Engineering compact dynamic data structures and in-memory data mining

Thesis submitted for the degree of
Doctor of Philosophy
at University of Leicester

by

Andreas Poyias
Department of Informatics
University of Leicester

July 2017

Engineering compact dynamic data structures and in-memory data mining

by Andreas Poyias

Abstract

Compact and succinct data structures use space that approaches the information-theoretic lower bound on the space that is required to represent the data. In practice, their memory footprint is orders of magnitude smaller than normal data structures and at the same time they are competitive in speed. A main drawback with many of these data structures is that they do not support dynamic operations efficiently. It can be exceedingly expensive to rebuild a static data structure each time an update occurs. In this thesis, we propose a number of novel compact dynamic data structures including m-Bonsai, which is a compact tree representation, compact dynamic rewritable (CDRW) arrays which is a compact representation of variable-length bit-strings. These data structures can answer queries efficiently, perform updates fast while they maintain their small memory footprint. In addition to the designing of these data structures, we analyze them theoretically, we implement them and finally test them to show their good practical performance.

Many data mining algorithms require data structures that can query and dynamically update data in memory. One such algorithm is FP-growth. It is one of the fastest algorithms for the solution of Frequent Itemset Mining, which is one of the most fundamental problems in data mining. FP-growth reads the entire data in memory, updates the data structures in memory and performs a series of queries on the given data. We propose a compact implementation for the FP-growth algorithm, the PFP-growth. Based on our experimental evaluation, our implementation is one order of magnitude more space efficient compared to the classic implementation of FP-growth and $2-3$ times compared to a more recent carefully engineered implementation. At the same time it is competitive in terms of speed.

Acknowledgments

The first acknowledgement words are undoubtedly to my PhD supervisor Rajeev Raman. Rajeev was more than just a supervisor during the last 5 and half years. It all started from a BSc dissertation project on memory management. His guidance inspired me to follow this journey to complete my PhD. He was always there willing to help and support me whenever it was necessary, reacting patiently to my "panicky" moments. I will definitely miss the late-Monday long and intense meetings that had been a huge part of my life.

A huge thank you to the remaining people of the superb "BARS" team. Simon J. Puglisi and Bella Zhukova. I feel grateful that I had the opportunity to work and leran from these people while at the same time having an amazing time. A special thank you to Simon for the great hospitality in Helsinki, and along with Rajeev for making me to get used to red wine and to actually like it.

I would like to thank the people I met in Melbourne. Initially, I would like to thank Simon Gog and Mathias Petri. During the few days I worked with them, I tried to absorb as much as I could. I would also like to thank Ian Munro for his help on the ALENEX presentation and of course all the people for the nice time we had in Melbourne.

I also want to acknowledge my brother Kyriakos. He is probably the reason I first thought of doing a PhD. He was an extra set of ears, ready to listen, discuss and brainstorm my research problems. I would like to thank my parents, George and Anastasia. I feel blessed having them as parents. Always supportive in their own way, trying to help and be next to me on every single choice of my life. A huge thank you to my grandmother Eleni, especially for the amazing food she cooked and sent me.

Moreover I would like to thank my fellow PhD researchers. I would like to thank Stelios Joannou for all the help and the C++ tips at the start of my research. Thanks especially to Gabriella, Octavian, Julien, Sam, Muz, Natasa (best marking cooperation ever), Al Sukkar, Alexey, Aram, Mohammad, George, Matt, Marco (even though Real Madrid fan), Samaneh, the Heintz siblings, Stefan. The slightly

newer generation Claudia, Mirzhan, Baraa, Sam, Pamela. Of course the amazing Rajeev team: Rafael, Yasemin, Laith and Hakeem. A huge thank you to the staff members of the department Emilio, Roy, Rick, Thomas, Reiko, Stephan, Fer-Jan, Nir, Neil.

Moving on to the needed distractions of my PhD. I would like to thank the Skliropiriniko team. A whatsApp group of 13 lifetime friends: Pourekos, Presvis, Panos, Synnos, Ermos, Pavlou, Leonidas, Mouyiasis, Kikis, Tsiakkas, Kyros, Christos, Nikos. They had been always there in their own special and unique way. By mocking each other, fighting with each other (not to mention the rest of the activities) always ending in a laughter; it offered amazing mini-breaks and it was great talking to them these 4 years. Especially, I would like to thank my house-removal friend Panos for being there for all 6 house removals I did in Leicester.

I would like to thank my friends based in Leicester for all the fun we had during my PhD. A friendship that started with "one" drink and am sure will last. I would like to thank my friend Nikos especially for the long discussions over a Starbucks iced americano. I would like to thank the Karapappas couple for all the support. Demos for all the math tips and the D&D organization, Prodromos for the basketball courses and the rest of the "Genika" group including Aristi, Katerina, Sofia, Stelios (both), George, Alexandra, Makis, Villy, Dimitris, Vassilina. A special thank you to Marica, Albert, Aggeliki, Jamie, Natalie, Constantinos, Yiota, Josephina, Emanouel, Katerina, Giannis, Terence and Carolina.

Last but not least. I would like to thank my girlfriend Eleni. Not only for her love and support, she was the person who believed in me and the person to set the target high. A smart girl that taught me how to chase my goals and I could not be more thankful.

The material of this thesis was published in the following papers:

- SPIRE'15, A. Poyias, R. Raman. Improved practical compact dynamic tries.
- ALENEX'17, A. Poyias, Simon J. Puglisi, R. Raman. Compact dynamic rewritable (CDRW) arrays.
- CoRR'17, A. Poyias, Simon J. Puglisi, R. Raman. m-Bonsai: a practical compact dynamic trie.

The material in Chapters 6 and 7 were presented in ALSIP'14.

Contents

List of Figures

Glossary

	Abbreviation	Full name
1.	SDS	Succinct Data Structures
2.	VLE	Variable Length Encoding
3.	ITLB	Information-theoretic lower bound
4.	ADT	Abstract Data Type
5.	TST	Ternary Search Tree
6.	DAT	Double Array Trie
7.	CHT	Compact Hash Table
8.	CDRW array	Compact Dynamic rewritable array
9.	m-Bonsai tree	mame-Bonsai tree
10.	FIM	Frequent Itemset Mining
11.	FP-Growth	Frequent Pattern Growth
12.	BFP-tree	Bonsai Frequent Pattern tree
13.	LFP-tree	Layered Frequent Pattern tree
14.	PFP-Growth	Piccolo Frequent Pattern Growth

Table 1: List of abbreviations used in this thesis.

Chapter 1

Introduction

"Big Data" is a term used when there are so large and complex datasets that traditional processing application softwares are unable to deal with them. Its size, is the main cause of the big data phenomenon, it describes the explosive growth of data: in 2007, more data was created than storage capacity could handle. Big Data exists as a scientific topic since 1970s, the concept gained momentum when companies invested in mining large amounts of data in order to support their decision-making [40]. Google's stated mission is to "organise the world's information and make it universally accessible and useful" [31]. This statement could not better capture people's need for gathering all kinds of data and putting them to use to improve their lives. While data storage in secondary memory seems challenging enough, to be able to perform the desired processing of these data in the main memory of computers is becoming more and more difficult. It is about 10^5 times faster to access data in main memory than on secondary memory [42], therefore operating in main memory is crucial for carrying out many data-processing applications.

1.1 In-memory data mining

Data mining refers to the process of going through the data sets to look for relevant or pertinent information. There are different examples of data that could be mined in main memory like graphs of hundreds of millions nodes, protein databases of hundreds of compounds and hundreds of genomes. To perform repeated accesses

and complex computations on such datasets, we first need to fit the data in memory. The larger the quantity of data the better as we can identify possible correlations more confidently.

1.1.1 Data mining workloads

Data mining workloads are either I/O-bound or compute-bound. This means that the time needed to execute the workload is either bound on the speed of I/O (retrieve data from disk) or it is bound on how fast the CPU can compute the retrieved data. In the first case, we compute the retrieved data "lightly", i.e. access data in memory once or twice. Since computation is much faster than I/O, there is a big disparity between data retrieval and use. For compute-bound workloads, we access data many times - often thousands - as compute-bound models are much more complex and their value improves with complexity. It is often essential that data fits in memory to be able to use data mining algorithms that require complex computations and repeated accesses on the given data [9].

1.2 Memory Hierarchy

We give an overview of the memory architecture of a computer to help understand the problems associated in representing and processing data in memory. The central processing unit (CPU) of a computer can receive instructions, decode the instructions received and perform a sequence of operations. The operations can be given from a program which may lead to access data held in memory. We have different levels in memory [20]:

- The *cache* memory is integrated directly in the CPU. It is the fastest memory, the most expensive and the smallest available. Currently computers have approximately $3 - 32$MB of cache memory.
- The *random access memory* (RAM) or *main memory* provides temporary data

storage whilst the computer is on. Main memory accesses are slower than cache memory and much faster than disk (explained below). It is less expensive than cache and bigger in capacity at a normal computer. Currently computers have about $2 - 32$GB. Main memory is connected to the CPU through a memory bus, which has a bandwidth or maximum throughput for transferring data to be processed.

- The *hard disk* provides a permanent storage of the data, even when the computer is switched off. This type of memory is the largest and the cheapest as personal computers have more than terabytes. Data here is not directly accessible by the CPU, but is first loaded (I/O) into RAM. When RAM is not enough, the hard disk operates as *virtual memory* (VM). More precisely, data is temporarily loaded from disk to RAM to be accessed and this process is considered very slow.

Most modern CPUs are so fast that for most program workloads, the difficulty is the balance of how often cache is used and how much memory transfer is done between different levels of the hierarchy. At a high level, a space efficient data structure that can fit in cache memory can make faster operations than when it needs to be accessed from RAM. On the other hand, a very space consuming data structure that can't fit in RAM may result to an "idling" CPU that spends much of its time, waiting for memory I/O to complete using the VM. This may cause *thrashing*, which is when data is repeatedly read and written back from RAM to virtual memory. When thrashing occurs, a system will slow down to an extent that it may crash.

1.3 Inefficient in-memory data representation

There are different examples of algorithms that require complex in memory accesses. Such algorithms need data structures that can fit in memory so they can perform their operations fast. For example, one of our test cases is the FP-growth algo-

rithm [34]. It is one of the fastest algorithms for the solution of Frequent Itemset Mining (FIM) [1] which will be explained in detail in Section 1.5. Algorithms like FP-Growth may comprise data structures like tries, arrays and hash tables. It is very possible that if we naively use traditional data structures to perform the queries and updates required by an algorithm, much more memory would be required than the data itself. This is called data *bloat*. For the FP-growth test case, the traditional data structure needs 40 bytes of memory (5 pointers) per item just to be able to perform the operations required by the algorithm. However, the actual information stored per item is only $2 - 3$ bytes making it space-inefficient. The resulting in-memory representation is upto 30 times more than the actual data on disk.

Another example of data bloat is on XML documents. Xerces-C++ is the standard implementation for XML DOM parser in C++. Xerces DOM parser is also using traditional data structures. It is using approximately 60 times more memory than the actual data [45].

1.4 Compact dynamic data representation

The main focus of this thesis is how to compactly represent the data in memory and at the same time efficiently perform the queries and updates required by an algorithm. Compact and succinct data structures are two types of data structures that can store data close to the minimum space bound. In practice, they use one or two orders of magnitude less memory than normal data structures, but are competitive in speed [49]. The vast majority of compact/succinct data structures are concerned largely with static data. Although the space savings is large, the main drawback to a more ubiquitous use is their notable lack of support for dynamic operations. Many applications require data structures that can index, query and dynamically update data. It can be exceedingly expensive to rebuild a static data structure from scratch each time an update occurs. The goal is then to answer queries efficiently, perform updates fast, and still maintain the small memory footprint on the

dynamically-changing data.

1.5 Frequent itemset mining (FIM)

In FIM, the data takes the form of sets of "baskets" (also called transactions) that each has a number of "items". The input data for FIM is essentially a many to many relationship between the items and the baskets. Each basket consists of a set of items (an itemset), and usually we assume that the number of items in a basket is small – smaller than the total number of items [56]. The data is assumed to be represented in a file consisting of a sequence of baskets. More precisely, FIM aims at finding regularities in the itemsets among different baskets [28]. A FIM algorithm (like FP-Growth mentioned above) is used for the solution of one of the most fundamental problems in data mining: the discovery of frequent itemsets [1].

The association rules are often used as a synonym to the market-basket analysis problem. Association rules usually come in the form of "*If A then B*". For example: 75% of those who buy cereals also buy milk; 50% of those who have high cholesterol are overweight. Therefore solving the FIM problem is a stepping stone to be able to propose some rules [48].

Applications There are numerous applications and many of them are not necessarily related with purchases and shopping [53]. Below we list some of the most common applications:

- Supermarket purchases: Associating purchased products, improve strategical placement in shelves in supermarkets.
  - Suggestion of other related products or product bundling.
- Insurance claims: unusual combinations of insurance claims can be a sign of fraud.

- Medical patient histories: certain combinations of conditions can indicate increased risk of various complications.

- Related concepts in documents: If we look for sets of words/items that appear together in many documents/buckets (e.g., Web pages, blogs, tweets), the sets will be dominated by the most common words [53]. We would hope to find among the frequent pairs some pairs of words that represent a joint concept. For example, we would expect a pair like { Messi , Barcelona } to appear rather frequently [48].

1.5.1 Challenges in frequent itemset mining

The FIM problem is one of the most studied problems in data mining. This is expected as there are a few challenges to be overcome [25]. There are different algorithms for the solution of this problem. The workload for some is I/O bound and for others is compute bound. For example, *Apriori* is considered the first efficient algorithm for finding frequent itemsets proposed by Agrawal and Srikant 1994 [1]. This is an I/O bound approach as it needs repeated scans to the data stored on disk to be able to output the frequent itemsets. For example, having frequent itemsets of size x (could be hundreds) could result to x scans of the whole dataset.

As we mentioned in Section 1.1.1, computation is much faster than I/O. In Chapter 6, we will discuss a compute bound algorithm called FP-growth [34]. This is considered among the fastest algorithms for the solution of this problem. It only needs to access the data on disk twice to load the entire data in memory. Therefore, we need to handle data efficiently: update the data structures while we read the data on disk, use compact data structures such that data fits in memory and then perform complex in-memory accesses (queries) on the given data.

1.6 Contribution

In summary, we propose a range of compact dynamic data structures, both novel and improved upon other implementations. These data structures include compact representations of tries, arrays and hash tables and we believe they can be used in practice on a variety of applications. We have detailed experimental analysis for our contributions and most of our approaches are applied on FP-growth. More precisely, our contribution includes:

- We introduce the problem of compact dynamic rewritable (CDRW) arrays. We implement different approaches of CDRW arrays and along with some heuristic optimizations they show excellent performance for both space and time.

- Not only have we (re)-confirmed that compact hash table (CHT) approach is very fast and space-efficient on modern architectures, we implemented it along with the delete operation which was not described in [11].

- We take a closer look to Bonsai, which is a compact dynamic representation of tries proposed by Darragh et al. [14]. We present improved alternative implementations of Bonsai, called m-Bonsai. This is a more practical approach as it achieves better memory usage, it is faster in practice and can be extended or traversed if needed.

- Finally, we give an efficient implementation for the FP-Growth algorithm. We propose *Piccolo* FP-Growth or PFP-Growth. Its performance is competitive in terms of speed and achieves significant reductions to the memory usage. PFP-Growth comprises a lot of ideas based on the above data structures and novel succinct data structures (SDS).

1.7 Thesis organization

The rest of the thesis is organized as follows. In Chapter 2, we give the background material of compact in-memory data representation. We describe in more detail succinct and compact data structures and the difference between them. We define the upper bound given by empirical entropy followed by different compression techniques like variable-length encoding. Finally, we show the importance of memory fragmentation and previous work on dynamic memory management.

In Chapter 3, we describe fundamental data representations and compact approaches done previously. We initially define dynamic dictionaries and then proceed with detailed analysis of hash table using open addressing with linear probing collision resolution. Then we describe compact hash table by Cleary [11] and give the implementation details of our approach. Next, we analyze previous work on standard implementations of tries. This is followed by a compact solution for the representation of tries, Bonsai, which was proposed by Dharragh et al. [14].

In Chapter 4, we present compact dynamic rewritable (CDRW) arrays. We show different approaches, that offer solutions to the problem of compactly representing rewritable arrays of bit-strings. We follow with a detailed experimental evaluation and benchmark tests for both memory and speed.

Next in Chapter 5, we give an improved implementation of Bonsai, m-Bonsai. Our experimental evaluation shows that m-Bonsai uses considerably less memory and is consistently faster than the original Bonsai.

Finally, we touch on the FP-growth algorithm. In Chapter 6, we define the FP-growth algorithm. We give standard implementations by Han et al. [34] and efficient approaches by Schlegel et al. [51]. Then we give a detailed analysis on the weaknesses of these approaches. Next in Chapter 7, we propose PFP-growth which is our approach for the implementation of the FP-growth algorithm. It tackles the weaknesses of the above algorithms, where we give a detailed experimental evaluation based on benchmarked tests.

Chapter 2

Succinct and compact data structures

In this chapter we look at different ways to deal with large volumes of data using data structures with very small memory footprint. A space efficient data representation should comply with different measures of compressibility. We give examples and show how data can be represented for the memory used to be close to the minimum space for the given data. We introduce space efficient data structures like *compressed*, *compact* and *succinct* data structures. Furthermore, we show compression techniques used for sequences of positive integers. Finally, we discuss memory management and external fragmentation.

2.1 Measures of compressibility

2.1.1 Information-theoretic lower bound (ITLB)

Suppose that we know that an object X needed to be represented is from a set of objects S. The *information-theoretic lower bound* (ITLB) to distinctly represent X is $Z = \lceil \log |S| \rceil$ bits in the worst case. Below, we give three examples of objects from different sets and their ITLB representation.

Bit-string

The first example is the representation of a bit-string. We have a bit-string X of length n. We can show that there are $|S| = 2^n$ possible bit-strings in the set S of bit

strings of length n. For example, for $n = 3$ we have 8 possible bit-strings: $S = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

Proposition 1. *The ITLB for representing a bit-string of n bits is $\lceil \log |S| \rceil = \lceil \log(2^n) \rceil = n$ bits.*

Remark - The ITLB of a bit-string shows that in the worst-case, the representation of a bit-string is actually writing the bit-string down itself.

Balanced parentheses

We are now looking at the representation of balanced parentheses. A balanced parentheses string of length $2n$ includes n opening and n closing parentheses in such a way that the parentheses are balanced. It can be shown that there are $|S| = \frac{1}{n+1}\binom{2n}{n}$ possible balanced parentheses of size $2n$. For example, for $n = 3$, we have $|S| = \frac{1}{4} * \frac{(6 * 5 * 4)}{6} = 5$ possible balanced parenthesis of size $2 * 3$: $\{((())), ()()(), (())(), ()(()), (()())\}$. Taking the $\lceil \log |S| \rceil$ we obtain:

Proposition 2. *The ITLB for a balanced parentheses representation of length $2n$ is $2n - \Theta(\log n)$ bits.*

Prefix Sums

Finally, the last example is slightly more complicated which is the prefix sums. We have a set of sequences where each sequence consists of n increasing positive integers up to m. We assume that we know n and m in advance. We can show that size of the set (the number of possible sequences) is $|S| = \binom{m-1}{n-1}$. For example, for $n = 3$ and $m = 6$, we have $s = \frac{5!}{2!3!} = 10$ possible sequences: $S = \{(1,2,6), (1,3,6), (1,4,6), (1,5,6), (2,3,6), (2,4,6), (2,5,6), (3,4,6), (3,5,6), (4,5,6)\}$. Taking the $\log |S|$, and using the inequality $\binom{m-1}{n-1} \leq (\frac{me}{n})^n$ [13] we obtain:

Proposition 3. *The information theoretic lower bound for representing a sequence of n positive integers that add up to m is $\lceil \log |S| \rceil \leq n \log(\frac{m}{n}) + n \log e$ bits.*

2.1.2 Empirical entropy

We now explain empirical entropy. Let x be a string over the alphabet $\Sigma = \{x_1, \ldots, x_\sigma\}$ and for each x_i, let $|x_i|$ be the number of occurrences of x_i in x. The probability of x_i in x is $p(i) = |x_i|/|x|$. The $0th$ order empirical entropy of the string x is $H_0(x)$ and is defined as:

$$H_0(x) = \sum_{i=1}^{\sigma} p(i) \log(1/p(i))$$

It is well known that H_0 is the minimum space bound one can achieve using a uniquely decodable code [38].

The ITLB is considered as the worst case entropy. Provided that we don't know the source/data that we are willing to compactly represent, we must comply on the worst case space representation and this is the ITLB. However, there are cases that we have some knowledge of the source/data. This may give us the freedom to push even lower than the initially calculated ITLB. For example, we have a bit-string X of size n. As we already mentioned the ITLB is n bits. Assuming that we know that bit-string has 80% ones and 20% zeros then $H_0(X) = 0.8 * \log(10/8) + 0.2 * \log(10/2) = 0.72$. Therefore, the minimum space bound to represent X is $0.72n$ bits.

2.2 word-RAM Model

Before we proceed with the explanation of succinct and compact data structures we describe the word-RAM model. The word-RAM (Random Access Machine) is a model of computation that takes into account the capacity of a computer to manipulate a word of w bits with a single instruction (for modern computers $w = 64$ bits) [33]. Each word is identified by an address. The word-RAM model also supports indirect addressing, which means a word can contain the address of another word that needs to be accessed.

We must consider a model before the analysis of any algorithm, since it defines

the costs for memory and speed. Random Access Machine (RAM) is such a model which includes a set of instructions. For example, arithmetic operations {addition, subtraction,...}, data movement{save, load operations}, control operations [13]. In the RAM model all these instructions take $O(1)$ time. The RAM model does not take into account the memory hierarchy (virtual, cache memory) used by modern machines. For example, in a normal machine there is a difference in running time when executing an instruction in main memory compared to cache. However, in the RAM model the cost of both these operations is equal. In other words, it costs one unit of computational time to access (read/write) a memory address, independent of the location of that address.

In a word-RAM model the algorithm has access to memory words which are numbered $0, 1, 2, ..., s$. The space usage at any given time is $s + 1$ where s is the highest-numbered word currently in use by the algorithm [33]. Therefore, the word-RAM model memory usage is counted by the amount of memory requested $(s+1)$ to be allocated which is not always the actual memory used [50]. This is something that needs to be considered when using dynamic data structures and when it is required to allocate and deallocate objects. We will expand more on this in Section 2.5.1.

2.3 Space-efficient data structures

There are different classes of space efficient data structures. The first class of data structures we are going to discuss is *compressed* or *opportunistic* data structures. For a sequence $X \in S$, the compressed data structures use $H_0(S) + o(\log |S|)$ bits. However, it is very hard to make these data structures dynamic therefore they are usually used to perform queries rather than updates on the represented data [42].

In this thesis we are looking closer into other sub classes of space efficient data structures like *compact* and *succinct* data structures. Let Z be the information-theoretical optimal number of bits needed to store some data. A data structure that represents this data in memory is considered succinct if it uses $Z + o(Z)$ bits and

compact if it is using $O(Z)$ bits.

For example, a data structure that uses $1.1Z$ bits of storage is considered compact and $Z + O(\frac{Z \log \log Z}{\log Z})$ bits is considered succinct. We chose these two examples specifically to show that in practice, there could be very little to no difference between compact and succinct data structures. If a compact data structure takes $(1 + \epsilon)Z$ bits, for $0 < \epsilon \leq 1$, it could be very close to a practical succinct data structure like the one above (in the example) [42].

2.3.1 Compact and succinct data structures

We now describe compact and succinct data structures. These data structures use space somewhere between traditional data structures and information theory. More precisely, when designing these data structures, one struggles with the trade-off to (1) support the desired operations as efficiently as possible and (2) increase the space as little as possible. For the same operations, usually traditional data structures require less steps than compact/succinct data structures. However, if these operations are carried out on a higher level in the memory hierarchy, the total runtime may be faster for the "smaller" representation.

