the performance of our dynamic structure.

Data set. The inverted lists we use for the presented analysis are from the TREC collection Gov2, which contains nearly 25 million crawled web pages from `.gov` and `.us` sites mostly. The data was packaged by D. Lemire on April 2014. We list below the basic statistics for this data set, for more information we point the reader to [22].

Documents	Terms	Postings
24 622 347	35 636 425	5 742 630 292

Table 10: Basic statistics of Gov2.

The results we report below have been obtained for the *longest* postings-list of the data set, namely a sequence of length 13 059 642 integers to make us sure we are not operating completely in cache and validate our experimental analysis in Chapter 7.

The first picture shows how the space occupancy of such inverted list decreases while we increase bucket size. The best experimentally found value for B is 3500 in this case.

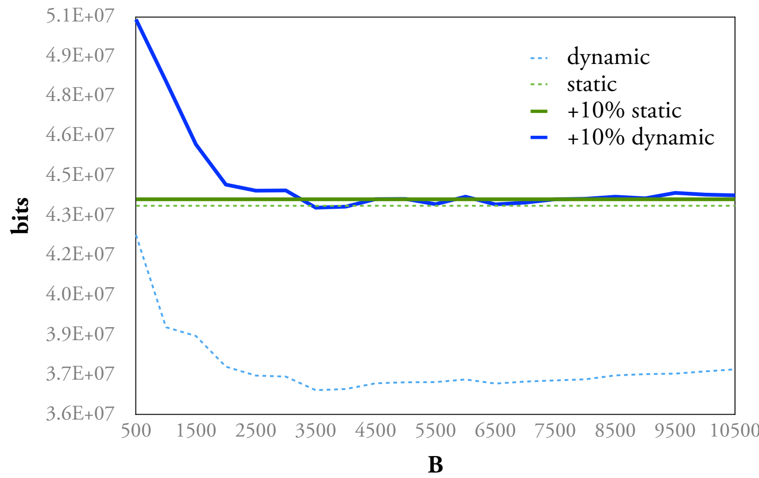

Figure 32: Bucket size against number of bits. Dotted lines illustrate the situation when no integers are added/deleted. Thicker curves take into account 10% more additions to both `static` and `dynamic`.

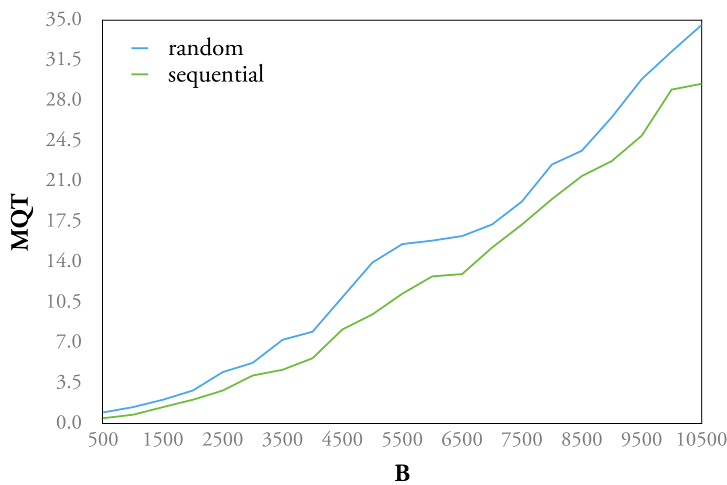

Figure 33: Bucket size against mean query times. Shown times are in μs.

Below tables offer a comparison between a static and a dynamic Elias-Fano representation of the same sequence: the first illustrates the situation in which no integers are added to nor deleted from the sequence; the second one accounts for 10% more additions.

	MB	bpi	
`static`	5.47	3.352	
`dynamic`	4.61	2.828	— 15.63%

Table 11: Comparison between `static` and `dynamic` representations when no additions/deletions are performed.

As already noticed for compressed graphs, also this example shows that a bucketing Elias-Fano strategy such as the one presented in this Thesis, can be better that a static encoding (Property 5.1.1). In this case, `dynamic` avoids a good 15.63% of the static counter part redundancy.