Compact data structures In some lucky cases, a compact data structure reaches almost the ITLB to represent the data and provides a rich functionality like what is provided by a number of independent data structures. In general trees and text collections are probably the two most striking success stories of compact data structures (and they have been combined to store the human genome and its suffix tree in less than 4 gigabytes! [42]). In this thesis we look into a number of compact data structures therefore we don't provide different examples like we will do with succinct data structures.

Succinct data structures (SDS): Succinct, or highly-space efficient data structures were pioneered by Jacobson [35]. They encode the given data using memory

close to the ITLB while performing operations rapidly (typically in $O(1)$ time). There is a a number of succinct representation techniques, including ordinal trees, multisets, suffix trees, suffix arrays.

Bit-vector: The bit-vector data structure is a basic problem. To store a subset S of a universe $\{1, \ldots, n\}$. This is usually represented as a bit-string $x = x_1 x_2 \ldots x_n$, where each x_i is a bit. The operations to be supported are [35]:

- $\mathsf{select}_1(x,i)$: Given an index i return the location of i_{th} `1` in x.
- $\mathsf{rank}_1(x,i)$: Return the number of `1`s upto and including location i in x.

select_0 and rank_0 are defined analogously for the 0 bits in the bit-strings. We continue with an example of a bit-vector that represents bit-string x. if $x =$ `110100101` then $\mathsf{select}_1(x, 3) = 4$ since the third `1` is in location 4 (note that we count bit positions starting from 1). $\mathsf{rank}_1(x, 6) = 3$ as there are three `1`s upto and including location 6. To perform the rank_1 and select_1 operations the bit-vector space usage is $n + O(\frac{n \log\log n}{\log n})$ bits or as we already mentioned $Z + o(Z)$ bits. The operations are supported in $O(1)$ time.

Elias-Fano Coding: This example shows how to represent prefix sums using SDS.

Theorem 1. *[18, 22] Given an increasing sequence X, of n positive integers and m be the largest integer of the sequence. X can be represented using $2n + n\log(m/n) + o(n)$ bits while each integer can still be accessed in $O(1)$ time.*

Proof. We now explain how to represent X in memory. We divide each of the n integers in two parts. h_i for the *high-part* and l_i for the *low-part* of the i^{th} integer. h_i represents the $\lfloor \log n \rfloor$ high-bits of each integer, and therefore l_i represents the remaining $\lceil \log m - \log n \rceil = \lceil \log(m/n) \rceil$ low-bits. Let $H = h_1 h_2 \ldots h_n$ be a sequence of the h_i parts, and since X is a sequence of increasing integers, H is non-decreasing. We gap encode H, i.e. for $i > 1$, $h_i = h_i - h_{i-1}$. We represent the gaps using unary-codes [37]. Note that the sum of the gaps is the number of zeros, which is

$2^{\lfloor \log n \rfloor} \leq 2^{\log n}$ simplified to n. Furthermore, it is clear that we have n unary numbers (one for each integer) therefore additional n ones. Thus, the memory usage of H is $2n$ bits. To be able to access $H[i]$ we perform the $\mathsf{select}_1(H, \text{i})$ which returns the location of the i^{th} one. As we already mentioned the select_1 operation can be performed in $O(1)$ time. The remaining sequence of low-parts stored in $L = l_1 l_2 \dots l_n$ can be represented explicitly using fixed size entries of $n\lceil \log(m/n) \rceil$ bits. This makes it straight forward to access $L[i]$ in $O(1)$ time. □

We now describe the example in Figure 2.1. We have a sequence X where $n = 7$ and $m = 29$. Each integer, represented in binary format is split into high-part (red) and low-part (blue). As we mentioned above the high-part is representing the first $\lfloor \log n \rfloor = \lfloor \log 7 \rfloor = 2$ bits. We do the gap encoding of the two and store them in unary in H. Finally L shows the remaining part where each part is stored explicitly using $\lceil \log m - \log n \rceil = \lceil \log 29 - \log 7 \rceil = 3$ bits.

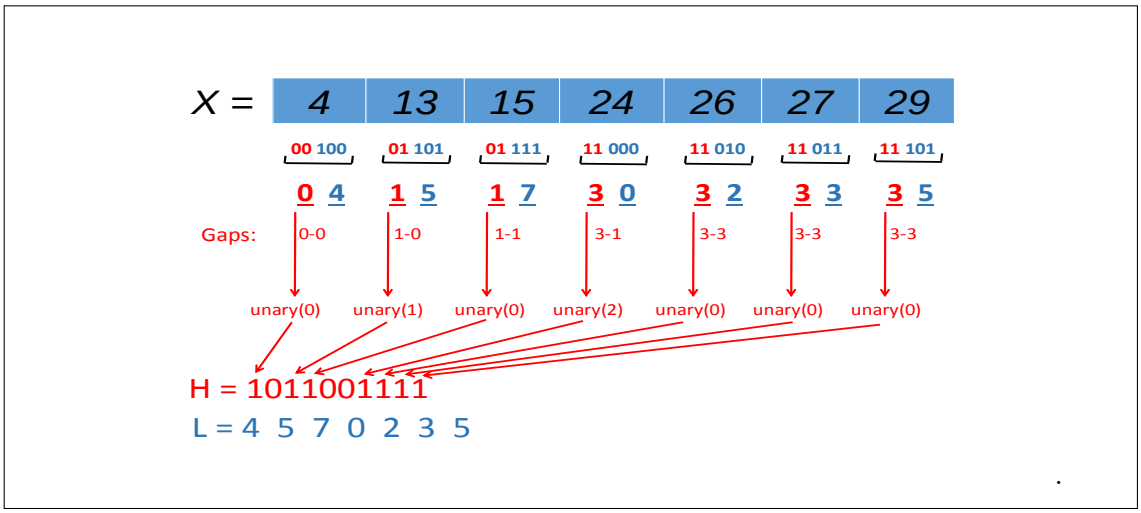


Figure 2.1: The sequence of integers X is represented using Elias-Fano encoding.

2.4 Variable-length code

Different programming languages store by default integers, characters, longs in fixed sizes in memory. For example, integers are typically stored in binary format, in

blocks of 32 bits per integer. For this example, let S be a sequence of integers that is distributed in such a way where a large percentage of them are 1s or 0s, this means they are 1-bit length in binary format [1]. Therefore, to avoid storing the "redundant" leading zeros and save space we can make use of variable-length encoding (VLE) [30]. VLE losslessly encodes S to a variable number of bits. There are different variable-length encoding (VLE) techniques [42]. Each technique performs best (nearer to the entropy) on a specific distribution of symbols X. The more suited the distribution of S is to X, the better the space savings.

2.4.1 Unique decodability

Whenever the input is coded as a variable-length sequence of bits, an important issue which arises is unique decodability. This refers to the ability to look at the compressed sequence of bits and definitively identify, break it up, or parse it into segments which represent the symbols [42].

If integers are stored next to each other in binary format without the leading zeros it would be impossible to distinguish the integers from each other. For example, if we store numbers 5, 2 and 1 (in binary they are 101, 10 and 1 respectively) without the leading zeros it would be 101101. This can be read as $5, 5$ or $2, 6, 1$ or any other possible combination, therefore binary format is not uniquely decodable.

2.4.2 VLE techniques

Throughout this thesis, we attempt to store series of variable-length numbers. We use different techniques like unary, γ and Golomb codes.

Unary codes. They represent integers $x \geq 0$ by having x 0s followed by one 1. In other words the unary encoding of integer x needs $x + 1$ bits [37]. For example, 5 is represented as 000001. Now when integer 1 needs to be encoded, it is much shorter than 5 as they are encoded using only 2 bits $\rightarrow$ 01. Therefore for the compression

[1] k-bit length integers in binary format means that their most significant bit is in the k_{th} position.

to be more effective, the probability of finding 1-values in the source should be more than the probability of finding 5s, as shown in Table 2.1.

Elias-γ codes. Also abbreviated as γ-codes [19]. This is a combination of binary and unary codes. We use unary codes to specify how long is the value in binary format, then we write the value in binary skipping the most significant bit which we know it is always 1 as the γ codes represent natural numbers ≥ 1. It is easy to verify that the length of the γ-code of the integer x is $2\lfloor \log x \rfloor + 1$ bits.

In this Thesis, when an integer x is γ-encoded we do $\gamma(x+1)$. This allows us to γ-encode zero values as well, as shown in Table 2.1. For example, if $x = 4$, we do $\gamma(4+1)$: 5 in binary is 101, we skip most significant bit therefore it is 01. The length of 01 is two, so we write two in unary (001) followed by the 01, thus the encoding is 00101. Now as shown in Table 2.1, γ-codes are scaling better than unary as numbers get larger. In fact, the only numbers that γ-codes use more bits than unary is when a value x is either 1 or 3.

Golomb-Rice code. Golomb–Rice codes [30] (abbreviated as Golomb-codes) perform based on three parameters; a pre-selected parameter m, a quotient q and remainder r for the encoding of integer x. More precisely, for an integer x we compute $q = \lfloor (x)/m \rfloor$ and $r = \lceil x \mod m \rceil$. The final encoding is the concatenation of (q) in unary and (r) where r is written in $\lceil \log m \rceil$ bits. In the example in Table 2.1 we use $m = 2$. Therefore to encode number 5, $q = 2$ and $r = 1 \rightarrow 0011$. If we choose $m = 1$, we end up having the same encoding as unary, but if we choose larger m, it is not good for small numbers as the r part will need to be $\lceil \log m \rceil$ bits. In our approaches, most numbers are 0s and 1s, therefore choosing $m > 3$ would not be ideal as it would need 3 bits to encode numbers one and zero.

The Table 2.1 shows how the first eight numbers are represented using the three variable length compression techniques explained above. We also show the frequency of each number such that if it is followed the compression technique would perform

x	Unary(x)	Unary_Fr	$\gamma(x+1)$	γ_Fr	Golomb(x)^2	Golomb_Fr
0	1	1/2	1	1/2	10	1/4
1	01	1/4	010	1/8	11	1/4
2	001	1/8	011	1/8	010	1/8
3	0001	1/16	00100	1/32	011	1/8
4	00001	1/32	00101	1/32	0010	1/16
5	000001	1/64	00110	1/32	0011	1/16
6	0000001	1/128	00111	1/32	00010	1/32
7	00000001	1/256	0001000	1/128	00011	1/32
8	000000001	1/512	0001001	1/128	000010	1/64

Table 2.1: Variable length codes and probability distribution.

best (closer to the entropy).

2.5 Memory management

2.5.1 Memory management and external fragmentation

The operating system that an algorithm is running on has its own memory allocator. The memory allocator makes use of two basic operations, the `allocate(`x`)` which allocates x contiguous memory locations and returns a pointer to the start of those [52, Ch. 6]. And the `free(`x`)` which frees these x memory locations. Usually, when the memory allocators use the `free` operation, they mark the memory as unused rather than actually delete the data in those memory locations. In the word-RAM model, memory locations allocated can be randomly accessed. However, the word-RAM model memory usage is counted by the amount of memory requested ($s + 1$ as discussed in Section 2.2) to be allocated which is not always the actual memory used [50].

As mentioned previously, let s be the contiguous memory locations allocated by an algorithm. Consider the scenario where we free x contiguous memory locations within the s space, where $x < s$ in size. This results to a gap within the s space. If we then do `allocate(`y`)` such that $y > x$, the total space requested according to word-RAM model will be $s+1+y$. However, if $y \leq x$ we can fill part of the gap and

the total space usage will be $s + 1$. This issue which causes the memory requested be bigger than the memory used (due to gaps created within s) is called external fragmentation. Note that if fixed-sized blocks are always allocated/freed then there is no issue of fragmentation.

2.5.2 Previous work: dynamic memory management

We now describe a potential solution to the problem of fragmentation. Jansson et al. [36], considered the dynamic memory management problem. We are given a number n and create n bit-strings each of initial length 0. Subsequently, we aim to support random (read-write) access to the i^{th} bit-string and also to resize the i^{th} bit string to b bits, for any $0 \leq b < w$, where w is the size of the word as explained in Section 2.2. The operations supported use $O(1)$ time:

- `address(`i`)`: Returns the pointer to where in the memory the i^{th} bit-string is stored where $(1 \leq i \leq n)$ and n is the total number of bit-strings stored.
- `realloc(`i`,`b'`)`: Changes the length of the i^{th} bit-string to b' bits. The physical address for storing the block (`address(`i`)`) may change.

Let S be the total length of all bit-strings, the space bound for Jansson et al. is $S + O(w^4 + n \log w)$ bits. While the data structure is not conceptually complex, the additive $O(n \log w)$ term increases the space bound making it hard to work in practice.

Another attempt is by Blandford et al. [8]. They considered the problem of maintaining a dynamic dictionary T of keys with associated data of bit strings that can vary in length from zero up to the length w. The dictionary uses an array $A[1...n]$ in which locations store variable-bit-length strings. The data structure used for this variable-bit-length array problem supports the operations in worst-case $O(1)$ time and uses $O(S + n)$ bits. This approach splits the memory allocation space into two parts. The first part uses the odd memory locations and the second the even ones. The concept is to allocate/free memory in one part until it gets very sparse and then

copy it more tidily in the second part. Even though in theory, this requires $O(1)$ time, it seems impractical to keep track of alternate memory locations (even/odd), avoid fragmentation issues and at the same time be competitively fast.

The final approach we are discussing is by Raman et al. [49]. The data structure is used for the problem of *extendible arrays*. Its space usage is $S + o(S)$. It allows $O(1)$ time for the read/write operations. The data structure can extend and shrink in size while keeping the blocks in equal sizes. As we explained in Section 2.5.1, this avoids the problem of fragmentation but the data structure appears to be too complex to implement in practice.

2.6 Conclusion

In this chapter we looked at compact and succinct data structures. We analysed how we can distinguish the two types of data structures. A data structure that represents this data in memory is considered succinct if it uses $Z + o(Z)$ bits and compact if it is using $O(Z)$ bits. Even though, the compact data structures are slightly more lenient with the space usage. We show with example that there are cases where compact data structure $(1.1Z)$ bits use space very similar to succinct data structures $(Z + O(\frac{Z \log\log Z}{\log Z})$ bits). They can both represent data close to the ITLB as they require additional memory $(O(\frac{Z \log\log Z}{\log Z})$ bits) to perform operations like rank_1 and select_1. We described different VLE techniques, that compress sequences of integers which will be used in latter chapters of the thesis. Finally, we discussed the problem of dynamic memory management and external fragmentation and we show previous work that considered this problem.

Chapter 3

Dynamic data structures and compact representations

In this chapter, we describe different fundamental data structures like dictionaries and tries. We give classic implementations of them where we expose their disadvantages in terms of space and speed. Furthermore, we show efficient implementations that would reduce the memory and bring it closer to the information-theoretic lower bound. More precisely, we start with the explanation of a dictionary which is one of the most important abstract data types. We continue with the description of the default implementation of a dictionary, the hash table. We analyze one of many approaches of a hash table, which is open addressing with linear probing collision resolution - as this approach is used in this thesis. Then we proceed with the explanation of the implementation of compact hash table (CHT), which is a compact approach that uses space close to the ITLB. Next, we proceed with the description of a dynamic Trie. We give standard and widely used implementations of a trie like ternary search tree (TST) and double array trie (DAT). Both implementations above require a lot of memory usage therefore we describe a compact implementation which was introduced by Dharragh et al. [14], the Bonsai tree.

3.1 Dictionaries

A dictionary is one of the most important abstract data types, formulated as follows. We are given a set S of key-value pairs $\langle x, y \rangle$, where the key x comes from a universe

$U = \{1, \ldots, u\}$ and the value y is from set $\{1, \ldots, 2^r\}$ (we will sometimes refer to the value as the satellite data of the key). Furthermore, all keys in S are distinct [49]. We wish to support the following operations:

- `insert`($\langle x, y \rangle$, S): Add the pair $\langle x, y \rangle$ to S, if S does not have x as a key.
- `find`(x, S): Given $x \in U$, if there is a pair $\langle x, y \rangle$ in S, return y, and a null value otherwise.
- `delete(`x`,` S`)`: Delete the pair (if any) of the form $\langle x, y \rangle$ from S.

Our interest, is in highly space-efficient approaches to the dictionary problem. In the worst case, a dictionary cannot use less space than the information-theoretic lower bound needed to store S. Using the terminology introduced earlier, if $|S| = n$, the information-theoretic lower bound is given by $Z(u, n, r) = \lceil \log \binom{u}{n} \rceil + nr = n \log u - n \log n + nr + O(n)$ bits (in what follows we abbreviate $Z(u, n, r)$ by Z). Following standard terminology, we refer to dictionaries that use $O(Z)$ bits as compact and those that use $Z + o(Z)$ bits as succinct.

3.1.1 Hash table

Hash table is often considered as the default implementation for a dictionary. It can support all the operations described above in $O(1)$ expected time[1]. In this thesis, we are considering the hash table implementation using open addressing and linear probing collision resolution. The classic implementation of this approach will be explained in Section 3.1.2 and Section 3.1.4 below.

3.1.2 Open addressing

We now explain open addressing which is the type of hashing that we will be using throughout this thesis. Hash tables often use an array as a storage medium. In

[1] Any reference throughout this thesis to the time complexity of a hash table (or compact hash table) would be in *expected* time.

open addressing, all elements occupy that array itself. Therefore, each array slot can either be empty or contain an element of the set of <key,value> pairs already inserted.

More precisely, when searching for an element, we systematically examine the array slots until we either find the desired element or we reach an empty slot to be asserted that "item is not found". As a result, in open addressing, the array representing the hash table must have at least one empty slot to be able to return "item is not found" if the item searched does not exist [13].

3.1.3 Collisions and simple uniform hashing

When we want to use all keys from universe of size u, we can easily initialize an array of size u where an element with key k would be stored in slot k. However, if we want to store n keys from universe of size u, where typically $u > n$ using a table T of size u could be very impractical and space inefficient.

Given that $|T| = M$, we use a hash function $h(k)$ that hashes key k and returns a value in the range $\{0, \ldots, M-1\}$. By doing this, we reduce the range of potential indices from u to M. This may cause *collisions* which is when 2 or more keys hash to the same slot i.e. $h(k) = h(k')$ where $k \neq k'$.

We now explain what is simple uniform hashing. A hash function should be able to perform under the assumption of simple uniform hashing. Simple uniform hashing is when each key is equally likely to hash to any of the M available slots in T. The hashing location must be independent of where any other key has hashed to. More precisely, for any random keys $x \neq y$, then $Pr[h(x) = h(y)] = 1/M$ [13].

3.1.4 Linear probing collision resolution

We now explain *linear probing* which is one of many collision resolution techniques for open addressing. It is considered easy to implement and fast in practice. It can perform all three operations `insert`, `delete`, and `find` in $O(1)$ time. The three

operations are explained in detail below.

Insert In the description of all operations we consider T as an array of length M that holds elements of <key, value> pairs. When we want to insert an element with key k and value v in table T, the hash function will calculate its *initial hash address*: `initAd = h`(k). We check if T[`initAd`] is empty and if it is, it means that there is no collision, thus we insert it there. However, if T[`initAd`] is not empty we *probe* to T[`initAd+1` mod M] and T[`initAd+2` mod M] and so on [2]. Finally, we find an empty location to insert the element in T[`initAd`+j mod M] where j is the number of probes we performed. In Algorithms for `insert` and `search`, we don't explain how to handle the *deleted* locations for simplicity, but it is straightforward once `delete` is explained.

Algorithm 1 Insert <key, value> pair in open addressing linear probing.

```
1: function INSERT(key, value, T)
2: int initAd ← h(key);        ▷ h calculates the initial hash address for the key.
3: while(table[initAd]!=empty)
4:     initAd ← initAd+1 mod M;
5: end while
6: T[initAd] ← getPair(key, value);
```

Search. When we want to search for an element with key k. If T[`initAd`] is empty, the search has immediately failed; we return -1 as element does not exist. Otherwise, we probe until we either find a match to return or an empty location which means item is not found.

Delete. When deleting an element with initial address `initAd`. If T[`initAd`] is empty, then the item to be deleted does not exist. Otherwise, we probe until we either find a match or empty location which again means that the element to be deleted does not exist. In case we find a match, we cannot just delete the element and

[2] We use mod M for clarity as we assume if the elements are probed at the end of T they wrap around to $T[0]$.

Algorithm 2 Search by key in open addressing linear probing.

```
1: function SEARCH(key, T)
2: int initAd ← h(key);        ▷ h calculates the initial hash address for the key.
3: while(T[initAd]!=empty)
4:     if(T[initAd].getKey() == key)
5:         return T[initAd];
6:     initAd ← initAd+1  mod M;
7: end while
8: return -1;
```

mark the location as empty; this would have a negative side-effect with the way insert and search operations work. Since insert/search operations rely on empty-marked slots to signal to stop probing, marking a slot as empty in-between a "cluster" of collisions, this will potentially render some elements as unfindable. The solution to this problem is to have two different flags for empty location: "empty" which is purely empty and "deleted" which is marked as deleted so during retrieval we know not to stop probing at the "deleted" slot. Furthermore, once the opportunity is given we can insert a new element in the "deleted" slot.

Algorithm 3 Delete element by key in open addressing linear probing.

```
1: function DELETE(key, T)
2: int initAd ← h(key);        ▷ h calculates the initial hash address for the key.
3: while(T[initAd]!=empty)
4:     if(T[initAd].getKey == key)
5:         markDeleted(initAd);
6:         break;
7:     end if
8:     initAd ← initAd+1  mod M;
9: end while
```

3.1.5 Expected number of probes in linear probing

In open addressing with linear probing collision resolution, we have a *load factor* $0 < \alpha < 1$, which defines how dense a hash table can be. Given that $|T| = M$ and n is the number of elements inserted in the hash table then $\alpha = n/M$.

Load factor α has an important role with respect to the average number of probes required by an element to travel until it lands to an empty location. The

lower the α, the lesser the number of probes, therefore the faster an element gets inserted/searched. On the other hand, the denser the hash table (higher α) the more memory is utilized, therefore less redundant space or less empty slots. Furthermore, the following theorem proved to be important in our implementations in Chapter 5:

Theorem 2. *[39] Assuming h is fully independent and uniformly random. The average number of probes, over all keys in the table, made in a successful search is $\approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$ for a load factor α.*

Finally, the following proposition will be proved in practice in Table 5.2.

Proposition 4. *[39] Assuming h is fully independent and uniformly random. We compute the probability that an element in the hash table will travel $> \ell$ probes after $n = \alpha M$ insertions. First, we calculate g_ℓ which is the probability that a run of size exactly ℓ starts at a given position:*

$$g_\ell = M^{-n}\binom{n}{\ell}(\ell+1)^{\ell-1}(M-\ell-1)^{n-\ell-1}(m-n-1)$$

From g_ℓ, we can extract the probability that an unsuccessful search encounters exactly ℓ occupied cells: $p_\ell < \sum_{i=\ell}^{n} g_i$. Therefore, the probability to have an unsuccessful search of $> \ell$ probes at a given α is $1 - \sum_{i=0}^{\ell} p_i$.

The proposition above proves to be beneficial in Chapter 5 where we estimate the number of elements that need to probe $> \ell$ times. The probability $1 - \sum_{i=0}^{\ell} p_i$ calculated above is used as upper-bound.

3.1.6 Hash table memory usage.

We recall that $|T| = M = n/\alpha$ and T stores the $< key, value >$ pairs. The memory usage for the classic approach of open addressing linear probing (explained above) is much more than the ITLB: Each key can be represented using $\lceil \log u \rceil$ bits and the value is r bits. Therefore, the total size of the T container is $O(M(\log(u) + r))$ bits which is much more than $n(\log u - \log n + r + O(1))$ bits which is the ITLB

for a dictionary. Therefore, in the next sections we explain compact hashing and an approach to come closer to the ITLB for dictionaries.