	MB	bpi	
static	5.5	3.13	
dynamic	5.46	3.108	— 0.71%

Table 12: Result of adding 10% more integers.

Adding 10% more integers, however, will almost annul the achieved improvement but still the space is less than the static one while having a fully dynamic sequence.

Choosing B

We consider now the sequence access time. Figure 33 reports MQTs for a pattern of random and sequential accesses as we did in Chapter 7. The plot should be used to help us deciding a good value for B to trade-off space/time complexity of the data structure. Using $B = 500$ allows us to randomly access each integer in 1μ on average, but we are using 15.16% more space. This may *not* be acceptable. A value of $B = 2000$ could be the right choice given that we access an integer in 2.8μs on average with a (negligible) space redundancy of 1.237%.

9 Conclusion and Related Works

The aim of this Thesis was to apply the Elias-Fano strategy to the compression of dynamic monotone integer sequences. Taking into account most recent results of its application [26, 7] in the context of Information Retrieval and Data Compression, we wonder if it can be extended to dynamic data structures.

For this purpose, we have presented three Elias-Fano compressed succinct data structures, showing space/time bounds and trade-offs.

The first two structures are *append-only*: new integers can only be appended to the end of the growing integer sequence. These two differs in the the way the size of a *bucket* is chosen. We call *bucket* the basic compressed span of integers that forms the fundamental building block of such structures. In the first append-only implementation it is fixed once and for all; in the second one it is *adapted* to the current dimension of the sequence with the purpose of minimizing the overall compressed space. The first structure assumes to know the future length of the sequence to evaluate best possible bucket size; otherwise an almost random choice must be performed. The other structure does not need this knowledge and can adapt to the current sequence size.

The last presented structure is, instead, a fully *dynamic* one in which no restriction is posed on the position where to add/remove an integer. This is the most *flexible* structure, yet the most *involved*. In this case we have provided a practical tool to find the proper value of bucket size to trade-off space/time complexity of the structure.

Moreover, the implemented data structures and algorithms form a library, publicly available under proper license, in the hope it will be useful for real-world applications and research.

9.1 Summary of results

We sum up here the achieved results and proofs.

1. Fundamental to this Thesis was the proof we only loose a negligible factor in both space and query time with respect to a *static* Elias-Fano compressed sequence. In particular, the first append-only structure introduce *only one cache miss more* on average performing a random access to very long (cache does not suffice) sequence. The extra memory consumption of the structure is negligible if compared to a static counter part for any real-world datasets.

2. Implementing an adaptive strategy to change bucket size, does not introduce any significant overhead on the first implemented structure. In fact, the second append-only structure *performs practically the same* as the first one, showing the care we have put in the engineering process. Again, through math proofs and large experiments, we confirm that both space and time are almost the same for the two structures for large sequences. For smaller sequences, the adaptive strategy reduces the gap in space with respect to a statically compressed sequence much better than a fixed choice of bucket size.

3. While being *not competitive* in query time with the append-only structures, the dynamic sequence introduces small extra space to maintain a dynamic index. *Iterating* through the structure is as fast as for the append-only structures. Moreover, we have also shown how to choose a proper value of bucket size to lower random access time while tolerating a bit larger redundancy.

The following table shows a complete re-cap of space/time performance of the designed structures. First row refers to the first append-only structure; second row to the second, adaptive, append-only structure; third row to the third, dynamic, structure.

As usual n represents the length of the sequence and u its maximum stored integer; $B = 2\sqrt{2n}$ is the bucket size; B_i the i-th bucket size (Subsection 5.2.1.1).

	access	*append*	*add/remove*	*nextGEQ*	space
append-only	$O(1)$	$O(1)$	—	$O(B)$	σ
adaptive	$O(1)$	$O(1)$	—	$O(B_i)$	σ
dynamic	$O(B)$	$O(1)$	$\mathcal{O}(\lg\sqrt{n})$	$O(B)$	σ

Table 13: Performance table. To distinguish between worst case and amortized running times we use symbols $O(\cdot)$ and $\mathcal{O}(\cdot)$ respectively. $\sigma = S^*(n,u) + o(n)$ bits, where $S^*(n,u) = n\left\lceil \lg\frac{u}{n}\right\rceil + 2n + o(n)$ bits.