3.2 Compact hashing

3.2.1 Quotienting-based hash schemes

We now explain quotienting-based hash schemes or quotienting for short. The concept of quotienting was further studied by many people [44, 39, 11, 49, 23]. When hashing an element to a table T, the slot in which the element is hashed to gives information about the key. Quotienting is using this information to achieve better space usage. More precisely, since $|T| = M$ and universe size is u, only u/M elements can be hashed to the same slot. Hence, the slot number is $\log M$ bits, we use this information about the key, which means we can reduce the set of possible keys by a factor of M (quotients). Since we make use of $\log M$ bits of the location in T, for keys are $\log u$ bits long, the size of each element in T is only $\log u - \log M$ bits.

3.2.2 Compact hash table (CHT)

We now discuss the *compact hash table* described by Cleary [11]. CHT allows the storage of a set of n key-value pairs, where the keys are from universe $U = \{0, \ldots, u-1\}$ and the values (also referred to as *satellite data*) are from $\Sigma = \{0, \ldots, \sigma - 1\}$. Cleary's approach can support `insert`*(key, value)* and `find`*(key)* operations where the latter can return the value for the specific key.

The data structure is a hash table, and it consists of two arrays, Q and F and two bit-strings of length M, where $M = \lceil (1+\epsilon)n \rceil$ for some $\epsilon > 0$. Keys are stored in Q using open addressing and linear probing[3] and the space usage of Q is kept low by the use of quotienting. The hash function has the form $h(x) = (ax \bmod p) \bmod M$ for some prime $p > u$ (where $p \approx u$)and multiplier a, $1 \leq a \leq p-1$. Q only contains

[3] A variant, *bidirectional* probing, is used in [11], but we simplify this to linear probing.

the quotient_value $q(x) = \lfloor (ax \bmod p)/M \rfloor$ corresponding to x. Given $h(x)$ and $q(x)$, we can reconstruct x to check for membership. Quotients are only at most $\log(u/n)+O(1)$ bits long. Thus, the overall space usage is $(1+\epsilon)n(\log(u/n)+\log\sigma+O(1))$ which is very close to the information-theoretic lower bound of a dictionary explained earlier.

We now sketch collision handling in CHT. Since $h(x) = (ax \bmod p) \bmod M$, there can be collisions since many keys can have the same initial address. We keep all keys with the same initial address in consecutive locations in Q (this means that keys may be moved after they have been inserted) and use the two bit-strings to effect the mapping from a key's initial address to the position in Q containing its quotient (for details of the bit-strings see [11]).

The array F has entries of size $\lceil \log\sigma \rceil$ and the entry $F[i]$ stores the value associated with the key whose quotient is stored in $Q[i]$ (if any). This leads to the following:

Theorem 3 ([11])**.** *There is a data structure that stores a set X of n keys from $\{0,\ldots,u-1\}$ and associated satellite data from $\{0,\ldots,\sigma-1\}$ in at most $(1+\epsilon)n(\log(u/n)+\lceil\log\sigma\rceil+3)$ bits, for any constant $\epsilon > 0$, and supports insertion, deletion, membership testing of keys in X and retrieval of associated satellite data in $O(1/\epsilon)$ time under the assumption of uniform and independent hashing.*

Remark. The assumption of uniform and independent hashing, namely that for any $x \in U$, $h(x)$ is an independent and uniformly distributed random variable over $\{0,\ldots,M-1\}$, is a common one in the analysis of hashing. A difficulty with Cleary's CHT in theory is that the kind of linear multiplicative hash functions used are known to perform significantly worse than uniform independent hash functions [44]. On the other hand, it is not obvious how stronger hash function classes [17] would support quotienting. The practical performance of these linear multiplicative hash functions, however, is very much in line with what one would expect from uniform independent hashing.

Implementation of CHT

We implemented our own CHT. This data structure is used heavily in many implementations in most of the approaches of the thesis. It is implemented using C++ and it can be found on my personal github repository `https://github.com/Poyias/`. CHT constructor receives as parameters: the size of the universe; the number of items to be inserted; and the size of the values (satellite data). The Q and F arrays along with the bit-strings are implemented using `sdsl-lite` containers. The size of the quotients in Q is calculated during initialization. It supports `insert`, `find` in $O(1)$ time as explained in [11]. However, we implemented the `delete` operation that deletes elements in $O(1)$ time even though `delete` is not provided in [11]. Finally, an important limitation of CHT is that it is of fixed size since the only way to extend or shrink it is by rehashing.

3.3 Trie

The trie is a classic data structure (the name dates back to 1959 [15]) that has numerous applications in string processing. The name trie comes from its use for re**trie**val. It is an ordered tree data structure that is used to store a dynamic set of data. As shown in Figure 3.1 the prefix path of each node has an important role on the information retrieval of the current node and its descendants. For example, we need to traverse through path 4 to be able to retrieve the word "tea". At the same time "tea", "ted", and "ten" have common prefix path upto characters "t e" therefore the tree is structured in a way that they have a common path upto "t e". The ADT of a dynamic trie is:

- `create()`: Create a new empty tree.
- `getRoot()`: return the root of the current tree.
- `getChild(v, i)`: return child of node `v` with symbol `i`.

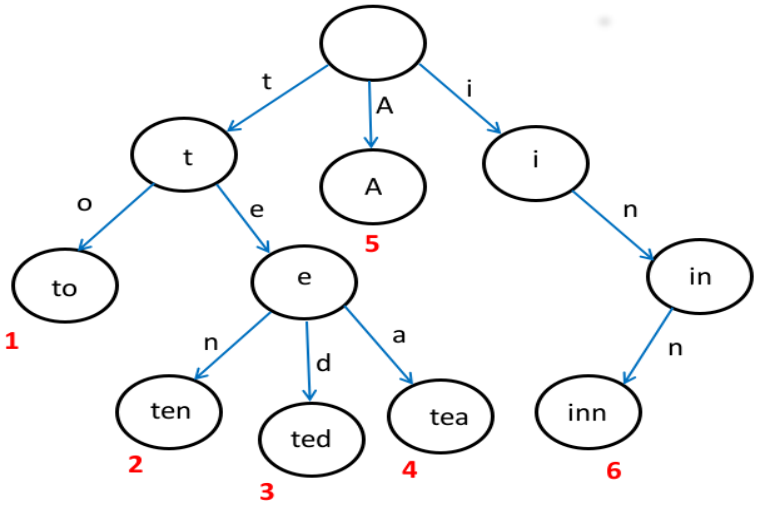


Figure 3.1: A trie for words "A", "tea","to", "ted", "ten" and "inn".

- `addChild(v, i)`: add new child with symbol `i` to node `v`.
- `getParent(v)`: return the parent of node `v`.
- `deleteChild (v, i)`: delete the child of `v` with symbol `i`.

3.3.1 Ternary Search Tree (TST)

A TST [7] is an implementation of a trie where nodes are arranged in a similar manner to a binary search tree. Each node has 4 pointers: *left* and *right* sibling, *suffix* or *child* and *parent*. In practice, in modern 64-bit machines, the space usage is 256-bits per node, or asymptotically we can say it uses $\Omega(\log n)$ bits per pointer. In addition, we need to allocate more space for the data (or satellite data) to be stored for each node. The average *insertion, lookup* and *deletion* runtime is $O(\log n)$. Like other prefix trees, a ternary search tree can be used for applications like incremental string search, spell-checking and auto-completion.

3.3.2 Double Array Trie (DAT)

DAT [2, 55] is another approach, which uses at least 2 integers per node (used as pointers), each of magnitude $O(n)$ for n-node trie. In practice, DAT shows excellent

benchmark tests compared with about ~ 30 other implementations in [54], but still there is a big distance between DAT and ITLB (explained below in more detail).

3.3.3 Minimum space for trie representation

TST and DAT approaches allocate memory of fixed size to point to a node, either by the use of pointers or or fixed size array entries. For a trie of n nodes they must asymptotically use $O(\log n)$ bits of memory. Therefore, the asymptotic space bound of TST and DAT is $O(n(\log n + \log \sigma))$ bits, where the second term is the space used for storing the node's satellite data. However, the ITLB for tries with n nodes and data with alphabet size σ is $n \log \sigma + O(n)$ bits (see e.g. [6]) corresponding to one symbol and $O(1)$ *bits* per node, so the space usage of both TST and DAT is asymptotically non-optimal.

In practice, there are many applications where $\log \sigma$ is a few bits, or one or two bytes at most. An overhead of 4 pointers per node, or $32n$ bytes (for TST) on today's machines, is what makes data *bloat* and makes it impossible to hold tries with even moderately many nodes in main memory. Although tries can be *path-compressed* by deleting nodes with just one child and storing paths explicitly, this approach (or more elaborate ones [43]) cannot guarantee a small space bound.

3.3.4 Bonsai: a compact representation of trees

We now sketch the Bonsai data structure proposed by Dharragh et al. [14]. Let M be the capacity of the Bonsai data structure, n the current number of nodes in the Bonsai data structure, and let $\epsilon = (M - n)/M$ and load factor $\alpha = (1 - \epsilon)$. The Bonsai data structure uses a CHT with capacity M, and refers to nodes via a unique *nodeID*, which is a pair $\langle i, j \rangle$ where $0 \leq i < M$ and $0 \leq j < \lambda$, where λ is an integer parameter that we discuss in greater detail below. If we wish to add a child w with symbol $c \in \Sigma$ to a node v with nodeID $\langle i, j \rangle$, then w's nodeID is obtained as follows: We create the *key* of w using the nodeID of v, which is a triple $\langle i, j, c \rangle$. We

insert the key into the CHT. Let $h : \{0, \ldots, M \cdot \lambda \cdot \sigma - 1\} \mapsto \{0, \ldots, M - 1\}$ be the hash function used in the CHT. We evaluate h on the key of w. If $i' = h(\langle i, j, c \rangle)$, the nodeID of w is $\langle i', j' \rangle$ where $j' \geq 0$ is the lowest integer such that there is no existing node with a nodeID $\langle i', j' \rangle$; i' is called the *initial address* of w.

Given the ID of a node, we can search for the key of any potential child in the CHT, which allows us to check if that child is present or not. However, to get the nodeID of the child, we need to recall that all keys with the same initial hash address in the CHT are stored consecutively. The nodeID of the child is obtained by checking its position within the set of keys with the same initial address as itself. This explains how to support the operations getChild and addChild; for getParent(v) note that the key of v encodes the nodeID of its parent.

Asymptotic space usage. In addition to the two bit-vectors of M bits each, the main space usage of the Bonsai structure is Q. Since a prime p can be found that is $< 2 \cdot M \cdot \lambda \cdot \sigma$, it follows that the values in Q are at most $\lceil \log(2\sigma\lambda + 1) \rceil$ bits. The space usage of Bonsai is therefore $M(\log \sigma + \log \lambda + O(1))$ bits.

Since the choice of the prime p depends on λ, λ must be fixed in advance. However, if more than λ keys are hashed to any value in $\{0, \ldots, M-1\}$, the algorithm is unable to continue and must spend $O((n\sigma)/\epsilon)$ time to traverse and possibly rehash the tree. Thus, λ should be chosen large enough to reduce the probability of more than λ keys hashing to the same initial address to acceptable levels. In [14] the authors, assuming the hash function satisfies the full randomness assumption argue that choosing $\lambda = O(\log M / \log \log M)$ reduces the probability of error to at most M^{-c} for any constant c (choosing asymptotically smaller λ causes the algorithm almost certainly to fail). As the optimal space usage for an n-node trie on an alphabet of size σ is $O(n \log \sigma)$ bits, the additive term of $O(M \log \lambda) = O(n \log \log n)$ makes the space usage of Bonsai non-optimal for small alphabets.

However, even this choice of λ is not well-justified from a formal perspective, since the hash function used is quite weak—it is only 2-universal [10]. For 2-universal hash

functions, the maximum number of collisions can only be bounded to $O(\sqrt{n})$ [24] (note that it is not obvious how to use more robust hash functions, since quotienting may not be possible). Choosing λ to be this large would make the space usage of the Bonsai structure asymptotically uninteresting.

Practical analysis. In practice, we note that choosing $\lambda = 16$ as suggested in [14] gives a relatively high failure probability for $M = 2^{56}$ and $\alpha = 0.8$. Choosing $\lambda = 32$, and assuming the hash function satisfies full randomness, the error probability for M up to 2^{64} is about 10^{-19} for $\alpha = 0.8$. Also, in practice, the prime p is not significantly larger than $M\lambda\sigma$ [44, Lemma 5.1]. As a result the space usage of Bonsai is typically well approximated by $M(\log \sigma + 8)$ bits for the tree sizes under consideration in this thesis (for alphabet sizes that are powers of 2, we should replace the 8 by 9).

Disadvantages

The Bonsai data structure allows a trie to be grown from empty to a theoretical maximum of M nodes, uses $M \log \sigma + O(M \log \log M)$ bits of space and performs the set of operations (explained in dynamic trie ADT) in $O(1)$ expected time. However:

- It is clear that, to perform operations in $O(1)$ expected time and use space that is not too far from optimal, n must lie between $(1 - c_1)M$ and $(1 - c_2)M$, for small constants $0 < c_2 < c_1 < 1$. The standard way to maintain the above invariant is by periodically rebuilding the trie with a new smaller or larger value of M, depending on whether n is smaller than $(1 - c_1)M$ or larger than $(1 - c_2)M$. To keep the expected (amortized) cost of rebuilding down to $O(1)$ time, the rebuilding must take $O(n)$ expected time. We are not aware of any way to achieve this without using $\Theta(n \log n)$ additional bits—an unacceptably high cost. The natural approach to rebuilding, to traverse the old tree and copy it node-by-node to the new data structure, requires the old tree to be traversed in $O(n)$ time. Unfortunately, in a Bonsai tree only root-to-leaf and leaf-to-root paths can be traversed efficiently. While a Bonsai tree can be

traversed in $O(n\sigma)$ time, this is too slow if σ is large.

- Even if the above relationship between n and M is maintained, the space is non-optimal due to the additive $O(M \log \log M)$ bits term in the space usage of Bonsai, which can dominate the remaining terms of $M(\log \sigma + O(1))$ bits when σ is small.

- The Bonsai data structure also has a certain chance of failure (in case the number of collisions get larger than λ: if it fails then the data structure may need to be rebuilt, and its not clear how to do this without affecting the space and time complexities.

- It is not clear how to support `deleteChild` in this data structure without affecting the time and space complexities (indeed, Darragh et al. [14] do not claim support for `deleteChild`). If a leaf v with nodeID $\langle i, j \rangle$ is deleted, we may not be able to handle this deletion explicitly by moving keys to close the gap, as any other keys with the same initial address *cannot be moved* without changing their nodeIDs (and hence the nodeIDs of *all* their descendants). Leaving a gap by marking the location previously occupied by v as "deleted" has the problem that the newly-vacated location can only store keys that have the same initial address i (in contrast to normal open address hashing). To the best of our knowledge, there is no analysis of the space usage of open hashing under this constraint.

- In addition, it is not obvious how to traverse an n-node tree in better than $O((n\sigma)/\epsilon)$ time. This also means that the Bonsai tree cannot be resized if n falls well below (or comes too close to) M without affecting the overall time complexity of `addChild` and `deleteChild`.

3.4 Conclusion

In this chapter, we gave a brief description of data structures that we are going to consider in the following chapters. Fundamental data structures like hash tables and dynamic tries are used frequently for a large range of problems like string verification, searching from an unpredictable set, prefix matching, autocomplete dictionary and so on [42]. In the following chapters, the main focus is memory usage and the compact implementation of such data structures. Therefore in this chapter, we provided alternative and compact approaches that already exist where we analyzed both their advantages and disadvantages.

Chapter 4

Compact dynamic rewritable (CDRW) arrays

In this chapter we introduce the problem for the implementation of compact dynamic rewritable (CDRW) arrays which are used for the compact representation of (short) bit-strings. We give 3 different implementations of CDRW arrays, which use low space usage with a low computational overhead. CDRW arrays use compact hash table to dynamically update bit-strings within. However, there is still work in progress to improve compact hash table (which is used a few implementations of CDRW arrays) to be able to extend efficiently ditto CDRW arrays.

In the chapter we proceed as follows. We start by defining the problem and we explain the need for the use of CDRW arrays by showing different applications that they could be used in. Then, we describe the related work i.e. dynamic memory management and findany dictionaries. In Section 4.3, we describe our approaches focusing on asymptotics. In Section 4.4, we proceed with the experimental evaluation. We describe the datasets used, followed by the heuristic optimizations used on our approaches. Finally, we give the implementation details for our approaches and we show the benchmark tests that we carried out.

4.1 Introduction

CDRW arrays are used for the compact representation of (short) bit-strings. We assume that the bit-strings are of size $k \leq w$ (leading zeros of a word are not stored) were w is the word size of the machine based on the word RAM model. The

operations supported are:

- $\mathsf{create}(N, k)$: Creates a new array A of size N, where each entry is of size at most k bits and equal to 0.
- $\mathsf{set}(i, v)$: Sets $A[i]$ to v, provided that v is at most k bits long.
- $\mathsf{get}(i)$ returns the value of $A[i]$.

The default solution for such data structure is to use an array of size Nk bits. Therefore, each element is of fixed length of $k \leq w$ bits per element, as the biggest bit-string would be k bits. Since w is the word size of the machine, we recall that accessing a word needs $O(1)$ time[1]. Therefore, for this approach the set and get operations are performed in $O(1)$ time. This is considered as a naive approach in terms of space usage since not all elements in the array may require k bits.

Our aim is to approach the space bound of $S(A) = \sum_{i=0}^{N-1} |A[i]|$, where $1 \leq |A[i]| \leq k$ denotes the length of a bit-string $A[i]$, while simultaneously supporting operations in $O(1)$ time. Note that $A[i]$ does not hold empty bit-strings. In addition, $S(A)$ (or just S when the context is clear) is the minimum possible space bound we can use to represent A without recoding of its elements. It is also easy to see, using simple information-theoretic arguments, that $S(A)$ is in general not achievable if A is to be represented losslessly. Nevertheless, S is an attractive "target" as it is intuitive, and can be approximated to within an $O(N) + o(S)$ additive factor. We call such a data structure a *Compact Dynamic Rewritable (CDRW)* array.

One can also consider A as containing symbols from a bounded alphabet and then aim to store S using space which is close to the *empirical* 0*-th order entropy* (explained in Section 2.1.2) of A. However, there are two difficulties with this. The notion of empirical 0-th order entropy excludes the cost for storing the alphabet, which in our case can be substantial—some of the applications we envisage involve array A containing integers from a large range. In such situations the empirical 0-th

[1]Bit-strins bigger than w bits would impact on $O(1)$ amortized time.

order entropy can lead to slightly counter-intuitive results. In addition, achieving entropy bounds in the dynamic setting requires additional ideas such as the ones by Jansson et al. and Blandford et al. explained in Section 2.5.2.

We now motivate this problem. A CDRW array can be interpreted as a fixed-length array of changeable integers — a natural data structure used in many applications. Indeed, storing a CDRW array in close to the S bound is a crucial sub-routine for achieving the entropy bounds shown in [36, 32]. Two other direct applications of CDRW arrays are counters, and spectral and counting Bloom filters [12, 21]. In addition to these motivations, the CDRW-array problem will be encountered in Chapter 5. In this chapter we use Bonsai (Section 3.3.4) variant to represent a trie. We recall that the n nodes stored in the Bonsai trie are numbered with nodeIDs $\{0, \ldots, M-1\}$ where $n < M$. In the applications below we need a CDRW array of size M were its i_{th} location holds information about the node with nodeID $= i$.

- As with the original Bonsai approach (Section 3.3.4), the variant in Chapter 5 also makes use of the compact hash table. However, the keys are inserted into and deleted from an open hash table with collision resolution using linear probing. In this application, we needed to store a CDRW array D of size M. In this array, if a key x is located in position j of the hash table, then $D[j] = j - h(x)$, where h is the hash function used. In other words, $D[j]$ records how far the key in position j is from the first location tried. During insertion of a key x, which eventually ends up (say) in position j, we needed to get the values of D in positions $i = h(x), i+1, \ldots, j-1$, and finally to set the value of $D[j]$. Provided the hash table is not too full, the value of $S(D)$ is linear in the size of the hash table. The expected maximum value is $O(\log M)$, therefore $Mk = O(M \log\log M)$.

- The trie used in Chapter 5 cannot delete any node of trie but only the leaf nodes. However, the operations supported by the trie cannot (quickly) identify if the node to be deleted is actually a leaf. A simple bit_vector that marks

the leaf nodes with 1s is not an ideal solution since once a node is deleted, we won't be able to determine if its parent is a leaf or if there are more children remaining. A CDRW array A would be able to provide the degree of a node and once a node is deleted we can update the parent's degree by decrementing it by one. In this application, since the entries of the array add up to number of nodes $(n-1)$, $S(A)$ is linear with the size of the hash table. The maximum value for the degree of the nodes is $O(\sigma)$ where σ is the alphabet size therefore $Mk = O(M \log \sigma)$ bits.

In this chapter, we give practical solutions based on compact hashing that achieve $O(1/\epsilon)$ expected time for get and set and use $(1+\epsilon)S + O(N)$ bits, for any constant $\epsilon > 0$. Experimental evaluation of our (only somewhat optimized) preliminary implementations shows excellent performance in terms of both space and time, particularly when heuristics are added to our base algorithms.

4.2 Related work.

Dynamic memory management. We recall the problem considered by Jansson et al. explained in Section 2.5.2. This is a dynamic memory management problem; given a number N and create N bit-strings each of initial length 0. The target is to support random (read-write) access to the i-th bit-string and also to resize the i-th bit string to b bits if required, for any $0 \leq b < w$. Recall that both operations `address` and `realloc` are supported in $O(1)$ time. Clearly, such a data structure is essentially the same as a CDRW array. The `address` operation returns the memory location of a bit-string (get) and `realloc` sets a bit-string of different size to a new location and returns the new location (both get and set). Recall that S is the total length of all bit-strings (which is identical with our own definition of S), the space bound is $S + O(w^4 + n \log w)$ bits, as shown in Section 2.5.2. However, the extra space $O(n \log w)$ bits makes it hard to be the basis of a good implementation for this problem. The other two approaches described in Section 2.5.2, are also targeting

the problem of dynamic memory management but prove to be too complex to be implemented in practice.

Findany dictionaries. A *c-colour findany dictionary* maintains a partition of the set $I = \{0, \ldots, n-1\}$ into a sequence of c disjoint subsets $I_0, \ldots, I_{c-1}$. Initially $I_0 = I$ and the remaining sets are empty. If $j \in I$ belongs to I_ℓ, it is convenient to say that the *colour* of j is ℓ. A findany dictionary supports the following operations:

- $\mathsf{setcolour}(j, \ell)$ – sets the colour of $j \in I$ to ℓ.
- $\mathsf{colour}(j)$ – returns the colour of $j \in I$.
- $\mathsf{enumerate}(\ell)$ – output the elements of I_ℓ (in no particular order).

The following result is implicit.

Theorem 4. *[3] A c-colour findany dictionary can be initialized in linear time, and supports* setcolour *and* colour *in* $O(1)$ *time, and* enumerate(ℓ) *in* $O(1 + |I_\ell|)$ *time, using a data structure of* $n \log c + O(nc \log c \log\log n / \log n)$ *bits.*

4.3 Approaches

All our approaches have the same general form. We let $L[0..N-1]$ be an array such that $L[i]$ is the size in bits of the value stored in $A[i]$. Letting $I_\ell = \{i \mid L[i] = \ell\}$ be the set of all indices storing ℓ-bit values, we divide A into k *layers*, one for each possible value of ℓ-bit value. For every index $i \in I_\ell$, the associated value $A[i]$ is stored in layer ℓ. A similar idea is used in the so-called alphabet partitioning technique of Barbay et al. [4] used in text indexing. Our approaches vary on exactly how L and the layers are represented.

4.3.1 Base approach.

In the base approach, we store the array L explicitly, using $N\lceil \log k \rceil$ bits. Each of the layers are represented using the data structure in Theorem 3 in Section 3.2.2

— specifically, for $i \in I_\ell$ the pair $\langle i, A[i]\rangle$ is stored in a compact hash table with i as key and $A[i]$ as satellite data. Letting $N_\ell = |I_\ell|$, the space used for layer i is therefore at most $(1+\epsilon)(N_\ell \log(N/N_\ell) + 3N_\ell + \ell N_\ell)$ bits. Summing over all layers, the overall space usage in bits is at most:

$$N\lceil \log k \rceil + (1+\epsilon)(S + 3N + \sum_{\ell=1}^{k} N_\ell \log(N/N_\ell)).$$

The final term is N times the *zero-th order empirical entropy* of the array L (denoted by $H_0(L)$). $H_0(L)$ is maximized when $N_1 = \ldots = N_k = N/k$ and $NH_0(L)$ is therefore at most $N \log k$ bits. Furthermore, $NH_0(L)$ is $o(S) + O(N)$[2]. This is because $NH_0(L)$ is (by definition) no more than any way of encoding L that encodes each entry of L separately. Hence, if we were to encode each entry of L using, say, the Elias γ-code (Section 2.4), which requires $2\lfloor \log \ell \rfloor + 1$ bits to encode an integer $\ell \geq 1$, then we know that $NH_0(L) \leq \sum_{i=1}^{N} (2 \log |A[i]| + 1) = O(N \log(S/N))$ bits. Note that if $S = O(N)$ then $N \log(S/N) = O(N)$ and if $S = \omega(N)$ then $N \log(S/N) = o(S)$. Thus, $H_0(L) = o(S) + O(N)$, and the $o(S)$ term can be absorbed in the ϵS term. The time taken for get and set is $O(1/\epsilon)$ in expectation.

The additive term of $N \log k$ bits, however, can be asymptotically larger than S. For example, if A contains N geometrically distributed random variables with mean 2, then $S = O(N)$, but $k = \Theta(\log N)$ with high probability. A simple approach is to represent L recursively. Applying recursion once, we store the lengths of L in an array L' and the actual values in a series of up to $\lceil \log k \rceil$ layers. The space usage of such a representation, in bits, is clearly at most:

$$N\lceil \log \lceil \log k \rceil \rceil + (1+\epsilon)(S + 6N + NH_0(L) + NH_0(L')),$$

and the time taken for get and set also clearly remains $O(1/\epsilon)$ time. This could, in theory, be continued to reduce the first term further. However, in practice,

[2]The asymptotic statement is in the situation where we take k to be an increasing function of N.

it is unlikely that the increase in the other terms, specifically the increase by up to $3N + NH_0(L')$ bits, will compensate for the reduction from about $n \log k$ to $n \log \log k$, so even one level of recursion is unlikely to lead to reductions in space usage.

4.3.2 Base approach with implicit lengths.

A source of wasted space in the base approach is that the length of $A[i]$ is encoded twice: in the L array and by the presence of the key i in the compact hash table for layer i. A simple solution is to omit L altogether, and replace it with a bit-string that indicates whether or not $A[i]$ has ever been set previously[3]. The space usage is $(1+\epsilon)(S + 3N + NH_0(L))$, which is $(1+\epsilon)S + O(N)$.

The complexity of the get and set operations is affected, however. A get(i) operation first checks if $A[i]$ is set, and if so it checks layers $1, \ldots, k$ until i is found, say in layer ℓ. The associated satellite data is then returned. This takes $O(\ell/\epsilon)$ time. To perform set(i, v), in effect a get(i) is performed to determine the current layer ℓ of i, upon which i is deleted from layer ℓ, and reinserted into layer ℓ' where ℓ' is the length of v. The running time is $O((\ell + \ell')/\epsilon)$ but is $O(\ell'/\epsilon)$ amortized, since the $O(\ell/\epsilon)$ term of the cost of this set can be charged to the previous set operation at index i. In summary, this approach of keeping the lengths of the $A[i]$s implicitly takes $(1+\epsilon)(S + 3N + NH_0(L)) = (1+\epsilon)S + O(N)$ bits of space, and get and set take $O(\ell/\epsilon)$ time where ℓ is the bit-length of the value returned, or given to set as an argument, respectively.

4.3.3 Dynamic function representation (DFR)

The *static function representation* problem has received interest in recent years (see [5] and references therein). We are given a universe $U = \{0, \ldots, u-1\}$ and a function $\phi : U \to \{0, \ldots, \sigma - 1\}$ that is defined only on a set $S \subseteq U$, $|S| = n$. The

[3] An alternative would be to initialize all entries of A to contain some fixed bit-string of length 1.

objective is to represent ϕ so that for any $x \in S$, $\phi(x)$ can be computed rapidly. However, for $x \in U \setminus S$, the computation may return an arbitrary answer. In particular, ϕ does not need to encode S. Thus, the first term in the space bound of $\Omega(n(\log(u/n) + \log \sigma))$ bits for storing pairs $\langle x, f(x)\rangle$ is not obviously justified and there are in fact schemes that use $O(n \log \sigma + \sigma + \log \log u)$ bits [5].

As already noted, the previous solutions to the CDRW-array problem are redundant: the length of $A[i]$ is encoded both in $L[i]$ and in the compact hash table for length-ℓ values, where $\ell = L[i]$. Recalling that $I_\ell = \{i \mid L[i] = \ell\}$, we can consider representing the function $\phi_\ell : \{0, \ldots, N-1\} \to \{0, \ldots, 2^\ell - 1\}$ where $\phi_\ell(i) = A[i]$ if $i \in I_\ell$, and is undefined otherwise. However, I_ℓ changes whenever a set operation takes place, and solutions to the static function representation problem do not work when the set is dynamic. On the other hand, in our situation, it is possible to store some additional information with an index $i \in I_\ell$.

We therefore consider the following problem, that we call *dynamic functions with hints*[4]. As in the static function representation problem, we are given a universe $U = \{0, \ldots, u-1\}$, a set $S \subseteq U$, $|S| = n$ and a function $\phi : U \to \{0, \ldots, \sigma - 1\}$. In addition to correctly evaluating $\phi(x)$ for a given argument $x \in S$, we may change ϕ either by:

- *resetting* $\phi(x) = y$ for any $x \in S$;
- *setting* $\phi(x) = y$ for some $x \notin S$ (effectively growing S to $S \cup \{x\}$) or
- *unsetting* $\phi(x)$ for some $x \in S$, i.e. making $\phi(x)$ undefined (effectively shrinking S to $S \setminus \{x\}$).

It is the "user's" responsibility to know whether an update is setting, resetting or unsetting. When setting $\phi(x)$, the data structure returns a value $v(x)$, which must be noted by the user. When evaluating, resetting or unsetting $\phi(x)$, the user must provide the previously returned value $v(x)$ along with x (and the new value of $\phi(x)$, if applicable).

[4]Dynamic functions without hints are considered in [41] and [16].

As in the classical static function representation problem, we are interested in minimizing the space required to represent ϕ as well as the time taken to evaluate $\phi(x)$. In addition, letting $V = \{v(x) \mid x \in S\}$, we also consider the space needed to store the set V. The simplest way would be to minimize the maximum value in V – this minimizes the space cost of storing the values of V in a fixed-width field. We can also consider minimizing the *average* length of a value in V, which minimizes the space cost of storing the values of V in a variable-length field.

We now give a simple solution to this problem, which has elements in common with solutions to the closely related *perfect hashing* problem, particularly that of Belazzougui et al. [5]. A slightly inelegant aspect to our solution is that if n changes "substantially", the data structure needs to be rebuilt. For the rebuilding, we require the "user" to enumerate S in linear time. We show how to accomplish this in our CDRW-array application.

The data structure consists of a bit string B of length $m = (1 + \epsilon)n$, for some constant $\epsilon > 0$, an array F of length m where each entry is $\lceil \log \sigma \rceil$ bits, and a sequence of hash functions $h_1, h_2, \ldots,$. To set $\phi(x) = y$, we compute $h_1(x), h_2(x), \ldots$ in sequence until we find a j such that $B[h_j(x)] = 0$. We then set $B[h_j(x)] = 1$ and $F[h_j(x)] = y$ and return j as $v(x)$. To evaluate $\phi(x)$, we simply return $F[h_{v(x)}(x)]$; to reset $\phi(x) = z$, we set $F[h_{v(x)}(x)] = z$ and to unset $\phi(x)$, we set $B[h_{v(x)}(x)] = 0$. It is clear that the worst-case running time for unsetting and resetting is $O(1)$, and the running time for setting the value of $\phi(x)$ is $O(v(x))$ (we are for now excluding the cost of any rebuilding). Since $v(x)$ is bounded by a geometric variable with mean $O(1/\epsilon)$, the expected value of $v(x)$, as well as the expected running time for setting, is $O(1/\epsilon)$. The maximum value of $v(x)$, and hence the maximum number of hash functions needed, is $O(\log(n/\epsilon))$ with probability at least $1 - n^{-c}$ for some constant $c \geq 0$.

An alternative is to apply hash functions only until $z = O(m/(\log \log n)^2)$ keys are left unhashed. We then store the pairs $\langle x, \phi(x) \rangle$ in a compact hash table. The size of the compact hash table is $O(z(\log(n/z) + \log \sigma))$ bits, which is $O(\log \sigma) + o(m)$

bits. The maximum value of $v(x)$ is $O(\log n/z) = O(\log\log\log n)$.

CDRW arrays using Dynamic Functions.

We again follow the base approach. This time, however, L is represented using the findany dictionary of Theorem 4. This allows us to retrieve and set an individual value of L in $O(1)$ time, and enumerate all indices in I_ℓ in time $O(|I_\ell|+1)$. The space used by this data structure is $N\log_2 k + O(Nk\log k\log\log N/\log N)$ bits. Provided that $\log k = O(\log N/(\log\log N)^2)$, this is $N\log_2 k + O(N)$ bits.

Each of the layers is represented using the dynamic function representation; the total space used by these is just $(1+\epsilon)N + o(N)$ bits. For the sake of simplicity, we do not consider the alternative where the recursion is terminated. The values v returned by the dynamic function representations are stored as a CDRW array. Since the v values are geometrically distributed, the value of S for the v values (denoted S_v in what follows) is $O(N\log(1/\epsilon))$. Storing these using the base representation with implicit lengths, we obtain a representation of v that only takes $O(S_v) = O(N\log(1/\epsilon))$ bits, and sets or gets an individual v value in $O(1)$ expected time as well. This leads to a solution that stores A using $(1+\epsilon)S + N\log k + O(N\log(1/\epsilon))$ bits, but requires $O(1)$ expected time to retrieve a given value in A. Further, the space bound only holds for $\log k = O(\log N/(\log\log N)^2)$

4.4 Experimental evaluation

We have a preliminary implementation of the above data structures in C++. All our implementations make use of the `sdsl-lite` library[5]. Arrays including L and the containers that comprise our compact hash tables are instances of `sdsl::int_vector` and `sdsl::bit_vector`. `sdsl::int_vector` is an array that allows us to specify the width of its entries and the length at the time of initialisation, whereas `sdsl::bit_vector` implements a normal bit-string over an array of 64-bit words

[5] `https://github.com/simongog/sdsl-lite`.

(wasting at most 63 bits irrespective of bit-string length).

In this section, we first describe the datasets used in our experimental evaluation and then describe three heuristic optimisations that can benefit our data structures in practice. We then fix several concrete schemes (combining approaches in previous sections with heuristics), the performance of which we measure in our experiments.

4.4.1 Datasets.

We used three datasets to show how the CDRW arrays perform under different distributions of values. The values of S and k for these datasets are given in Table 4.1.

- As noted in the introduction, one motivation for the CDRW array comes from the approach to compact hashing taken in [47]. We create a dataset that simulates insertion of $0.8N$ keys into a CHT of size N. The dataset is created as follows. We create a bit-string B of size N with every bit initialized to 0. We then repeat the following steps $0.8N$ times: *(a)* Choose a random location i from $\{0, \ldots, N-1\}$. *(b)* Inspect $B[i], B[i+1], \ldots$ until we find an index j such that $B[i+j] = 0$. We then perform $\mathsf{get}(o)$ for $o = i, i+1, \ldots, i+j$. Finally, we perform $\mathsf{set}(i+j, j)$, and also set $B[i+j] \leftarrow 1$.

  This can both be viewed as a mixed sequence of `get` and `set` operations, or by focussing just on the `set` operations, it can be viewed as a distribution over the integers that are stored in the CDRW array. We create these datasets and operations for $N = i \times 10^6$ for $i = 4, 16, 64$ and 256 and name them `hash_`i, where i refers to the number of values stored in CDRW arrays.

- Our second dataset arises from the problem of computing the degree of each node in a compact trie. The dataset is the degrees of the nodes of a trie that stores the frequent pattern mining benchmark `webdocs` [25]. `webdocs` has 231 million values, which is also the number of nodes in the trie.

- Our final datasets are synthetic and consist of 4, 16, 64 and 256 million values.

We populate A with N independent random numbers, taken from a Pareto distribution with mean 0.5. The expected maximum value and maximum length are $\Theta(N^2)$ and $\Theta(\log N)$ respectively. The Pareto distribution is a heavy-tailed distribution, in contrast to the other distributions that are biased towards smaller values. As shown in Table 4.1 a dataset of 256 million numbers with mean 0.5 can result in a distribution where the maximum value is 62 bits.

	h_4	h_16	h_64	h_256	Webdocs	p_4	p_16	p_64	p_256
S/N	1.691	1.691	1.695	1.70	1.01	3.414	3.415	3.414	3.414
k	9	9	9	10	12	50	54	58	62

Table 4.1: Values of $S = \sum_{i=0}^{N-1} |A[i]|$ and k, the maximum number of bits per key, for the datasets used, where `h_i` is `hash_i` and `p_i` is `pareto_i`.

4.4.2 Optimizations.

Many of our implementations can benefit from practical heuristic optimizations. The worst-case overhead of these implementations is small. We describe the optimizations below. Some of these optimizations (Opt 2 and 3) are tailored to the fact that the values are all more naturally interpreted as non-negative integers rather than bit-strings in our datasets.

Opt 1: Overload L.

The first optimization is to use L itself to store all values that are up to $\lceil \log k \rceil$ bits long. This potentially saves space in all layers $\ell = 1, \ldots, \lceil \log k \rceil$. The cost of this optimization is the use of N bits to indicate for each value in L whether the value stored therein indicates the layer where the value is stored or is the value itself. In addition, this optimization potentially speeds up the `get` operation since many values will be accessed directly from the array without the extra step of searching in the hash table. A similar optimization can also be used where L is not part of the data structure, namely by creating a new array to hold small values in A.

Opt 2: Omit the most significant bit (MSB).

Since layer ℓ represents values of length ℓ bits, we can avoid storing the MSB of each value. This optimization only applies for layers $\ell \geq 1$.

Opt 3: Omit inferred values.

Again this optimization is based on using the information known from L. Upon accessing $L[i]$ we know the sought item exists in the appropriate hash table according to the $L[i]$ value. Therefore, we can avoid storing the pair $\langle i, A[i] \rangle$ in the CHT for the appropriate layer for *one* value of $A[i]$ per layer. If we subsequently fail to find the array index in the layer, we can infer the value. For example, if $L[i] = 2$ we know that $A[i]$ must be a 2-bit number, and hence must be either $10_2 = 2$ or $11_2 = 3$. Hence, we simply store in the second layer all the indices that contain 2-bit values which equal 2. If an index i with $L[i] = 2$ is not found in layer 2, then its value must be 3. A potential drawback is that the searches in the CHTs will no longer all be successful searches, and unsuccessful searches can take longer.

4.4.3 Implementation details

Compact hash table (CHT).

In Section 3.2.2, we have described the implementation details of our approach for the CHT. This data structure is used heavily in the Base approaches, usually representing the layer of values of a certain length ℓ. We recall that CHT constructor receives as parameters: the size of the universe (N); the number of items to be inserted (a percentage of N); and the size of the satellite data (ℓ). The size of the quotients in Q is calculated during initialization whereas the size of satellite data is given as a parameter. As we mentioned in Section 3.2.2, an important limitation of CHT is that it is of fixed size. This affects the CDRW arrays not to be able to extend dynamically. The improvement of CHT is currently a work in progress which will allow the CDRW arrays be fully dynamic. Furthermore, we improved the CHT

to perform the `delete` operation which allows us to "realloc" values from one layer to another (if needed) in the CDRW array.

CDRW arrays.

Since we have not yet implemented dynamic resizing of our CHTs, we assume that the CDRW array constructor knows not only N and k but also the sizes of all the layers.

Benchmark baselines.

As a speed baseline for our implementations we used a normal C++ array of unsigned integers. We chose to use unsigned integers for our experiments even though Pareto datasets required $>$ 32-bits. This, if anything, favours the access time of the normal array, because it is smaller than it would need to be, and so can pontentially benefit more from CPU cache than it otherwise would.

For the space baseline, we implemented an Elias-γ approach [19]. The Elias-γ implementation has an `sdsl` container which is an `int_vector` serialised into γ-encoded values. The container is split into consecutive blocks of 256 values each. The values are stored as a concatenation of their Elias-γ codes, using `sdsl::encode` and `decode` functions. Each block needs a pointer to be accessed. To be able to `set`, or `get` the i^{th} value of a block we need to encode/decode sequentially all values up to i which leads to very high running time. Finally, in order to deal with 0 values with γ we encode i as $\gamma(i+1)$.

Base.

In this implementation L is represented as an `sdsl::int_vector`. L has a predefined width $\lceil \log k \rceil$ and all values are initialised to 0. Furthermore, Base has k layers, where each layer is a different CHT. We can subject Base to all three previously described optimizations, separately and in combination.

Base with implicit lengths (Base IL).

This is the same as Base, but omits L and replaces it with a bit-string, which is implemented using `sdsl::bit_vector`. Although there is no L to overload, we can nevertheless apply Opt 1 by having an array that stores small values in their raw form. Opt 2 can be applied to Base IL, however Opt 3 cannot be applied because when testing each CHT there is no way to know at which layer to stop, and so the values cannot be inferred.

Dynamic function representation (DFR).

We implement L as an `int_vector` and not as a c-colour findany dictionary, because we have not yet implemented dynamic resizing of our CHTs. We store a logical array F of length N, where $F[i]$ indicates the hash function that was used to store $A[i]$. F is implemented as an optimised Base IL approach. In our experiments 32 distinct hash functions were used. The sequence of hash functions are stored in a separate array, into which $F[i]$ provides us an index. If all hash functions fail then we insert the pair $\langle i, A[i] \rangle$ into an `std::map`.

4.4.4 Experiments

Setup.

The aforementioned approaches were implemented in C++. A variety of experiments was conducted to examine the performance of these implementations in terms of memory usage and runtime speed. The machine used for the experimental analysis is an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz with 3MB L2 cache, running Ubuntu 12.04.5 LTS Linux. All the code was compiled using g++ 4.7.3 with optimization level 6. To measure the resident memory (RES), `/proc/self/stat` was used. For the speed tests we measured wall clock time using `std::chrono::duration_cast`.

	Implementations	hash_256	pareto_256	webdocs
1	Base	11.61	15.37	9.21
2	Base Opt 2	10.25	13.99	9.20
4	Base Opt 1	5.42	9.92	4.05
5	Base Opt $1+2+3$	5.37	9.08	4.04
6	Base IL	7.65	10.37	5.21
7	Base IL Opt 1	4.13	8.68	2.02
8	Base IL Opt $1+2$	3.98	8.32	2.02
9	DFR	10.78	11.24	7.73
10	Elias-γ	2.76	5.46	3.14

Table 4.2: Memory usage in bits per value for each implementation on `hash_256`, `pareto_256`, `webdocs`.

Space Usage.

The memory used by our implementations is shown in Table 4.2. We observe that the heuristic optimizations often substantially reduce space usage. For example, using all three optimizations halves the memory usage of our Base implementation. In addition, there are cases for `hash_256` and `pareto_256` where we get close to the Elias-γ space baseline. The `webdocs` dataset contains much more 1s than 0s, which has an adverse effect on γ-encoding performance (since we add 1 to all values before encoding them with the Elias-γ code, all 1 values are converted to 2, whose Elias-γ code is 3 bits long). However, our approaches exploit this distribution better, and optimised Base IL takes less than half the space of γ. Finally, we note the compressibility of Pareto distribution with $k = 62$ and the effectiveness of our implementations on such a heavy-tailed distribution.

Runtime.

We ran a series of tests to measure the run time of `set` or `get` operations (reported in nanoseconds). For clarity we do not show Opt 2 combined with Opt 3 as this combination gave a negligible improvement in runtime. The runtimes of the baseline Elias-γ based implementations are not shown because they are nearly an order of magnitude larger than even our slowest implementations for random `set` opera-

tions, and still significantly slower than our slowest implementations for random get operations. All reported runtimes are the average of 10 runs.

Linear probing simulation, set operations.

In this experiment we focus on the set operation for the linear probing simulation. We consider the dataset `hash_`i described in Section 4.4.1 as a distribution of values in the CDRW array. Specifically, we perform the simulation of insertions into a CHT as described in Section 4.4.1 thus only perform the set. Fig. 4.1 shows the average set time for the `hash_`i datasets. By reducing space, there are less cache misses and for example, on `hash_4` the optimized approaches achieve speeds very close to a normal array, and remain competitive for the bigger datasets. Opt 1 has a dramatic effect on both Base and Base IL.

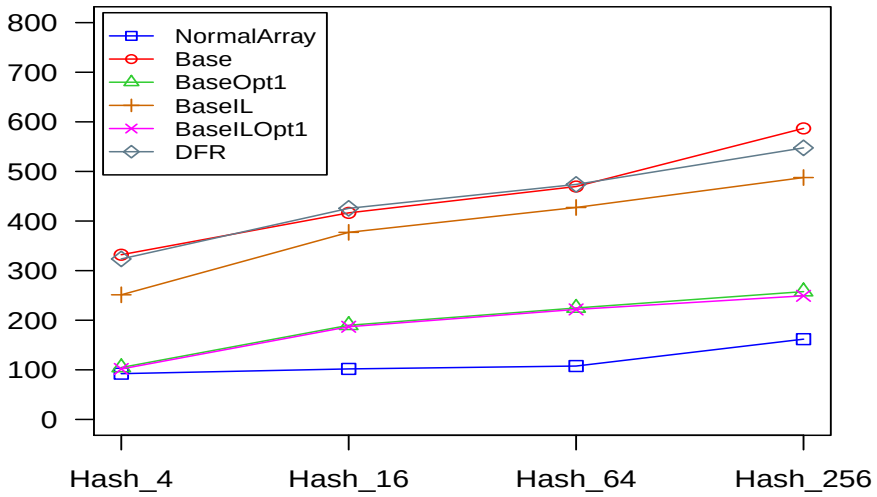


Figure 4.1: Runtimes per set operation in nanoseconds for all `hash_n` datasets.

Linear probing simulation, set and get operations.

We consider the dataset `hash_`i described in Section 4.4.1 as a sequence of intermixed set and get operations. The results can be seen in Fig. 4.2. The Base Opt 1 implementation is again very competitive with the normal array: < 2 times slower. The unoptimized Base IL was very slow and so was omitted from the graph.

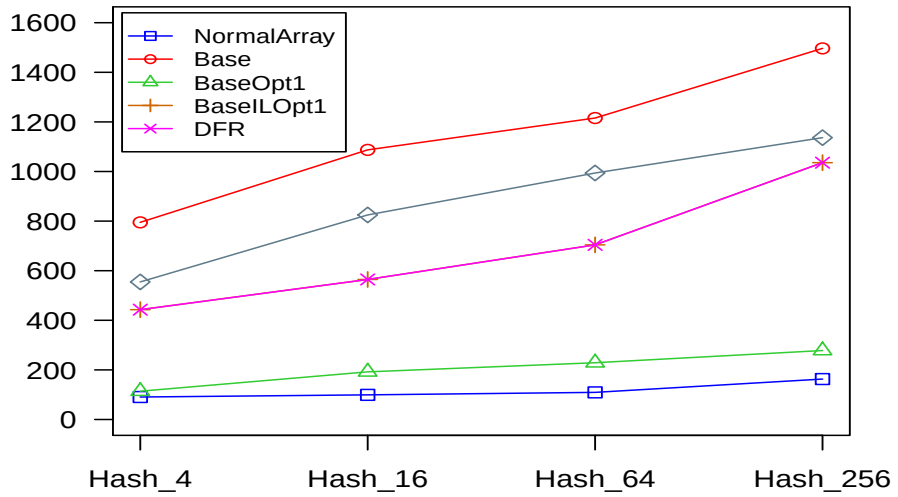


Figure 4.2: Runtimes per set and get operation in nanoseconds for all `hash_n` datasets.