9.2 Open problems

We would like to conclude this dissertation, illustrating some related problems the Author would be pleased to work on.

Smaller redundancy for small sequences. The redundancy introduced by the implemented data structures is *theoretically* $o(n)$ (and therefore negligible) for small integer sequences while in practice it is not so negligible because of the large constant involved in the asymptotic notation (Subsection 5.1.3.2). In particular this is due to the *buffer* of uncompressed integers that, for small sequences, constitutes the prominent space. A further work could study solutions to cope with this space overhead.

One possibility could be applying *standard compression strategies* on buffer integers, such as *gap*-encoding followed by *delta*-encoding [25]. Accessing an element from the buffer would require, in this case, to uncompress all the others up to the first[1]. This will anyway slow down access

1 Unless we adopt a bucketing strategy inside the buffer itself.

time. Therefore, guaranteeing smaller redundancy for small sequences with also small access overhead is an interesting open problem.

FURTHER REDUCTION OF ACCESS TIME. We have seen that for append-only structures the random access operation introduces *one cache miss more*, on average. Being already a good result, we wonder if there is further room for improvement to reduce this extra time up to the point it becomes *practically* negligible. Especially for the case of dynamic sequences, the access operation could be speeded up using additional information on each bucket-index. It would be great a further engineering work on the code trying to avoid this additional redundancy or, at least, minimize it.

This page intentionally left blank

Bibliography

[1] V. N. Anh, A. Moffat: *Inverted index compression using word-aligned binary codes*, Information Retrieval, 8(1): 151-166, 2005. ⇝ 95

[2] A. Apostolico, G. Drovandi: *Graph Compression by BFS*, Algorithms(2): 1031-1044, August 2009. ⇝ 92

[3] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, R. Sedgewick: *Resizable Arrays in Optimal Time and Space*, in Proceedings of the 6-th International Workshop on Algorithms and Data Structures (WADS'99), Lecture Notes in Computer Science(1663): 37-48, Vancouver, British Columbia, Canada, August 11-14, 1999. ⇝ 43

[4] M. Busch *et al.*: *Earlybird: Real-Time Search at Twitter*, In Proceedings of the 2012 IEEE, 28-th International Conference on Data Engineering: 1360-1369, 2012. ⇝ 95

[5] D. Clark: *Compact Pat Trees*, Ph.D. Thesis, University of Waterloo, Canada, 1996. ⇝ 1 and 21

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms*, Third Edition, Massachusetts Institute of Technology, The MIT Press, 2009. ⇝ 6, 7, 13, 67, 68, and 87

[7] M. Curtiss *et al.*: *Unicorn: A System for Searching the Social Graph*, VLDB, 6(11): 1150-1161, August 2013. ⇝ 2 and 98

[8] T. A. Davis, Y. Hu: *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software, November, 2010. ⇝ 89

[9] E. W. Dijkstra: *Why numbering should start at zero*, EWD 831, 1982. `https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF` ⇝ 5

[10] P. Elias: *On binary representation of monotone sequences*, Proceedings of the 6-th Princeton Conference on Information Sciences and Systems: 54-47, Princeton University, 1972. ⇝ 2 and 25

[11] P. Elias: *Efficient storage and retrieval by content and adress of static files*, J. ACM 21(2): 246-260, 1974. ⇝ 2 and 25

[12] R. M. Fano: *On the number of bits required to implement an associative memory*, Memorandum 61, Computer Structures Group, Project MAC, Massachusetts Institute of Technology, 1971. ⇝ 2 and 25

[13] M. L. Fredman, D. E. Willard: *Surpassing the Information Theoretic Bound with Fusion Tree*, Journal of Computer and System Sciences 47 (3): 424-436, 1993. ⇝ 13