Pareto distribution, random set and get operations.

The random set and get operations were carried out as follows. We first generated a random permutation of $\{0, \ldots, N-1\}$ and stored it in an array R. For $i = 0, \ldots, N-1$, we performed $\mathsf{set}(R[i], v)$, where v is a random number generated from a Pareto distribution. Observe that this sequence of set operations does not set a previously set location. We then performed N get operations at a position in A chosen uniformly at random from $\{0, \ldots, N-1\}$.

The results for the set operations are shown in Fig. 4.3. The normal array is at least three times faster than our optimized approaches on the small datasets. Furthermore, unoptimized Base IL is faster than unoptimized Base as it avoids an extra access to L. Finally, DFR is only competitive with the above unoptimized approaches.

The results for the get operations are shown in Fig. 4.4. For get operations all our optimized approaches and DFR are competitive with a normal array. DFR is much faster than it is for set operations as we now simply apply the appropriate hash function. On the other hand, Base IL performance is badly affected as the number of layers that must be inspected increases, and the Pareto distribution has

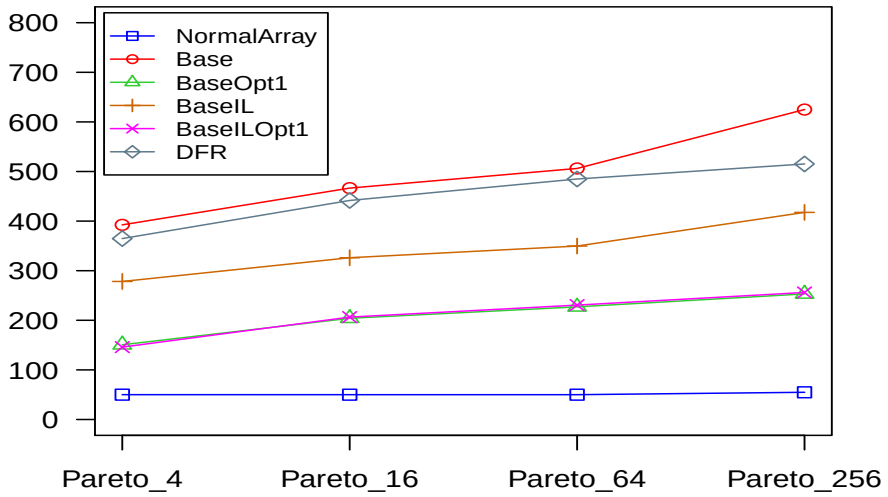


Figure 4.3: Runtimes per random set operation in nanoseconds for `pareto_n` datasets (each index set only once).

large values of k (cf. Table 4.1).

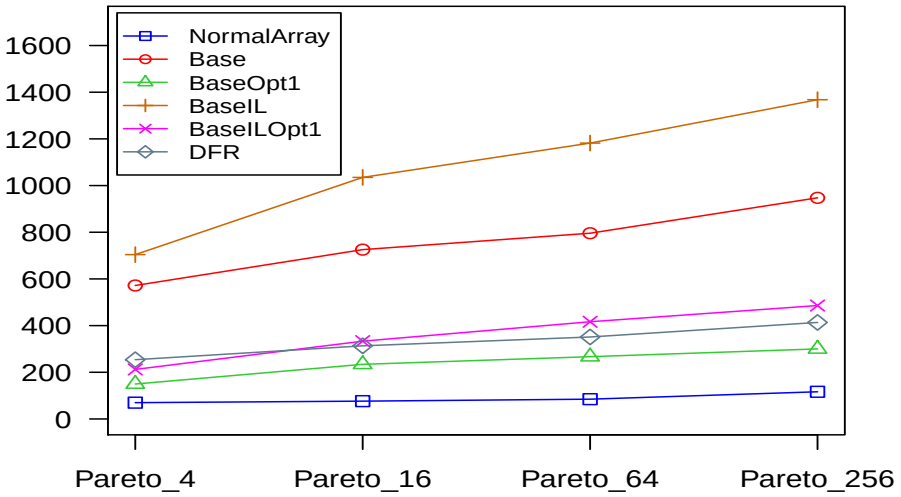


Figure 4.4: Runtimes per random get operation in nanoseconds for all `pareto_n` datasets.

Pareto distribution, random set operations.

We perform $4 \times N$ set operations at locations chosen uniformly at random from $\{0, \ldots, N-1\}$, each time generating a new random value from the Pareto distribution as the second argument to set. In contrast to the previous experiment, in this experiment we will set locations that have previously been set. As a result, we ex-

pect the set operations to be slower than in the prevous experiments, as potentially changes need to be made in two different layers. For the Base IL implementations we used a bit-string to mark the indices in A that have previously not been set, so that when we set to a location for the first time, we avoid unsuccessful searches in all k layers.

The results are shown in in Fig. 4.5. As expected, the running times are slower than those in Fig. 4.3. However, both optimised implementations remain competitive, approximately 3 times slower than the normal array. DFR is still somewhere between optimised and unoptimised Base implementations.

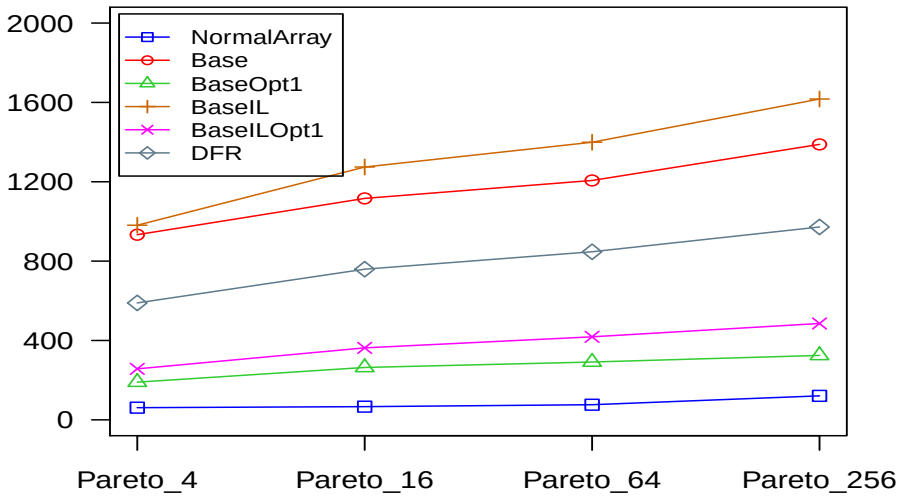


Figure 4.5: Benchmarks per random set and get operations in nanoseconds for `pareto_n` datasets.

Webdocs dataset, random set and get operations.

In this experiment we create a random permutation R as we did in the Pareto benchmark. Another array W of length N contains the `webdocs` values as described in Section 4.4.1. We then measure the average time for set operations by sequentially accessing R and W to set the values in random locations in our data structures. Results are shown in Fig. 4.6. Optimized Base in particular has performance competitive with that of the normal array.

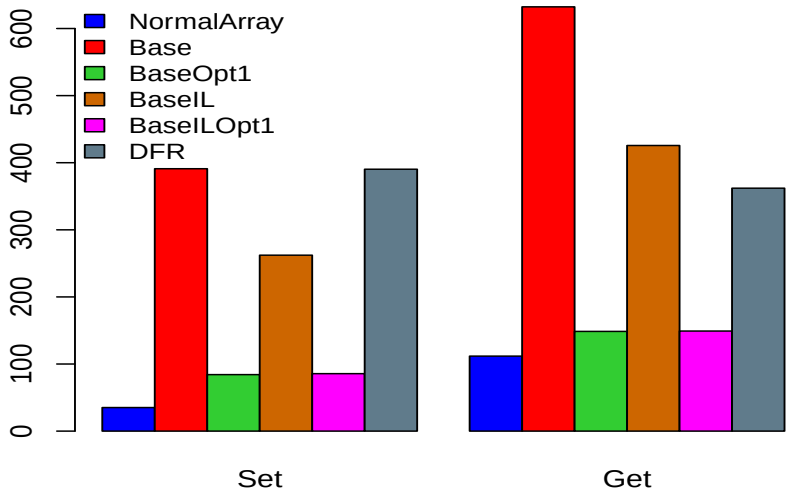


Figure 4.6: Benchmarks per random `set` and `get` operation in nanoseconds for `webdocs` dataset.

4.5 Conclusion

The CDRW array is, to the best of our knowledge, introduced here as a first-class object worthy of study in its own right (although it has been implicit in previous works). The asymptotic results, although not the main thrust of the CDRW arrays, indicate that there is room to study this problem in further detail. For example, there is no obvious reason why a theoretical solution cannot be found that uses $S + o(S)$ bits of space and supports operations in $O(1)$ time. We have, again to the best of our knowledge, given the first preliminary practical implementations of this data structure. These, albeit not highly optimized, implementations already show that low space usage with a low computational overhead is an eminently achievable goal. The runtime speed for the random accesses compare well with normal array and in some experiments the optimized approaches are just 10% to 15% slower. However, the unoptimized approaches can be upto 10 times slower than a normal array. In terms of space, again the optimized approaches compare well with Elias-γ and in the case of Webdocs dataset they can use 30% less memory. An interesting future direction is to tailor CDRW arrays to particular distributions of the values in A, or better yet to design CDRW arrays that adapt to the distribution to optimize space and time.

Chapter 5

m-Bonsai: a practical compact dynamic trie

In this chapter, we turn to the problem of representing a compact dynamic trie with an emphasis on good practical performance. As mentioned in Section 3.3 a dynamic trie is a rooted tree, where each child of a node is labeled with a distinct symbol from an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$. We recall that the ITLB for a tree of n nodes and an alphabet of size σ is $n \log \sigma + O(n)$ bits. This chapter will proceed as follows. We initially give a brief description of the original Bonsai implementation described in Section 3.3.4. We focus on the disadvantages of Bonsai which we try to overcome in this chapter. In Section 5.2, we describe our implementation m-Bonsai (a variant of Bonsai)[1] and give an approach that could potentially use $O(M)$ bits to handle collisions using the *displacement array*. In Section 5.3, we give the ADT required by the displacement array which is the same as the CDRW-arrays from Chapter 4. However, none of the solutions in Chapter 4 are space and time efficient for the specific application. Therefore, we continue with two practical approaches for its implementation. In Section 5.4, we address the problem of traversing m-Bonsai and propose two different approaches for the implementation of the traversal of m-Bonsai in $O(M)$ time. Then, we give the implementation details where we show that m-Bonsai appears to behave in line with the assumptions about the behaviour of the hash function. Finally, we proceed with the description of space usage and runtime tests comparing Bonsai, m-Bonsai variants and ternary search tree.

[1] m-Bonsai stands for *mame* or *mini* Bonsai.

5.1 Introduction

We consider dynamic tries that support the following operations:

- create(): create a new empty tree.
- getRoot(): return the root of the current tree.
- getChild(v, c): return child node of node v having symbol c, if any (and return -1 if no such child exists).
- getParent(v): return the parent of node v.
- getLabel(v): return the label (i.e. symbol) of node v.
- addLeaf(v, c): add a new child of v with symbol c and return the newly created node.
- delLeaf(v, c): delete the child of v with symbol c, provided that the child indicated is a leaf (if the user asks to delete a child that is not a leaf, the subsequent operations may not execute correctly).

As mentioned in Section 3.3.4 Bonsai data structure comes closer to the ITLB compared to other approaches like TST and DAT. The Bonsai data structure, although displaying excellent practical performance in terms of run-time, has some deficiencies as a dynamic trie data structure. We recall that its capacity is expressed in terms of a parameter M, and number of nodes n. The load factor $a = (1 - \epsilon)$ such that $\alpha = n/M$, where $\epsilon > 0$. It supports traversal and update operations in $O(1/\epsilon)$ expected time based on assumptions about the behaviour of hash functions as specified in Section 3.3.4. The limitations of Bonsai are summarized below:

- The user must specify an upper bound M on the trie size in advance, this cannot be changed easily after initialization.
- The space usage of $M \log \sigma + O(M \log \log M)$ is asymptotically non-optimal for smaller σ or if $n \ll M$.

- Bonsai does not support the delLeaf operation.
- In addition, it is not obvious how to traverse an n-node tree in better than $O((n\sigma)/\epsilon)$ time. This also means that the Bonsai tree cannot be resized if n falls well below (or comes too close to) M without affecting the overall time complexity of update operations.

In this chapter we propose a variant of Bonsai, m-Bonsai, that addresses the above problems. The advantages of m-Bonsai include:

- Based upon the same assumptions about the behaviour of hash functions as original Bonsai, our variant uses $M \log \sigma + O(M \log^{(5)} M)$ bits[2] of memory in expectation. Recall that original Bonsai uses the additive term of $O(M \log \log M)$ instead of $O(M \log^{(5)} M)$. This makes a difference in practice especially for tries with low σ as in the case of Bonsai the additive term dominates the total space usage.
- m-Bonsai supports getChild in $O(1/\epsilon)$ expected time, the same as Bonsai. However getParent, getRoot and getLabel are supported in $O(1)$ time, whereas Bonsai uses $O(1/\epsilon)$ expected time.
- Using $O(M \log \sigma)$ additional bits of space, we are able to traverse m-Bonsai tree in $O(M)$ expected time (in fact, we can traverse it in sorted order within these bounds).
- addLeaf and delLeaf can be supported in $O((1/\epsilon)^2)$ amortized expected time. Note that in delLeaf we need to ensure that a deleted node is indeed a leaf.
  - If the application cannot ensure this, one can use the CDRW array data structure from Chapter 4 to maintain the number of children of each node (as it changes with insertions and deletions). For example, choosing Base approach (Section 4.3.1), there will be no asymptotic slowdown in

[2] In this thesis $\log^{(3)} M$ stands for $\log \log \log M$

time complexity and since the maximum number of children per node is $k = \log \sigma$, the space cost is $O(n \log \log \sigma)$ bits.

More precisely, we obtain a trie representation that, for any given constant $\beta > 0$, uses at most $(1 + \beta)(n \log \sigma + O(n \log^{(5)} n))$ bits of space, and supports the operations addLeaf and delLeaf in $O((1/\beta)^2)$ expected amortized time, getChild in $O(1/\beta)$ expected time, and getLabel and getParent in $O(1)$ worst-case time. This trie representation, however, periodically uses $O(n \log \sigma)$ bits of temporary additional working memory to traverse and rebuild the trie.

Finally, we implemented two different approaches for the implementation of m-Bonsai, m-Bonsai(recursive) and mBonsai(γ). Our experimental evaluation shows that the first is consistently faster, and significantly more space-efficient, than the original Bonsai. The second is even more space-efficient but rather slower. Of course, all Bonsai implementations use at least 20 times less space than TSTs for small alphabets and compare well in terms of speed with TSTs. We also note that our experiments show that the hash functions used in Bonsai appear to behave in line with the assumptions about their behavior.

5.2 m-Bonsai approach

First we give an overview of our approach. As in the case of the original Bonsai in Section 3.3.4, each node again has an associated key that needs to be searched for. The hash table implemented is also using open addressing with linear probing and quotienting. Again m-Bonsai uses an array Q of size M to store $n = \alpha M$ nodes for load factor $0 < \alpha < 1$. However, recall that in Bonsai the *nodeID* consists of a triple $< i, j, c >$, where $0 \leq i < M$, $0 \leq j < \lambda$ (λ is discussed in detail in Section 3.3.4) and $0 \leq c < \sigma$ is the node's label. For m-Bonsai, the ID of a node i is also a number from $\{0, \ldots, M-1\}$ that refers to the index in Q that contains the quotient corresponding to i. If a node with ID i has a child with symbol $c \in \Sigma$, the child's key is comprised of the pair $\langle i, c \rangle$ avoiding the j value used in Bonsai. Using the pair

$\langle i, c\rangle$ we need to map the key from a range $\{0, \ldots, M \cdot (\sigma - 1)\}$ to $\{0, \ldots, M - 1\}$ which is the initial hash address i'. More precisely, when inserting a node with label c that is a child of a node with nodeID i the key is $x = i \cdot c$ therefore $0 < x < M \cdot \sigma$. We use a quotienting hash function h which makes use of randomizer r and prime P. P is the next prime number after max key $(M \cdot \sigma)$ and r is a random number $> 2P/3$ as it was used in original Bonsai [14]. The hash function comes in the form of $h(x) = (rx \mod P)$ and therefore $i' = h(x) \mod M$. Finally, if i'' is the smallest index $\geq i'$ such that $Q[i'']$ is vacant, then we store the quotient $q(i)$ in $Q[i'']$ where $q(i) = h(i, c)/M$ (or $h(x)/M$). Observe that $q(i) \leq \lceil 2\sigma \rceil$ since P is less than two times the max key [39], so Q takes $M \log \sigma + O(M)$ bits.

As discussed in Section 3.2.1, in the quotienting scheme the slot i' (initial hash address) gives information about the key. However, as mentioned above due to collision the nodes can be probed to location i''. The key challenge of this chapter is how we keep track of the probed nodes and be able to identify their initial hash address. This should not affect the assymptotic space usage and at the same time be fast in practice.

We now introduce the *displacement array* D. D is also of size M and is used to map the corresponding quotient in Q to its initial address. As mentioned above, if $Q[i'']$ is the first vacant position $\geq i'$, we set $D[i''] = i'' - i'$. From the pair $Q[i'']$ and $D[i'']$, we can obtain both the initial hash address of the key stored there and its quotient, and thus reconstruct the key. We show based on Theorem 2 that the average value in D is small:

Proposition 5. *Assuming h is fully independent and uniformly random, the expected value of $\sum_{i=0}^{M-1} D[i]$ after all $n = \alpha M$ nodes have been inserted is $\approx M \cdot \frac{\alpha^2}{2(1-\alpha)}$.*

Proof. Recall Theorem 2, the average number of probes, over all keys in the table, made in a successful search is $\approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$. Multiplying this by $n = \alpha M$ gives the total average number of probes. However, the number of probes for a key is one more than its displacement value. Subtracting αM from the above and simplifying

gives the result:

$$\approx \frac{\alpha M}{2} \cdot (1 + \frac{1}{1-\alpha}) - \alpha M$$

$$= M \cdot (\frac{\alpha}{2} + \frac{\alpha}{2(1-\alpha)} - \alpha)$$

$$= M \cdot (\frac{\alpha - \alpha^2 + \alpha - 2\alpha + 2\alpha^2}{2(1-\alpha)})$$

$$= M \cdot \frac{\alpha^2}{2(1-\alpha)}$$

□

Therefore, encoding D using variable-length encoding could be very beneficial. For example, using unary codes (see Section 2.4) to encode D, would take $M + \sum_{i=1}^{M} D[i]$ bits, as unary(x) is $x + 1$ bits. Therefore, based on the proof of Proposition 5, D requires $O(M)$ bits which is better than the additive $O(M \log \log M)$ bits of Bonsai.

More precisely, plugging $\alpha = 0.8$, by Proposition 5, we get $1.6M$ bits. Since unary(x) is $x + 1$ bits, that would make it $2.6M$ bits. As shown in Table 5.1, the results based on our experiments show that they go in line with Proposition 5. However, there is no reason to say that encoding D in unary is the best approach. For example, we can use γ and Golomb encodings. We are not aware of any closed form analysis of other encoding methods therefore we turn to experimentation. Table 5.1 is showing the average space per D value using unary, γ and Golomb-codes based on some datasets introduced in Table 5.3. It suggests that encoding each $D[i]$ using the γ-codes, we would come down to about $2.1M$ bits, for $\alpha = 0.8$. We consider, γ-encoding of D to also use $O(M)$ bits, since as we describe in Section 2.4 for all values > 4, γ-codes use less bits than unary. In expectation, Golomb[3] encoding should use even less space than γ-codes as it scales better for bigger numbers. However, due to large percentage of zero values in the distribution of the displacement array, Golomb space usage is badly affected and is slightly worse than γ-codes. Zero values arise when the nodes are hashed in an empty location (without collision) or when the

[3]In Table 5.1, Golomb-codes use parameter $m = 2$ as explained in Section 2.4

	unary			γ			Golomb		
Load Factor	0.7	0.8	0.9	0.7	0.8	0.9	0.7	0.8	0.9
Pumsb	1.81	2.58	5.05	1.74	2.11	2.65	2.32	2.69	3.64
Accidents	1.81	2.58	5.06	1.74	2.11	2.69	2.33	2.69	3.91
Webdocs	1.82	2.61	5.05	1.75	2.11	2.70	2.33	2.70	3.92

Table 5.1: Average number of bits per entry needed to encode the displacement array using the unary, Elias-γ and Golomb encodings. For the unary encoding, Proposition 5 predicts $1.81\dot{6}$, 2.6 and 5.05 bits per value.

location is empty (if $\alpha = 0.8$ then $0.2M$ locations remain empty). As explained in Section 2.4, Golomb encodes zeros using at least 2 bits instead of 1 (compared to unary and γ codes) when parameter $m > 1$.

5.3 Representing the displacement array

We now describe how to represent the displacement array. The displacement array holds a sequence of non-negative integers and can perform the following operations:

- $\mathsf{create}(M)$: Create an array D of size M with all entries initialized to zero.
- $\mathsf{set}(D, i, v)$: Set $D[i]$ to v where $v \geq 0$ and $v = O(M)$.
- $\mathsf{get}(D, i)$: Return $D[i]$.

This is the same ADT as CDRW-arrays from Chapter 4. However, the solutions in Chapter 4 are general purpose arrays and prove not to be as practical for the specific application of the displacement array:

- Base (Section 4.3.1): Since a value v to be inserted is $O(M)$, Base approach is not a good option as it requires the storage of the length of each value which leads to $O(M \log \log M)$ bits like original Bonsai.
- Base IL (Section 4.3.2): Recall that in linear probing once we hash to the initial address we check the elements in every probe. However, we don't have a guarantee that the number of probes are going to be constant. Therefore,

Base IL is not ideal as accessing the displacement values in every probe does not guarantee $O(1)$ time.

- DFR (Section 4.3.3): This approach has the same problem as Base, it requires $O(M \log \log M)$ bits for this application.

Note that the apparently slow running time of set is enough to represent the displacement array without asymptotic slowdown: setting $D[i] = v$ means that $O(v)$ time has already been spent in the hash table finding an empty slot for the key. We now discuss two representation of D that have guarantees in terms of space and work in practice.

5.3.1 m-Bonsai (γ)

We now describe m-Bonsai (γ). We divide D into contiguous blocks of size b. The i-th block $B_i = D[b_{i0} \dots b_{i(b-1)}]$ will be stored in a contiguous sequence of memory locations. There will be a pointer pointing to the start of B_i. Let $G_i = \sum_{j=bi}^{bi+b-1} |\gamma(D[j] + 1)|$. All values in a block are encoded using γ-codes and concatenated into a single bit-string. A set operation is performed by decoding all the γ-codes in the block, and re-encoding the new sequence of γ-codes. Since each γ-code is $O(\log M)$ bits, or $O(1)$ words long, it can be decoded in $O(1)$ time. Decoding and re-encoding an entire block therefore takes $O(b)$ time, which is also the time for the set operation. The get operation requires to access the block and decode the values one by one until we reach the desired value to be returned. Therefore, get takes $O(b)$ time as well. The space usage is $\sum_i G_i + ((M \log M)/b)$ bits where the second term accounts for the pointers and any unused space in the "last" word of a block representation.