[14] R. González, S. Grabowski, V. Mäkinen, G. Navarro: *Practical implementation of rank and select queries*, In Poster Proceedings Volume of 4-th Workshop on Efficient and Experimental Algorithms, WEA'05: 27-38, 2005. ⇝ 1, 20, 21, 22, 23, and 41

[15] R. Grossi, R. Raman, S. S. Rao, R. Venturini: *Dynamic Compressed Strings with Random Access*, Automata, Languages, and Programming - 40th International Colloquium, (ICALP) 2013, Riga, Latvia, Proceedings, Part I: 504-515, July 2013. ⇝ 66

[16] A. Gupta, W. K. Hon, R. Shah, J. S. Vitter: *Compressed data structures: Dictionaries and data-aware measures*, Data Compression Conference, 213-222, March 2006. ⇝ 41

[17] *Intel® SSE4 Programming Reference*, D91561-003, July 2007. ⇝ 22

[18] G. Jacobson: *Succinct Static Data Structures*, Ph.D. Thesis, CMU-CS-89-112, Carnegie Mellon University, 1989. ⇝ 1 and 18

[19] D. E. Knuth: *Structured Programming With Go To Statements*, Computing Surveys, Volume 6, Number 4, December 1974. ⇝ 15

[20] D. E. Knuth: *The Art of Computer Programming*, Volume 4, Fascicle: *Bitwise Tricks & Techniques*; Binary Decision Diagrams, Addison-Wesley Professional, 2009. ⇝ 23

[21] D. Lemire, L. Boytsov: *Decoding billions of integers per second through vectorization*, Software: Practice and Experience, 2013. ⇝ 95

[22] D. Lemire, L. Boytsov, N. Kurtz: *SIMD Compression and Intersection of Sorted Integers*, arXiv preprint arXiv:1401.6399v9 [cs.IR] 24 December, 2014. ⇝ 95

[23] C. D. Manning, P. Raghavan, H. Schütze: *An Introduction to Information Retrieval*, Cambridge University Press, Cambridge, England, 2009. ⇝ 95

[24] K. Mehlhorn, P. Sanders: *Algorithms and Data Structures - The Basic Toolbox*, Springer, March 11, 2008. ⇝ 87

[25] A. Moffat: *Compressing Integer Sequences and Sets*, Encyclopedia of Algorithms, Springer, 2009. ⇝ 25 and 99

[26] G. Ottaviano, R. Venturini: *Partitioned Elias-Fano Indexes*, Proceedings of the 37-th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR): 273-282, 2014. ⇝ 2, 35, 95, and 98

[27] A. Papoulis: *Probability, Random Variables, and Stochastic Processes*, Third Edition, Polytechnic Institute of New York, McGraw-Hill, 1991. ⇝ 12

[28] R. Sedgewick, K. Wayne: *Algorithms*, Fourth Edition, Princeton University, Addison-Wesley, 2011. ⇝ 6, 16, 32, 42, 67, 68, and 87

[29] C. E. Shannon: *A Mathematical Theory of Communication*, The Bell System Technical Journal(27): 379-423, 623-656, July, October, 1948. ⇝ 11 and 12

[30] S. Vigna: *Broadword Implementation of Rank/Select Queries*, 7-th International Workshop, WEA'08: 154-168, 2008. ⇝ 1, 2, 22, 23, 41, 57, and 84

[31] S. Vigna: *Quasi-Succinct Indices*, In Proceedings of the 6-th ACM International Conference on Web Search and Data Mining (WSDM), 2013. ⇝ 95

[32] J. S. Vitter: *External Memory Algorithms and Data Structures: Dealing with Massive Data*, ACM Computing Surveys 33(2): June 2001. Revisited on March 31, 2007. ⇝ 14

[33] D. Zhou, D. G. Andersen, V. Mäkinen, M. Kaminsky: *Space-Efficint, High-Performance Rank & Select Structures on Uncompressed Bit Sequences*, Proceedings of the 12-th International Symposium on Experimental Algorithms, SEA'13, Rome, June 2013. ⇝ 1, 22, and 23