5.3.2 m-Bonsai (recursive)

The additive $((M \log M)/b)$ of m-Bonsai(γ) suggests the time and space trade-off. The bigger the b, the better the compression but the slower the speed. The m-Bonsai

(recursive) approach is asymptotically slightly less space-efficient than m-Bonsai(γ) but faster. In this approach, we have two integer parameters $1 < \Delta_0 < \Delta_1$, and the displacement array D is split into three layers:

- The first layer consists of an array D_0 of equal-length entries of Δ_0 bits each, which has size $M_0 = M$. All displacement values $\leq 2^{\Delta_0} - 2$ are stored as is in D_0. If $D[i] > 2^{\Delta_0} - 2$, then we set $D_0[i] = 2^{\Delta_0} - 1$.

- The second layer consists of a CHT with maximum size $M_0 \leq M$. If $2^{\Delta_0} - 1 \leq D_1[i] \leq 2^{\Delta_0} + 2^{\Delta_1} - 2$, then we store the value $D[i] - 2^{\Delta_0} + 1$ as satellite data associated with the key i in the second layer. Note that the satellite data has value between 0 and $2^{\Delta_1} - 1$ and so fits into Δ_1 bits.

- The third layer consists of a standard hash table. If $D[i] > 2^{\Delta_0} + 2^{\Delta_1} - 2$, we store $D[i]$ in this hash table as satellite date associated with the key i.

Clearly, $D[i]$ can be accessed in $O(1)$ expected time. We now describe how to choose the parameters Δ_0 and Δ_1. The following theorem [46] is a large overestimation as the α considered is the resulting density of the hash table. However, in our application we need to consider the density while the hash table is being filled up. Furthermore, we reproduce the proof given in [46] to get the precise value for c_α and b_α explained below.

Theorem 5. *Given an open-address hash table with load factor $\alpha = n/M < 1$, assuming full randomness, the probability that an unsuccessful search makes $\geq k$ probes is at most $b_\alpha \cdot c_\alpha^k$ for some constants b_α and $c_\alpha < 1$ that depend only on α.*

Proof. Let $A[0..M-1]$ be the hash table. Let x be a key that is not in the hash table, and let $h(x) = i$. If the search for x makes $\geq k$ probes then the locations $A[i], A[i+1], \ldots, A[i+k-1]$ must be occupied.[4] A necessary condition for this to happen is that there must exist a $k' \geq k$ such that k' keys are mapped to

[4]To simplify notation we ignore wrapping around the ends of A.

$A[i-k'+k], \ldots, A[i+k-1]$, and the remaining keys are mapped to $A[0..i-k'+k-1]$ or $A[i+k..M-1]$. Under the assumption of full randomness, the number of keys mapped to $A[i-k'+k], \ldots, A[i+k-1]$ is binomially distributed with parameters k'/M and n, and the expected number of keys mapped to this region is $k'n/M = \alpha k'$. Using the multiplicative form of the Chernoff bound, we get that the probability of $\geq k'$ keys being hashed to this region is at most:

$$\left(\frac{e^{1-\alpha}}{(1/\alpha)^{1/\alpha}}\right)^{k'} = c_\alpha^{k'},$$

where $c_\alpha < 1$ is a constant that depends only on α. Summing over $k' = k, k+1, \ldots, n$ we get that the probability of an unsuccessful search taking over k probes is at most $b_\alpha \cdot c_\alpha^k$, as desired. □

Since $\sum_{i=k}^{n} c_\alpha^i \leq c_\alpha \cdot \sum_{i=0}^{\infty} c_\alpha^i$ then we can say that $b_\alpha = \sum_{i=0}^{\infty} c_\alpha^i = \frac{1}{1-c_\alpha}$. Given Theorem 5, we analyze the asymptotic space usage:

- The space usage of the first layer is $M\Delta_0$. We will choose $\Delta_0 = O(\log^{(5)} n)$ so the space usage of this layer is $O(n \log^{(5)} n)$ bits.

- Let the expected number of displacement values stored in the second layer be n_1. Only displacement values $\geq \theta_1 = 2^{\Delta_0} - 1$ will be stored in the second layer, so by Theorem 5, $n_1 \leq b_\alpha \cdot c_\alpha^{\theta_1} \cdot n$. We choose θ_1 so that $c_\alpha^{\theta_1} = \frac{1}{\log^{(3)} n}$, so $\theta_1 \log c_\alpha = -\log^{(4)} n$ and finally $\theta_1 = \frac{\log^{(4)} n}{\log 1/c_\alpha}$. It follows that $\Delta_0 = O(\log^{(5)} n)$ which would make $n_1 = O(\frac{n}{\log^{(3)} n})$.

  We now discuss the asymptotic space usage of the CHT. By Theorem 3, the space usage of this CHT is $M_1(\log(M/M_1) + \Delta_1 + O(1))$ bits, where $M = n/\alpha$ is the universe and $M_1 = n_1/\alpha$ is the size of the CHT[5]. Since $n_1 = O(n/\log^{(3)} n)$, we see that $M/M_1 = O(\log^{(3)} n)$, and hence the asymptotic space usage of the CHT is $O(\frac{n}{\log^{(3)} n}(\log^{(4)} n + \Delta_1 + O(1)))$ bits. We will choose $\Delta_1 = O(\log^{(3)} n)$, so the space usage of the CHT is $O(n)$ bits.

[5]For simplicity, we choose the same load factor for m-Bonsai and the CHT.

- Finally, we discuss the asymptotic space usage of the third layer. Let n_2 be the expected number of keys stored in the third layer. By Theorem 5, $n_2 \leq b_\alpha \cdot c_\alpha^{\theta_2} \cdot n$, where $\theta_2 = 2^{\Delta_1} - 3$. Similar to the Δ_0 scenario, we choose Δ_1 so that $n_2 = O(n/\log n)$, so $\theta_2 = \frac{\log^{(2)} n}{\log_{1/c_\alpha} 2}$ and it follows that $\Delta_1 = O(\log^{(3)} n)$. Since the expected space usage of the third layer is $O(n_2 \log n)$ bits, this is also $O(n)$ bits as required.

5.4 Traversing the Bonsai tree

In this section, we discuss how to traverse a Bonsai tree with n nodes stored in a hash table array of size M in $O(M)$ expected time. We first give an approach that uses $O(M \log \sigma)$ additional bits. This is then refined in the next section to an approach that uses only $n \log \sigma + O(M)$ additional bits. Finally, we show how to traverse the Bonsai tree in *sorted* order.

5.4.1 Simple traversal

Traversal involves a preprocessing step to build three simple support data structures. The first of these is an array A of M $(\log \sigma)$-bit integers. The preprocessing begins by scanning array Q left to right. For each non-empty entry $Q[i]$ encountered in the scan, we increment $A[\mathsf{getParent}(i)]$. At the end of the scan, a non-zero entry $A[j]$ is the number of children of node j. We then create bitvector B, a unary encoding of A, which requires $M + n + o(M + n)$ bits of space. Observe $A[i] = \mathsf{select}_1(B, i) - \mathsf{select}_1(B, i-1)$ and $\mathsf{rank}_0(\mathsf{select}_1(B, i))$ is the prefix sum of $A[1..i]$. We next allocate an array C, of size n to hold the labels for the children of each node. To fill C, we scan Q a second time, and for each non-zero entry $Q[i]$ encountered, we set $C[\mathsf{select}_1(B, \mathsf{getParent}(i)) - \mathsf{getParent}(i) - A[\mathsf{getParent}(i)]] \leftarrow \mathsf{getLabel}(i)$ and decrement $A[\mathsf{getParent}(i)]$. At the end of the scan, $C[\mathsf{rank}_0(\mathsf{select}_1(B, i))..\mathsf{rank}_0(\mathsf{select}_1(B, i+1))]$ contains precisely the labels of the children of node i, and A contains all zeroes. Note that the child labels in

C for a given node are not necessarily in lexicographical order. Preprocessing time is dominated by the scans of Q, which take $O(M)$ time. Space usage is $(M+n)(\log\sigma+1+o(1))$ bits.

With A, B and C we are able to affect a depth first traversal of the trie, as follows. B allows us to determine the label of the first child of an arbitrary node i in constant time: specifically, it is $C[\mathsf{rank}_0(\mathsf{select}_1(B,i))]$. Before we visit the child of i with label $C[\mathsf{rank}_0(\mathsf{select}_1(B,i))]$, we increment $A[i]$, which allows us, when we return to node i having visited its children, to determine the label of the next child of node i to visit: it is $C[\mathsf{rank}_0(\mathsf{select}_1(B,i))+A[i]]$. Traversal, excluding preprocessing, clearly takes $O(n)$ time.

5.4.2 Reducing space

We can reduce the space used by the simple traversal algorithm described above by exploiting the fact that the M values in A sum to n and so can be represented in n bits, in such a way that they can be accessed and updated in constant time with the help of B. Essentially, we will reuse the $M\log\sigma$ bits initially allocated for A to store the C array and a compressed version of A.

We allocate $\min(M\log\sigma, M+n(1+\log\sigma))i$ bits for A, compute it in the manner described in the simple traversal algorithms, and then use it to compute B. The space allocated for A is sufficient to store C, which is of size $n\log\sigma$ bits, and at least another $n+M$ bits. Denote these $n+M$ bits A'. We will use B to divide A' into variable length counters. Specifically, bits $A'[\mathsf{select}_1(B,i-1))+1..\mathsf{select}_1(B,i))]$ will be used to store a counter that ranges from 0 to the degree of node i. A' replaces the use of A above during the traversal phase.

5.4.3 Sorted traversal

We now describe a traversal that can be used to output the strings present in the trie in lexicographical order. In addition to the data structures used in simple traversal,

we store L, a set of σ lists of integers, one for each symbol of the alphabet.

We begin by scanning Q left to right, and for each non-empty entry $Q[i]$ encountered in the scan, we increment $A[\mathsf{getParent}(i)]$ (as in the simple traversal algorithm) and append i to the list for symbol $\mathsf{getLabel}(i)$. At the end of the scan the lists contain n elements in total. Note also that the positions in the list for a given symbol are strictly increasing, and so we store them differentially encoded as Elias-γ codes to reduce their overall space to $n \log \sigma$ bits.

We then compute B as in the simple traversal algorithm and allocate space for C. Now, however, where in the simple algorithm we would make a second scan of Q, we scan the lists of L in lexicographical order of symbol, starting with the lexicographically smallest symbol. For each position j encountered in the scan, we access $Q[j]$ and add $getLabel(j)$ to the next available position in the region of C containing the child labels for node $\mathsf{getParent}(j)$ (which we can access as before with B and A. The child labels in C for a given node now appear in lexicgraphical order, allowing us to affect a lexicographic traversal of the trie.

5.5 Conclusion

Theorem 6. *For any given integer σ and $\beta > 0$, there is a data structure that represents a trie on an alphabet of size σ with n nodes, using $(1+\beta)(n \log \sigma + O(\log^{(5)} n))$ bits of memory in expectation. It supports* getRoot, getParent, *and* getLabel *in $O(1)$ time,* getChild *in $O(1/\beta)$ expected time, and* addChild *and* delLeaf *in $O((1/\beta)^2)$ amortized expected time. The data structure uses $O(n \log \sigma)$ additional bits of temporary memory in order to periodically restructure. The expected time bounds are based upon the existence of a hash function that supports quotienting and satisfies the full randomness assumption.*

Proof. Let M be the current capacity of the hash table representing the trie. We choose an arbitrary location $r \in \{0, \ldots, M-1\}$ as the root of the tree, and set $Q[r] = 0$ (or indeed any value that indicates that r is an occupied location) and

$D[r] = 0$ as well. We store r, which is the ID of the root node, in the data structure. The operations are implemented as follows:

- getRoot: We return the stored ID of the root node.
- getChild(v, c): We create the key $\langle v, c\rangle$ and search for it in the CHT. When searching for a key, we use the quotients stored in Q and $O(1)$-time access to the D array to recover the keys of the nodes we encounter during linear probing.
- addChild(v, c): We create the key $\langle v, c\rangle$ and insert it into the CHT. Let $i = h(\langle < v, c\rangle)$ and suppose that $Q[j]$ is empty for some $j \geq i$. We set $D[j] = j - i$; this takes $O(j - i + 1)$ time, but can be subsumed by the cost of probing locations $i, \ldots, j$.
- getParent(v): If v is not the root, we use $Q[v]$ and $D[v]$ (both accessed in $O(1)$ time) to reconstruct the key $\langle v', c\rangle$ of v, where v' is the parent of v and c is the label of v.
- getLabel(v): Works in the same way as getParent.
- delLeaf(v, c): We create the key $\langle v, c\rangle$ and search for it in the CHT as before. When we find the location v' that is the ID of the leaf, we store a "deleted" value that is distinct from any quotient or from the "unoccupied" value (clearly, if an insertion into the CHT encounters a "deleted" value during linear probing, this is treated as an empty location and the key is inserted into it.

We now discuss the time complexity of the operations. If n is the current number of trie nodes, we ensure that the current value of M satisfies $(1 + \beta/2)n \leq M \leq (1 + \beta)n$ which means that $\epsilon = (M - n)/n$ varies between $(\beta/2)/(1 + \beta/2)$ and $\beta/(1 + \beta)$. Under this assumption, the value of $\epsilon = (M - n)/M$ is $\Theta((1 + \beta)/\beta)$, and the operations addChild, getChild and delLeaf take $O(1/\epsilon) = O(1/\beta)$ (expected) time. If an addChild causes n to go above $M/(1 + \beta/2)$, we create a new hash table

with capacity $M' = \frac{(1+3\beta/4)}{(1+\beta/2)}M$. We traverse the old tree in $O((M)/\beta)$ time and copy it to the new hash table. However, at least $\Omega(\beta n)$ addChild operations must have occurred since last time that the tree was copied to its current hash table, so the amortized cost of copying is $O((1/\beta)^2)$. The case of a delLeaf operation causing n to go below $M/(1+\beta)$ is similar. The space complexity, by the previous discussions, is clearly $M(\log \sigma + O(1))$ bits. Since $M \leq (1+\beta)n$, the result is as claimed. □

5.6 Implementation

In this section, we first discuss details of our implementations. First in in Section 5.6.1, we describe the implementation of CHT. Next we describe the implementation of a naive approach for traversal of m-Bonsai and then the simple linear-time traversal which was explained in Section 5.4.1. All implementations are in C++ and use some components from the `sdsl-lite` library. Finally, we describe the specifications of the machine used, the datasets used, and benchmarks comparing space and speed performance with the original Bonsai implementation and TST.

5.6.1 Cleary's CHT and original Bonsai

We recall the implementation details from Section 3.2.2. We recall that the CHT mainly comprises three `sdsl` containers: firstly, the `int_vector<>` class, which uses a fixed number of bits for each entry, is used for the Q array. In addition, we have two instance of the `bit_vector` container to represent the bit-strings (as in [11, 14]) to map a key's initial address to the position in Q that contains its quotient. The original Bonsai trie is implemented essentially on top of this implementation of the CHT.

5.6.2 Representation of the displacement array

We now give the details of the two implementations for the representation of the displacement array. First, we describe m-Bonsai (γ) and then the alternative approach

of Section 5.3.2, which we call m-Bonsai (recursive) or m-Bonsai (r).

m-Bonsai (γ)

We now describe the implementation details of m-Bonsai (γ). D is split into M/b consecutive blocks of b displacement values each. In our experiments, we choose $b = 256$ considering the trade-off between runtime speed and space usage. We have an array of pointers to the blocks. The displacement values are stored as a concatenation of their γ-codes. We used `sdsl`'s `encode` and `decode` functions to encode and decode each block for the `set` and `get` operations. To perform a `get`(i), we find the block containing $D[i]$, decode all the γ-codes in the block up to position of $D[i]$, and return the value. To perform `set`(i, v), we decode the block containing i, change the appropriate value and re-encode the entire block.

m-Bonsai (r)

As described in Section 5.3.2, m-Bonsai(r) is split into three layers: The first layer D_0, has equal-length entries of Δ_0-bits and is implemented as an `int_vector<>`. The second layer is a CHT and the third layer is implemented using the C++ `std::map`.

Before we proceed with the choice of parameters, we discuss Table 5.2. It shows the expected probability for a node to probe $> k$ steps based on Proposition 4 (Section 3.1.5) and Theorem 5 (Section 5.3.2), after $n = \alpha M$ insertions where $\alpha = 0.8$. The results are compared with real data taken from Webdocs dataset (all datasets behaved similarly) showing the percentage of nodes that probed $> k$ steps. Even though, we did not have a closed form for Proposition 4, we applied the given calculations using MATLAB. The results based on our datasets (Table 5.2, first column) are in-line with the calculations of Proposition 4. The description in Section 5.3.2 is clearly aimed at asymptotic analysis: Theorem 5 proves to be very conservative, as the threshold θ_1 which values end up in the second or third layers, for $n = 2^{65,536}$ and $\alpha = 0.8$, is about 2/9. Consequently, it is not giving important information for a practical size of n as the probabilities are > 1 which could not be

k	Webdocs	Proposition 4	Theorem 5
0	0.314	0.308	13.176
1	0.199	0.181	12.176
2	0.142	0.121	11.252
3	0.109	0.086	10.398
4	0.087	0.062	9.609
5	0.071	0.045	8.880
6	0.051	0.033	8.206
7	0.043	0.023	7.583

Table 5.2: The first column shows k (the percentage of values traveled $> k$ probes). The 2_{nd} column is the results of Webdocs dataset. The 3_{rd} and 4_{th} column shows the expected probability for $> k$ probes based on Prop. 4 and Theorem 5 respectively. The results shown are after $n = \alpha M$ insertions were $\alpha = 0.8$.

possible. The results in Table 5.2 are calculated based on the proof of Theorem 5. For the given example we chose $\alpha = 0.8$, $M = 100$, $b_\alpha = 1$ and calculate c_α based on the proof of Theorem 5 which is $= 0.92$. Since c_α^k is the probability that the keys are hashed at the k^{th} probe exactly, we sum up the probabilities ranging from k to $\alpha \cdot M$ (80 in this example) to get the probability that a node will do $> k$ probes.

We now discuss the choice of parameters Δ_0 and Δ_1. The values of Δ_0 and Δ_1 are currently chosen numerically. Specifically, we compute the probability of a displacement value exceeding k for load factors $\alpha = 0.7, 0.8$, and 0.9 using the exact analysis in Proposition 4. This numerical analysis shows, as the example in Figure 5.1, that for $\alpha = 0.8$, choosing $\Delta_0 = 2$ or 3 and $\Delta_1 = 6, 7$, or 8 give roughly the same expected space usage. Clearly, choosing $\Delta_0 = 3$ would give superior performance as more displacement values would be stored in D_0 which is just an array, so we chose $\Delta_0 = 3$. Given this choice, even choosing $\Delta_1 = 7$ (which means displacement values ≥ 134 are stored in the third layer), the probability of a displacement value being stored in the third layer is at most 0.000058. Since storing a value in the `std::map` takes 384 bits on a 64-bit machine, the space usage of the third layer is negligible for a 64-bit machine.

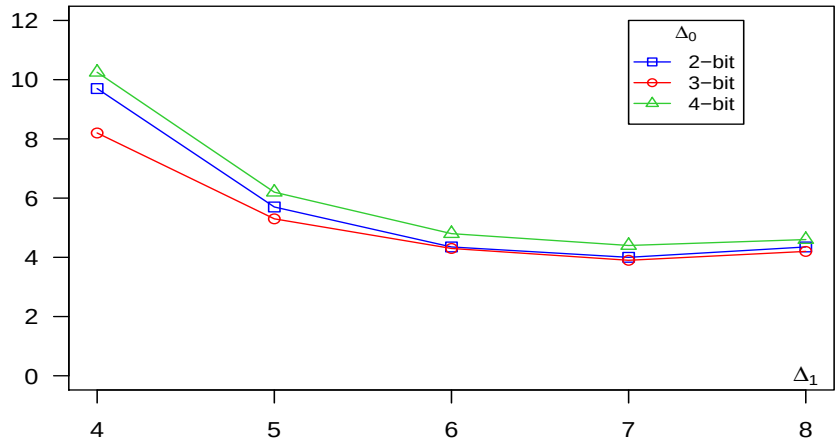


Figure 5.1: This graph is an example based on Webdocs dataset. Used m-Bonsai (r) data structure with $\alpha = 0.8$. The y-axis shows the bits divided by M required by the displacement array. The x-axis shows parameter Δ_1 and each line is based on parameter Δ_0.

5.6.3 m-Bonsai traversal

We now discuss the implementation of traversals. As discussed, the difficulty with both Bonsai data structures is that the getChild and getParent operations only support leaf-to-root traversal.

One approach to traversing a tree with this set of operations is as follows. Suppose that we are at a node v. For $i = 0, \ldots, \sigma - 1$, we can perform getChild(v, i) to check if v has a child labelled i; if a child is found, we recursively traverse this child and its descendants. This approach takes $O(n\sigma)$ time.

The algorithm described in Section 5.4.1 was implemented using `sdsl` containers as follows. The arrays A and C are implemented as `sdsl int-vectors` of length M and n respectively, with width $\lceil \log \sigma \rceil$. The bit-vector was implemented as a `select_support_mcl` class over a `bit_vector` of length $M + n$.

5.7 Experimental evaluation

5.7.1 Datasets

We use benchmark datasets arising in frequent pattern mining [25], where each "string" is a subset of a large alphabet (up to tens of thousands). We also used sets of short read genome strings given in the standard FASTQ format. These datasets have a relatively small alphabet $\sigma = 5$. Details of the datasets can be found in Table 5.3.

5.7.2 Experimental setup

All the code was compiled using g++ 4.7.3 with optimization level 6. The machine used for the experimental analysis is an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz with 3MB L2 cache, running Ubuntu 12.04.5 LTS Linux. To measure the resident memory (RES), `/proc/self/stat` was used. For the speed tests we measured wall clock time using `std::chrono::duration_cast`.

5.7.3 Tests and results

We now give the results of our experiments, divided into tests on the memory usage, benchmarks for build, traverse and successful search operations. Our m-Bonsai approaches where compared with Bonsai and Bentley's C++ TST implementation [7]. The DAT implementation of [55] was not tested since it apparently uses 32-bit integers, limiting the maximum trie size to 2^{32} nodes, which is not a limitation for the Bonsai or TST approaches. The tests of [55] suggest that even with this "shortcut", the space usage is only a factor of 3 smaller than TST (albeit it is ~ 2 times faster).

Memory usage. We compare the aforementioned data structures in terms of memory usage. For this experiments we set $\alpha = 0.8$ for all Bonsai data structures.

Then, we insert the strings of each dataset in the trees and we measure the resident memory. Table 5.3 shows the space per node (in bits). We note that m-Bonsai (γ) consistently uses the least memory, followed by m-Bonsai (r). Both m-Bonsai variants used less memory than the original Bonsai.

Since the improvement of m-Bonsai over Bonsai is a reduction in space usage from $O(n \log \sigma + n \log \log n)$ to $O(n \log \sigma + n \log^{(5)} n)$, the difference will be greatest when σ is small. This can be observed in our experimental results, where the difference in space usage between m-Bonsai and the original Bonsai is greatest when σ is small. In the FASTQ data sets, original Bonsai uses 85% more space than m-Bonsai (r) while the advantage is reduced to 23% for webdocs. Of course, all Bonsai variants are significantly more space-efficient than the TST: on the FASTQ datasets, by a factor of nearly 60. Indeed, the TST could not even load the larger datasets (FASTQ and webdocs) on our machine.

Datasets	n	σ	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Chess	38610	75	13.99	11.94	17.51	389.56
Accidents	4242318	442	16.45	14.42	19.99	388.01
Pumsb	1125375	1734	18.95	16.93	22.51	387.52
Retail	653217	8919	22.71	20.69	26.25	384.91
Webdocs8	63985704	364	16.45	14.44	19.99	386.75
Webdocs	231232676	59717	25.20	23.19	28.72	—
SRR034939	3095560	5	8.94	6.78	12.51	385.88
SRR034944	21005059	5	8.93	6.74	12.51	385.76
SRR034940	1556235309	5	8.93	6.77	12.51	—
SRR034945	1728553810	5	8.93	6.79	12.51	—

Table 5.3: Characteristics of datasets and memory usage (bits per node) for all data structures. TST was not able to complete the process for larger datasets.

Tree construction speed. In Table 5.4 we show the wall clock time in seconds for the construction of the tree. Of the three Bonsai implementations, m-Bonsai (r) is always the fastest and beats the original Bonsai by 25% for the bigger datasets and about 40% for the smaller ones.

We believe m-Bonsai (r) may be faster because Bonsai requires moving elements in Q and one of the bit-vectors with each insertion. When inserting a node in m-Bonsai (r) each $D[i]$ location is written to once and D and Q are not rearranged

after that. Finally, m-Bonsai (γ) is an order of magnitude slower than the other Bonsai variants. It would appear that this is due to the time required to access and decode concatenated γ-encodings of a block, append the value and then encode the whole block back with the new value.

Comparing to the TST, m-Bonsai (r) was comparably fast even on small datasets which fit comfortably into main memory and often faster (e.g. m-Bonsai is 30% faster on webdocs). The difference appears to be smaller on the FASTQ datasets, which have small alphabets. However, the TST did not complete loading some of the FASTQ datasets.

Datasets	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Chess	0.02	0.21	0.08	0.02
Accidents	2.18	21.46	3.01	2.31
Pumsb	0.43	5.85	0.69	0.57
Retail	0.22	2.27	0.25	0.31
Webdocs8	26.07	252.59	32.75	18.25
Webdocs	96.38	869.22	130.92	—
SRR034939	0.61	10.25	0.79	0.61
SRR034944	5.72	70.83	7.34	4.31
SRR034940	730.55	6,192.14	970.81	—
SRR034945	841.87	7,199.11	1,106.39	—

Table 5.4: The wall clock time in seconds for the construction of the Trie. TST was not able to complete the process for larger datasets.

Traversal speed. In Table 5.5 we compare the simple linear-time and naive traversals, using m-Bonsai (r) as the underlying Bonsai representation. After we construct the trees for the given datasets, we traverse and measure the performance of the two approaches in seconds. The simple linear-time traversal includes both the preparation and the traversal phase. Since the naive traversal takes $O(n\sigma)$ time and the simple traversal takes $O(n)$ time, one would expect the difference to be greater for large σ. For example, Retail is a relatively small dataset with large σ, the difference in speed is nearly two orders of magnitude. Whereas all FASTQ datasets are consistently only 2 times slower.

Surprisingly, even for datasets with small σ like the FASTQ, naive approach does worse than the simple linear-time approach. This may be because the simple linear-

time traversal makes fewer random memory accesses (approximately $3n$) during traversal, while the naive approach makes $n\sigma = 5n$ random memory accesses for these datasets. In addition, note that most searches in the hash table for the naive traversal are *unsuccessful* searches, which are slower than successful searches.

Datasets	Simple traversal	Naive traversal
Chess	0.02	0.38
Accidents	4.82	228.92
Pumsb	1.01	233.11
Retail	1.04	788.36
Webdocs8	104.92	6,617.39
Webdocs	150.81	—
SRR034939	2.61	4.52
SRR034944	24.78	41.94
SRR034940	3,352.81	7,662.37
SRR034945	4,216.82	8,026.94

Table 5.5: The wall clock time in seconds for traversing the tree using simple and naive approach.

Successful search speed. Now we explain the experiment for the runtime speed for successful search operations. For this experiment we designed our own *search*-datasets, where we randomly picked 10% of the strings from each dataset, shown in Table 5.6. After the tree construction, we measured the time needed in nanoseconds per successful search operation. It is obvious that TST is the fastest approach. However, m-Bonsai (recursive) remains competitive with TST and is consistently faster than Bonsai by at least 1.5 times, whereas m-Bonsai (γ) is the slowest. Note that there is an increase in runtime speed per search operation for all Bonsai data structures as the datasets get bigger, since there are more cache misses. For TST, we see that Retail with high σ is affecting the runtime speed, as TST can search for a child of a node in $O(\log \sigma)$ time.

Datasets	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Chess	130	1240	288	59
Accidents	187	1342	399	60
Pumsb	134	1204	301	55
Retail	407	1244	418	102
Webdocs8	296	1573	586	61
Webdocs	352	1705	795	—
SRR034939	173	1472	350	65
SRR034944	247	1682	498	66
SRR034940	451	1946	709	—
SRR034945	511	1953	718	—

Table 5.6: The wall clock time in nanoseconds per successful search operations.

5.8 Conclusion

We have demonstrated a new variant of the Bonsai approach to store large tries in a very space-efficient manner. Not only have we (re)-confirmed that the original Bonsai approach is very fast and space-efficient on modern architectures, both m-Bonsai variants we propose are significantly smaller (both asymptotically and in practice) and one of them is a bit faster than the original Bonsai. More precisely, m-Bonsai (r) is the fastest of the Bonsai approaches. For the insert operation it is between 25% and 40% faster than original Bonsai, and in some cases faster than TST. The simple traversal proposed proves to be much faster than the naive traversal especially when the alphabet size is large (thousands), simple traversal can be upto 2 orders of magnitude faster. In terms of space, Bonsai uses upto 85% more space usage than m-Bonsai (r). Neither of our approaches is very close to the information-theoretic lower bound of $(\sigma \log \sigma - (\sigma - 1) \log(\sigma - 1))n - O(\log(kn))$ bits [6]. For example, for $\sigma = 5$, the lower bound is $3.61n$ bits, while m-Bonsai (γ) takes $\sim 5.6M \sim 7n$ bits. Closing this gap would be an interesting future direction. Another interesting open question is to obtain a practical compact dynamic trie that has a wider range of operations, e.g. being able to navigate directly to the sibling of a node.

Chapter 6

Preliminaries: Frequent Pattern (FP)-Growth

In this chapter we give the preliminaries required for the efficient implementation of the FP-growth algorithm. It is considered one of the fastest algorithms for the solution of frequent itemset mining. We give the potential challenges of in-memory data representation when implementing FP-growth. Furthermore, we show the classic approach by Han et al. [34] and an efficient implementation by Schlegel et al. [51]. We emphasize on their weaknesses which we try to overcome in the following chapter.

6.1 Definition of frequent itemset mining (FIM)

The problem of computing frequent itemsets in a transaction database, or frequent itemset mining, is among the most heavily studied sub-problems in data mining. As we mentioned in Section 1.5, a number of applications, such as mining association rules, rely on the FIM problem as a crucial subroutine [1]. Note that this chapter and Chapter 7 are strongly related, therefore the terminology used in the following definition, is going to be used in both chapters.

The FIM problem may be defined as follows. Let Σ be a set of *items* and let $\sigma = |\Sigma|$. A *database* $D = \{t_1, t_2, \ldots, t_n\}$ is a (multi)-set of *transactions* where $t_i \subseteq \Sigma$, for $i = 1, \ldots, n$. For any $s \subseteq \Sigma$, define its *support* in D, denoted $Sup(s, D)$, as $|\{t \in D \mid s \subseteq t\}|$. The objective is to find all frequent itemsets, namely all sets s such that $Sup(s, D) \geq \theta n$ for some user-defined threshold $0 \leq \theta \leq 1$.

6.2 FP-growth algorithm

Among the numerous approaches to solving the FIM problem, the FP-growth algorithm [34] and its variants are generally considered among the fastest approaches [28]. It is split into two phases; the *build phase* and the *mine phase*. Initially, (for the build phase) we load the entire database D into a data structure in memory called the *FP-tree*. More precisely, FP-growth views each transaction as a string, and essentially inserts each transaction one by one into a trie: this trie is (almost) the FP-tree. Due to prefix-sharing, the number of nodes in the FP-tree, N, can be significantly smaller than $|D| = \sum_{i=1}^{n} |t_i|$. After the build phase, FP-growth traverses the FP-tree to output all frequent itemsets in the mine phase. During the mine phase, it is essential that the FP-tree fits entirely in main memory for the mining to complete quickly.

In the remaining chapter, we first sketch the build and mine phases of the FP-growth algorithm [34]. In Section 6.2.3, we give the operations required by the FP-tree data structure. Next, in Section 6.3, we give the classic implementation of the FP-tree along with the physical design of the data structure used. Finally in Section 6.4, we analyze a compact approach for the FP-growth algorithm proposed by Schlegel et al. [51] and its disadvantages.

6.2.1 Build phase

We assume that D contains no infrequent items (since such items cannot be in any frequent itemset); that Σ is sorted in decreasing order of support (i.e. if $x, y \in \Sigma$ and $x < y$ then $Sup(x, D) > Sup(y, D)$); and finally that each transaction is sorted in increasing order of items. If these assumptions are not satisfied, we make a second pass over D to satisfy these assumptions.

We view the database D as a (multiset) of strings over the alphabet Σ. The FP-Tree, which is essentially a trie (as Section 3.3) over the strings in D, is constructed by scanning D. We let N denote the number of nodes in this trie. Each node in

the FP-tree is labelled with an item, called the node's *itemID*. Each node represents a prefix p of a transaction in D: the number of transactions in D of which p is a prefix is the node's *count*. Finally, pointers called *nodelinks* are stored to connect all the nodes that have the same itemID as they are needed for the mining phase, explained below.

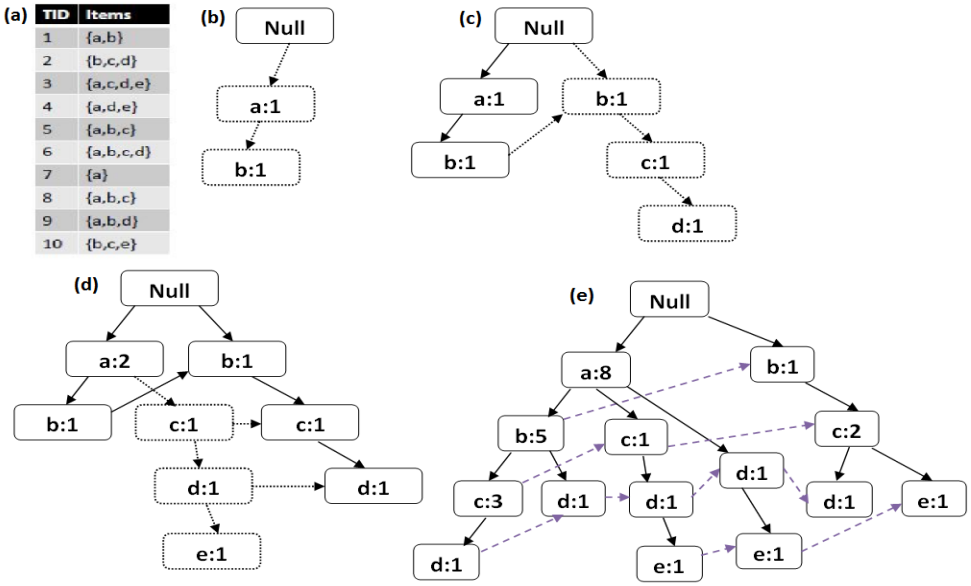


Figure 6.1: FP-Tree creation. (a) shows the database, with transactions sorted according to frequency. States (b), (c) and (d) show how the FP-Tree is populated for TIDs 1, 2, and 3 respectively. (e) Final FP-tree.

6.2.2 Mine phase

We now describe the mine phase. Let $s = x_1 x_2 \ldots x_k$, where $x_i \in \Sigma$ and $x_1 < x_2 < \ldots < x_k$. To create the *conditional pattern base* for itemset s, denoted by D_s, we select all transactions in D that (a) contain all the items in s and (b) contain no items that are less frequent than x_1. From these transactions, delete all items $\geq x_1$. For example, the conditional pattern base for $s = \{e\}$ in the example database in Figure 6.1 (State e), is $\{acd, ad, bc\}$. Given a conditional pattern base for s, and a new item $x < \min_i x_i$, the operation of *projection* produces the conditional pattern base for $x \cup s$. For example, given the conditional pattern base for e, projection

on c would result to itemset $x \cup s = s' = \{c, e\}$, thus in our example $D_{s'} = \{a, b\}$. The mining proceeds as follows. Starting with the conditional pattern base D_s for some s, we determine all the frequent items D_s, and for each such frequent item x, we recurse on $D_{x \cup s}$. The algorithm is started with s set to λ, the empty string (and $D_\lambda = D$). The FP-growth algorithm extensively uses nodelinks to create the conditional pattern bases.

6.2.3 FP-Tree ADT

In order to support the build phase, the FP-Tree should support the following operations:

- getItemID(v): return itemID of a specific node v.
- incrementCount(v): increment count of node v by one.
- getChild(v, i): return child node of node v with itemID i, if any (and return null if no such child exists).
- addChild(v, i): add a new child with itemID i and return the node number of the newly created node.

For the mine phase, the following operations are needed:

- getCount(v): return count of node v.
- getParent(v): return parent node of node v.
- listNodes(x): List all the nodes with itemID x.
- project(D_s, x): Given a conditional pattern base for s, and an item $x < \min(s)$, returns the conditional pattern base for itemset $x \cup s$.

Note that the last operation is used to create conditional FP-trees.

6.3 FP-tree implementation and physical design

An important part of FP-growth or any prefix tree-based algorithm is the physical representation of the tree. The tree must be able to provide the operations required by the algorithm. During the build phase, one wants to find/create the prefix corresponding to the current transaction fast. For example, to insert the transaction $\{a, b\}$ into the FP-tree, we have to find the direct suffixes[1] of the root with itemID $= a$, and then the node with itemID $= b$ among the direct suffixes of a. The counts of both a and b are increased by one. Search for the next node among the direct suffixes must be fast as it is important for the overall runtime speed.

We now explain the original implementation of the FP-tree proposed by Han et al. [34]. The physical representation is based on a ternary search tree (TST) (Section 3.3). We recall that the main idea of a TST is to arrange direct suffixes in a binary search tree. To do this we need two pointers for left and right sibling, a suffix (or child) pointer. Now for the purposes of the FP-tree we also need a parent and a nodelink pointer. Finally, a node also requires space to store two integers, the itemID and the count.

Searching for a direct suffix in a TST requires logarithmic complexity. However, its biggest disadvantage is the high memory consumption: the five pointers require 40 bytes per node on a 64-bit machine. This is considered very inefficient since we need to use 40 bytes of memory whereas the actual information is two integers itemID and count; that is if compressed, they would require approximately 1 byte each. On trees with a lot of nodes, these high memory requirements could potentially force out-of-core computation and thus severely slow down computation.

6.4 CFP-growth

Schlegel et al. [51] recently addressed the problem of the memory usage of the FP-tree. They gave two data structures, one each to replace the FP-tree during the build

[1] In this chapter, we consider a child or descendant of a node as its suffix.

and mine phases, called the *CFP-tree* and *CFP-array* respectively. Having two data structures means that, after the build phase, the CFP-tree must be traversed to construct the CFP-array during a *conversion phase*. During the conversion phase, the two data structures coexist in memory; this is the point of the peak memory consumption of such approach.

On the benchmark datasets discussed in [51], their peak memory usage is 5-10 times less than standard approaches like the one in Section 6.3. There was a small reduction in the computation time for small trees, and a huge gain on larger trees when the standard FP-tree implementation does not fit in main memory. Despite these impressive achievements, their approach has some weaknesses:

- The CFP-tree is based on *Patricia trie* compression [39], i.e. removal of nodes with only one child from the FP-tree. The effectiveness of Patricia compression is data-dependent: in the worst case no nodes may be removed. Indeed, as seen in the results of Schlegel et al. the space usage of the CFP-tree ranges between just under $2N$ bytes to $5.6N$ bytes, where N is the number of nodes in the original FP-tree, depending on how successful Patricia compression is on the particular dataset [51]. We calculate that if there is no Patricia compression, the space usage of the CFP-tree would be $> 7N$ bytes.

- In the CFP-tree implementation, all pointers are limited to 40 bits. This restricts their implementation to at most 2^{40} bytes of addressable memory, a limitation that does not affect standard approaches [34]. Since Schlegel et al. make ingenious use of the "remaining" 24 $(= 64 - 40)$ bits in a 64-bit word to minimize memory wastage due to alignment and fragmentation, it is not easy to estimate the space usage of a full 64-bit implementation of the CFP-tree. In what follows, we conservatively estimate the space usage of a full 64-bit implementation of the CFP-tree by adding 3 bytes to the space usage for every pointer.

- The data structure that Schlegel et al. use in the mine phase, the CFP-array,

shows less variation in space usage across the benchmark datasets considered by Schlegel et al. (ranging from 4 to 4.5 bytes per node). Furthermore, in contrast to the CFP-tree, the CFP-array is a full 64-bit implementation. However, we give an artificial transaction database where the per-node space usage of the CFP-array increases with N while σ remains the same. The space usage ranges from 5 bytes (for $N = 409$) to 8 bytes (for $N = 403$ million).

Seen from an asymptotic perspective, the space usage of both the CFP-tree and CFP-array approaches is non-optimal. The FP-tree can be viewed as a trie over an alphabet of size σ. As discussed in Section 3.3, the minimum space bound for representing a trie with N nodes over an alphabet of size σ is $N \log \sigma + O(N)$ bits [6]. In other words, the space usage per node of an optimal trie representation is $\log \sigma + O(1)$ bits: it is only dependent on the alphabet size, and not on the number of nodes in the trie. On the other hand, both the CFP-tree and CFP-array have a worst-case asymptotic space usage of $\Theta(N \log N)$ bits.

6.4.1 CFP-tree

We now describe the CFP-Tree of Schlegel et al.. In contrast to standard FP-tree implementations, the CFP-tree is based on essentially a *Patricia* trie [39], where trie nodes of degree one are eliminated, and instead the sequence of labels on the path of degree-1 nodes is stored in the last node on the path. The CFP-tree is represented as a ternary search tree, but with a number of additional heuristics and coding tricks to reduce space usage:

- Firstly, they reduce the number of pointers by suppressing null pointers: each node in their ternary search tree has 0 to 3 pointer fields, and the header stores three bits to indicate which null pointers were suppressed. This is quite effective since each Patricia trie node, on average, is guaranteed to have only one non-null pointer. Furthermore, as already mentioned, each pointer is limited to 5 bytes, thus limiting the addressable memory to 2^{40} bytes.

- Secondly, itemIDs are stored as differences. Since transactions are composed of items with increasing itemIDs, we only store the Δ-*item*, i.e. the difference between the itemIDs of a node and its parent.

- In addition, only *partial* count (*pcount*) information is stored: with each node in the trie that represents the end of a transaction, we store the number of times that transaction appears in the database D (the count values in the FP-tree are thus only available implicitly), suppressing zero pcounts altogether.

  - Variable-length encoding (7-bit varint [42]) is used for storing Δ-items (which consequently rarely exceed two bytes) and pcounts (though in some datasets most pcounts need zero bytes).

- The good memory performance of the CFP-tree is heavily reliant on efficient storage of *chains* of degree-1 nodes in the FP-tree: a chain of m nodes is normally represented using just $m + 6$ bytes.

- Finaly, each Patricia tree node in the CFP-tree consists of a *compression mask*, which indicates which pointers are present and auxiliary information for the variable-length encoding, the Δ-item, the pcount and up to three ternary tree pointers. Each non-leaf node may require 7-24 bytes: one byte for the compression mask, 1-4 bytes for the Δ-item, 0-4 bytes for the pcount and 5 bytes per existing outgoing pointer. Hand-crafted memory management is needed to minimize fragmentation caused by the variable node sizes.

Unpredictable memory usage. This highly optimized approach yields very good performance on some datasets using different heuristics. This results in inconsistent and unpredictable performance. There can be many reasons why on an individual dataset, the space usage is unexpectedly higher per node. We proceed with the analysis of the space usage, and how Patricia compression may be affected by datasets properties.

Table 6.1 shows that the percentage of leaves per node can potentially give an intuition of the expected space usage. The higher the percentage of leaves, the more likely to have a wider and more bushy FP-tree. This means fewer and/or shorter chains (worst performance on Patricia compression). However, Retail has half the percentage of leaves than Mushroom but the space usage per node is more. There are a few reasons why this is happening. As it will be shown in Table 7.1, Retail has average transaction length = 10 and at the same time σ = 8919, i.e. most of the Δ-items are greater than 255. On the other hand, Mushroom has average transaction length = 23 and σ = 119. This shows that if we have a wide FP-tree and a dataset with large σ it may cause a dramatic increase in CFP-tree memory usage.

The second column in Table 6.1, shows the space usage per Patricia node in bytes. For the CFP-tree, a Patricia node can be a single node or a chain of m nodes. It is clear that the fewer the Patricia nodes the longer the chains and/or the less the number of chains. Therefore, the more bytes used per Patricia node the better the compression thus the better the space usage per node. Below we give reasons for a chain to break:

- Δ-item > 255 break the chain, since a chain can only use 1 byte per node.
- A node with pcount > 0 breaks the chain since a chain does not hold pCounts.
- Degree of a node. If a child has siblings it cannot be in a chain.

The properties of a dataset can also impact the Patricia compression, individually or a combination of them:

- The transaction length. Long transactions may cause long chains.
- The alphabet size may result to bushier tree or itemIDs$>$ 255.

Looking at Table 6.1 – space per Patricia node – Retail space per node could be expected to be close to Chess however, it is is more than double. One of the properties

File	Leaves per node	space per Patricia-node	space per node
Mushroom.dat	0.23	6.88	5.28
Connect.dat	0.19	6.98	4.49
Retail.dat	0.11	8.24	5.66
Chess.dat	0.08	8.86	2.64
Accidents.dat	0.08	9.25	2.4
Pumsb.dat	0.04	11.16	2.38
Webdocs.dat	0.02	14.77	1.91

Table 6.1: CFP-Tree performance on real datasets. "Patricia-node" is any node using Patricia compression (including single and chain nodes). 2_{nd} and 3_{rd} columns show the space in bytes per Patricia node and per node respectively.

of Chess is that all transactions are of the same length. This makes all the pcounts be zero until we reach a leaf. On the other hand, Retail transactions vary in length. Storing pcounts > 0 is a reason for the chain to break. In addition, because of the fact that Chess has longer transactions (average transaction $= 37$) and smaller $\sigma = 75$; the space per node is 3 bytes smaller in comparison to Retail.

In summary, in the worst case, the Patricia trie may have the same number of nodes as the FP-tree. If N is the number of nodes in the FP-tree, since there is on average one pointer per node, the space usage must be $\Omega(\log N)$ bits per node, since a pointer must use $\geq \log N$ bits to address N nodes. As noted already, this is asymptotically non-optimal.

6.4.2 CFP-array

The CFP-Array is the data structure used by Schlegel et al. for the mine phase. It is an array divided into σ blocks, one for each item in Σ; the block for an item i contains all nodes with itemID i. Each node is represented by a triple (Δ-item, Δ-pos,count), where Δ-pos is the relative position of the node's parent from the start of the parents block, and the count (not the pcount) is the full count information as in the standard FP-growth algorithm. All the above values are stored using 7-bit varint encoding (also known as varint128). This encoding works by splitting an n-bit integer into a series of blocks. The successive blocks use 1 byte each, in which the lower 7 bits

are used to store the actual data, and the highest bit is a continuation bit which indicates whether or not an additional block follows. For example, the hexadecimal value `00000090` is encoded using the following 2 bytes: `(1)0000001 (0)0010000`. The CFP-array, unlike the CFP-tree, is a static data structure, and in principle it is a full 64-bit implementation. The performance of the CFP-array on the benchmark data sets has less variation compared to the performance of the CFP-tree — it uses from 4 to 4.5 bytes per node of the original FP-tree.

However, the per-node space usage of the CFP-array is essentially guaranteed to increase if one increases the number of transactions while keeping Σ fixed: as the size of the block for each item increases, the number of bytes needed to represent Δ-pos also increases. The rate of increase would be proportional to $\log N$ where N is the number of nodes in the FP-tree. This is illustrated clearly by the artificial transaction database we now describe.

"RxEy.dat" dataset was constructed to highlight the issue. This dataset was formed based on the size of set $R = \{1, 2, \ldots, x\}$ which is x, and the size of sequence $E = (x + 1), (x + 2), \ldots, (x + y)$ which is y. Given R and E above, the dataset's $\Sigma = \{1, 2, \ldots, (x + y)\}$, therefore $\sigma = (x + y)$ where itemIDs are already sorted according to their frequency. For simplicity the $D_{x,y}$ database is the product of the *"RxEy.dat"* dataset. $D_{x,y}$ contains the following transactions.

- All non-empty subsets of $R = \{1, \ldots, x\}$. There are $2^x - 1$ such transactions.
- We have additional $2^x - 1$ transactions such that $E = \{(x + 1), \ldots, (x + y)\}$ sequences are appended to the above subsets of R.

For further clarification, to form such a trie, we have the $2^x - 1$ transactions containing R subsets and then append the E sequences to the R subsets. The ordering of the values is not the main focus of this experiment as there are many datasets where transactions are repeated and alter the ordering accordingly. We are mostly interested on the tree structure as it directly impacts the memory usage per node. Further, notice that the FP-tree of $D_{x,y}$ contains $O(2^x y)$ nodes. Since $\sigma = x + y$, by

incrementing x and decrementing y, we can increase the number of nodes while keeping the alphabet fixed. There are 2^x nodes each with itemIDs $x+1, x+2, \ldots, x+y$, so the Δ-pos fields for all nodes with itemIDs $x+2, \ldots, x+y$ must be at least x bits long. Thus, by increasing x and decreasing y we not only increase the number of nodes, but the number of bits per node.

The Δ-pos value is compressed using 7-bit variable encoding. Figure 6.2 shows what happens to the CFP-array on $D_{x,y}$ when x is increased and y decreased. At certain values of x, the variable-length encodings of the Δ-pos values require an extra byte, causing a significant "jump" in space usage per node. It is not hard to see that asymptotically the space usage of the CFP-array is $\Omega(N \log N)$ bits on these transaction databases.

In summary, the CFP-Array space usage is mainly depending on the compression of Δ-pos value. CFP-Array requires $N \log(N/\sigma)$ bits per node in most cases. However, the characteristics of given datasets may affect the scalability of CFP-Array significantly raising the space usage to $N \log(N)$.

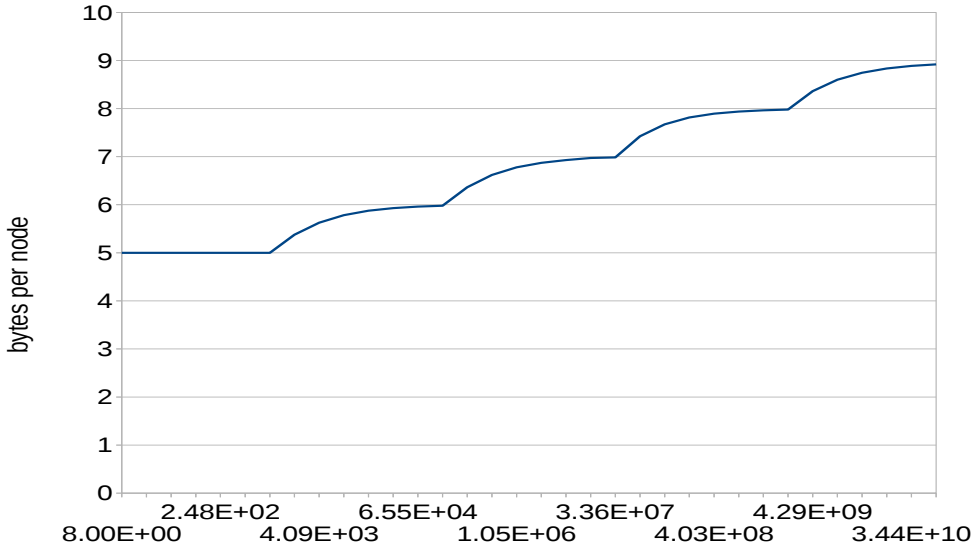


Figure 6.2: While σ remains the same, when the number of nodes increases (x-axis), bytes per node (y-axis) increases.

6.5 Conclusion

In this chapter we defined the problem of frequent itemset mining. We gave a detailed analysis of the FP-growth algorithm along with the expected requirements for the data structures to be able to execute this algorithm. Furthermore we proceeded with detailed analysis of the classic implementation by Han et al. [34] and the efficient approach by Schlegel et al. [51]. Given that the minimum space bound for the trie representation is $N(\log \sigma + O(1))$, the data structures proposed by Schlegel et al. (space $\Theta(N \log N)$ bits for both) have room for improvements. Finally, we identified the weaknesses of those data structures. In Figure 6.2, we proved that space usage increases proportionally with the number of nodes ranging from 5 bytes per node to 9 bytes per node. In the next chapter we will propose alternative solutions to such weaknesses.

Chapter 7

Compact implementation of FP-growth

As we described in Chapter 6, representing data in memory for the FP-growth algorithm is a challenge. We recall that the terminology used in Chapter 6 and especially the definition in Section 6.1 is used by this chapter as well. The approach proposed by Schlegel et al. [51] (CFP-growth) that we described in the previous chapter, uses two data structures one each for the build phase (CFP-tree) and mine phase (CFP-array). CFP-growth approach manages to reduce the memory requirements by nearly an order of magnitude over the standard approaches to FP-Growth [34]. However, as explained in Section 6.4, their heuristic approaches to reduce memory are not always effective. In summary:

- CFP-tree is based on Patricia compression: the effectiveness of Patricia compression is data-dependent making the space usage of CFP-tree unpredictable and inconsistent.
- CFP-tree uses 40-bit pointers, which limits the addressable memory to 2^{40}.
- CFP-array is a 64-bit implementation but there are worst case scenarios proving that the CFP-array space usage increases with the number of nodes.

This chapter will proceed as follows. Firstly, we give a summary of our contributions for all three phases build, mine and conversion phase. In Sections 7.3 to 7.5, we analyze the data structures used for the three phases. Furthermore, we proceed with the implementation details and finally in Section 7.7, we proceed with the

experimental analysis where we compare the memory usage and the runtime speed of our approach with the CFP-growth [51].

7.1 Our contributions

In this chapter we propose *Piccolo* FP-Growth or *PFP-Growth*, which is based on compact and succinct data structures. PFP-growth uses significantly less space than the approach of Schlegel et al. [51] on benchmark data sets. As explained in Section 2.3.1, the SDS come with mathematical guarantees on space usage, leading to predictable memory performance, and also give competitive speed performance. In addition, we propose a (novel) SDS developed for PFP-growth which may have wider applications. As with the approach of Schlegel et al., PFP-growth also has separate data structures – one each for the build phase and mine phase – which results to the additional conversion phase.

Build phase. For the build phase we propose the *Bonsai* FP-tree or BFP-tree. BFP-tree is a TST representation based on m-Bonsai tree implementation (Chapter 5). The space usage of BFP-tree is $(1+\epsilon)N\log\sigma + O(N\log^{(5)} N)$ bits, for any constant $\epsilon > 0$, which is close to the ITLB for representing tries of N nodes and alphabet σ. In practice for the last term to be impactful on the space usage is when $N > 2^{256}$ (also explained in Section 5.6.2). Therefore, the "per-node" space cost is heavily dependent on σ, whereas CFP-tree nodes use $O(\log N)$ bits.

A major difficulty with using m-Bonsai tree directly, which the BFP-tree overcomes, is that m-Bonsai tree *only* permits efficient root-to-leaf and leaf-to-root traversals. Our approach requires to traverse the build phase data structure and construct the mine phase data structure during the conversion phase (peak memory usage). Using the approach in Section 5.4, to traverse the tree we would require extra $O(N\log\sigma)$ bits, which turns out to be very expensive for this application. In addition, the BFP-tree stores the itemIDs separately in an array, whereas for

m-Bonsai they are part of the key. Therefore, we have the freedom to compress itemIDs as Δ-items which proves to be very effective in CFP-growth approach. In principle, we can use a CDRW-array from Chapter 4.

The BFP-tree's memory usage is upto 2 times less than the CFP-tree on the same benchmark datasets considered by Schlegel et al. It is upto 3 times less than our conservative estimate of the space usage of a full 64-bit implementation of the CFP-tree.

Mine phase. For the mine phase we propose a novel succinct data structure, the *Layered FP-tree* (*LFP-tree*). The LFP-tree also represents the tree structure of the FP-tree. However, the mine phase of FP-growth requires different operations to the build phase. Primarily we are required to efficiently enumerate *all* nodes with the same itemID (this is accomplished by the use of the *node-link* pointers in the FP-tree). We are not aware of any SDS for tries that supports this operation efficiently (in time that is linear in the number of nodes enumerated).

The space usage of an N-node LFP-tree is at most $N \log \sigma + O(N)$ bits; the per-node space usage is again independent of N. The LFP-tree's memory usage is consistently 2.5 to 3 times less than the CFP-array on the same benchmark datasets considered by Schlegel et al.

Conversion phase. At a high level, the conversion process is similar to CFP-growth. We traverse the BFP-tree and construct the LFP-tree. However, there are many challenges. LFP-tree is an Elias-Fano representation which is primarily a static representation (Theorem 1). To be able to keep the memory usage low during the conversion phase, we build the LFP-tree dynamically.

We show that the peak memory usage of PFP-growth is upto 2 times less than the CFP-growth on the same benchmark datasets considered by Schlegel et al. and it is upto 2.8 times less than our conservative estimate of the space usage of a full 64-bit implementation of CFP-growth.

7.2 PFP-growth

We now explain how the PFP-growth algorithm proceeds through the three phases. During the build phase, we need to read the dataset. The items in each transaction are sorted according to their support. We read one transaction at a time and we insert it in the BFP-tree.

Next, we move to the conversion phase. We dynamically construct the LFP-tree while traversing the BFP-tree in linear time. To do this, we traverse the tree in an order which is a combination of breadth-first and depth-first traversal. Although our approach takes linear time, we have to store nodeIDs in a collection of queues, which in the worst case is no more than an extra word per transaction.

Finally, for the mine phase, the LFP-tree is able to project conditional pattern base FP-trees of itemsets. However, it was not immediately obvious how to rebuild new LFP-trees for each conditional pattern base and do it fast. Given that the conditional pattern base FP-trees are very small compared to the main LFP-tree which holds the entire database, we can use the LFP-tree as the core data structure for the mine phase which can project conditional pattern bases into other implementations of FP-tree. For the conditional FP-trees we use the implementation by Bart Goethal available in [26] and extensively benchmarked in [27]. Finally, we can compare it with CFP-array implementation. Note that we could as well use the CFP-array for the conditional FP-trees.

7.3 BFP-tree

We now describe the BFP-tree which is the data structure used for the build phase. This data structure is a TST using the m-Bonsai(r) implementation from Section 5.6.2. Given a sorted (according to the support of the items) transaction T, a node with itemID $T[i]$ will have a child with itemID $T[i+1]$. This would be ideal for m-Bonsai(r) as we could use the `addLeaf` operation for each item in the transaction.

However, we proceeded with modifications because during the conversion phase we need to traverse the BFP-tree as space efficiently as possible to construct the LFP-tree. The traversal of m-Bonsai(r) (Section 5.4) is using $O(N \log \sigma)$ additional space which is a serious problem for this application. As we already mentioned, BFP-tree is a TST representation: the nodeID instead of containing the itemID of a node, it contains the "direction" of a pointer. The direction in a TST can only be one of the three {left sibling, right sibling, suffix}. This makes the quotient size of each entry in the Q array 2 bits long. Furthermore, we need to store the actual itemIDs in a new array I of size $(1+\epsilon)N\lceil \log \sigma \rceil$ bits, such that $I[i]$ has the itemID of node with nodeID $= i$. As we mentioned before, we could use a CDRW-array to store the itemIDs as Δ-items. This could in principle reduce the memory usage especially for datasets with large σ like Retail and Webdocs. Unfortunately, we did not have time to implement it, therefore array I is of fixed size entries of $\lceil \log \sigma \rceil$ bits. The asymptotic space usage of BFP-tree is: $(1+\epsilon)N(\log \sigma + O(1))$ bits.

pcounts: We now outline how we store the pcount information in the BFP-tree. We store a sequence of M pcount values where i-th pcount value is associated with the node with nodeID $= i$. We use the same approach with the one used in Section 5.3.2 to store the displacement values for m-Bonsai(r). Again the parameters Δ_0 and Δ_1 define in which layer we store the pcounts. The data structure again makes use of three layers: the first layer stores values upto $2^{\Delta_0} - 2$. The second layer is a CHT which stores the pcount values ranging between $2^{\Delta_0} - 2$ and $(2^{\Delta_1} + 2^{\Delta_0} - 3)$ as satellite data and `std::map` for pcounts that need even more space.

BFP-tree ADT. We now give the ADT of the BFP-tree which is followed by a recap on the advantages and disadvantages of BFP-tree over m-Bonsai(r).

- getItemID(v): return itemID of a specific node v in $O(1)$ time.
- incrementCount(v): increment count of node v by one in $O(1)$ time.

- getSuffix(v): return suffix of a node v with itemID i in $O(1/\epsilon)$ time (and return null if no such child exists).
- getLeft(v): return left sibling of node v with itemID i in $O(1/\epsilon)$ time (and return null if no such child exists).
- getRight(v): return right sibling of node v with itemID i in $O(1/\epsilon)$ time (and return null if no such child exists).
- addLeaf(v, i): add a new leaf with itemID i and return the node number of the newly created node in $O((\log \sigma)/\epsilon)$ time.
- getChildren(v): return a list of nodeIDs of all children of node v in $O(x)$ time, where $x \leq \sigma$ is the number of children of node v (return empty list if v is a leaf).

In summary, the modifications have given BFP-tree some advantages and disadvantages compared to m-Bonsai(r). The disadvantages include: (a) Even though the asymptotic space usage is the same, in practice BFP-tree uses $(1 + \epsilon)2N$ bits mores space. (b) The addLeaf operation for the BFP-tree takes logarithmic time whereas for the m-Bonsai(r) takes $O(1/\epsilon)$ time. The advantages include: Additional operations like the ones allowed by TST (getChildren, getLeft, getRight, getSuffix). No extra space is required for the traversal. No extra space required to be able to determine if a node is a leaf. Finally, we can potentially use Δ-encoding on itemIDs and store them in a CDRW-array from Chapter 4.

7.4 LFP-tree

In this section we first describe a simplified version of the LFP-tree. Then we describe how we store the count information. Next we give the ADT of the LFP-tree which is followed by implementation details and examples based on Figure 7.1.

We begin by storing a logical bit-vector Z which has exactly N 1 bits. Each 1 bit represents a node in the FP-tree. Given that $\Sigma = \{1, \ldots, \sigma\}$, Z is the concatenation of a series of bit-strings called *zones*, denoted as $Z = z_0 z_1 \ldots z_\sigma$, where $z_0 =$"1" by definition. z_0 represents the root and is followed by σ zones where the 1s in each z_i represent the nodes with itemID i. Assume by induction that $z_0 \ldots z_{i-1}$ (and hence all nodes with itemIDs $< i$) have been created. For $i > 0$, z_i is created to be of length equal to the number of 1s in $z_0 \ldots z_{i-1}$ as follows. The parent of any node with itemID i must be the root or a node with itemID $< i$ and the j-th bit in z_i is set to 1 or 0 depending on whether or not the j-th node (or j-th 1) in $z_0, \ldots, z_{i-1}$ has a child with itemID i or not.

Since for $i \geq 2$, $|z_i| \leq N$, Z has length at most $(\sigma - 1)N + 2 \leq \sigma N$ bits (since $N \geq 2$ always) and has exactly N 1s. We number the nodes in the LFP-tree $1, \ldots, N$ (we also refer to it as the *node number*) in the order that the 1 bits representing them appear in Z. Observe that all nodes with the same itemID are consecutively numbered. More generally, let the *upward prefix* of a node be the sequence of itemIDs obtained by starting from the node and going to the root: it can be seen that nodes are numbered in sorted order of their upward prefix. In addition to the bit-vector Z we also store a bit-vector B of length N that (in essence) stores the number of nodes N_i with itemID i in unary, i.e. $B =$unary(N_0)unary$(N_1) \ldots$ unary(N_σ). Finally, we have an array A, with which we can determine the start of each zone in Z of size $(\sigma + 1) \log N$ bits.

Δ-count For the mine phase, we are not able to use pcounts. The count of a node is the sum of all the pcounts of the descendants of that node. For the mine phase, we traverse towards the root and we cannot get the count of a node fast. Therefore, during the conversion phase, while BFP-tree is traversed we construct the Δ-counts for the LFP-tree. In an FP-tree the most frequent nodes are closer to the root than the least frequent ones. Therefore, it is always the case that parent's count will be greater than or equal to a child's count. Instead of storing the count we store the

difference from the parent. This helps especially for nodes in lower-level that tend to have the same counts. The Δ-counts are stored in a CDRW array (Base approach) from Chapter 4. The CDRW array is of size N where the i_{th} value is Δ-count of the i_{th} node in Z.

LFP-tree ADT. We now give the ADT of the LFP-tree.

- $\mathsf{getItemID}(v)$: return the itemID of node v in $O(1)$ time.
- $\mathsf{getCount}(v)$: return the Δ-count of node v or -1 if node does not exist in $O(1/\epsilon)$ time.
- $\mathsf{getParent}(v)$: return parent node of node v in $O(1)$ time.
- $\mathsf{listNodes}(x)$: List all the nodes with itemID x in linear time to the size of the list.
- $\mathsf{project}(D_s, x)$: Given a conditional pattern base for s, and an item $x < \min(s)$, returns the conditional pattern base FP-tree for itemset $x \cup s$, in $O(N_{x \cup s})$ time where $N_{x \cup s}$ is the size of the new FP-tree.

We now proceed with implementation details that will give a better intuition of how LFP-tree would be able to function during the mine phase. Then we proceed with examples based on Figure 7.1.

We start with the $\mathsf{getItemID}(v)$ operation. Bit-vector B allows us to support the $\mathsf{getItemID}(v)$ in $O(1)$ time since we can perform the rank_1 operation on B. More precisely, to get the itemID of node v we do $\mathsf{rank}_1(B, v+1)$. Since the itemID of the current node v is known we are able to perform the $\mathsf{getParent}(v)$ operation in $O(1)$ time as well. Since z_i contains all nodes with itemID i, the node number of the parent is the position of v within z_i. Thus, we can support $\mathsf{getParent}(v)$ as $\mathsf{select}_1(Z, v+1) - A[\mathsf{getItemID}(v)]$.

In Figure 7.1, there is an example of Z and B bit-vectors which are representing a projected database from itemset "*d*". In (a) we see the logical view of the FP-tree

where the nodes are numbered (in brackets) according to the order they are in Z from left to right. The 1-bits inside the black area of Z represent nodes in the FP-tree. At the bottom part of each 1-bit is the node's node number (note they are the same as in the brackets in (a)) and on top of 1-bits it is the parent node number. For example, the 5-th 1-bit, has itemID c as it is in the fourth zone, its node number is "(4)" and its parent is node with node number "(1)". Each zone z_i has a total number of bits equal to the total 1-bits from z_0 to z_{i-1}. Another way to look at Z, for example: the 3-rd bit in zone c has as parent the 3-rd node (or 3-rd 1) in Z. Furthermore, each zone in bit-vector B is the count of 1-bits in the relative zone in Z. To obtain $\mathsf{getItemID}(7)$, we first get $\mathsf{rank}_1(B, (7+1)) = 4$ meaning that the node lies in z_4 with the relative itemID $= d$ counting from zero (the root). To obtain $\mathsf{getParent}(7)$ we firstly do $\mathsf{select}_1(Z, 7+1) = 10$, then get the starting position of z_4 which is $A[4] = 9$ counting from one. Finally, we do $\mathsf{select}_1(Z, 7+1) - A[4]$ to return the parent node number, which is 1.

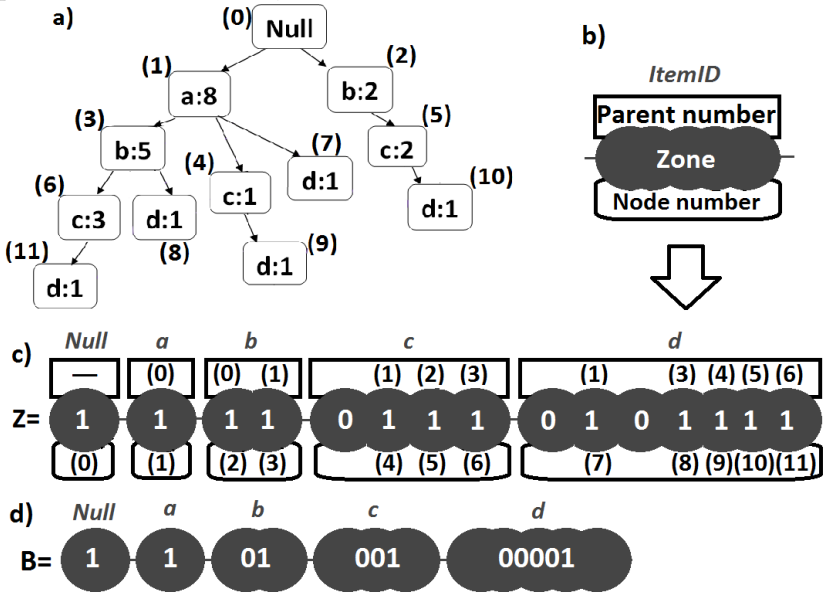


Figure 7.1: (a) Logical view of FP-Tree with numbered nodes (0) to $(N-1)$. (b-c) Logical view of bit-vector Z: Each 1 is a node that has a node number and an associated parent number. (d) Bit-vector B, encodes in unary the number of nodes in each zone.

As we mentioned above, this is a logical view of bit-vector Z. Implementing Z in this way is not space-efficient as it requires $O(N\sigma)$ bits. Therefore we try to

succinctly represent Z using the locations of 1s in Z. Clearly, the locations are in increasing order. The maximum potential location is $m = N\sigma$ and the number of nodes is N. As we see in Theorem 1, with the above setting it would be ideal to succinctly represent Z in Elias-Fano encoding. This would make the space usage for Z equal to $2n + n\log(m/n) + o(n)$ bits where we could access the i-th integer in $O(1)$ time using the select_1 operation. However, Elias-Fano encoding is a static representation. It would not be space efficient if we store the Z bit-vector ($O(N\sigma)$ bits) in memory and then compress it using Elias-Fano encoding. Therefore, we show a way to dynamically insert the nodes in LFP-tree representing them in Elias-Fano. This will be explained in detail in the next section.

7.5 Conversion

As we mention at the start of this chapter, the conversion of BFP-tree to LFP-tree is required before we proceed with the mining. LFP-tree is an Elias-Fano representation. The nodes of the BFP-tree must be traversed in the order they are numbered in Figure 7.1-(a). This would help in dynamically inserting the nodes in LFP-tree by filling the zones of Z from left to right.

We now explain how we prepare Elias-Fano representation of Z to dynamically insert nodes in it. We recall that $Z = z_0 \dots z_\sigma$ where z_i is the i^{th} zone in Z representing the nodes with itemIDs i and z_0 is the root. Using array A we are able to get the size of each $|z_i| = m_i$. At the same time, $\mathsf{rank}_0(\mathsf{select}_1(B, i)) = N_i$, which is the number of nodes in z_i. Since we have m_i and N_i for each zone we are able to have a different Elias-Fano representation per zone. According to Theorem 1 the total space is $\sum_{i=0}^{\sigma} N_i(\log(m_i/N_i) + 2)$ bits. Based on Jensen's inequality [42] our approach can potentially achieve better compression than representing the whole Z bit-vector with Elias-Fano as $\sum_{i=0}^{\sigma} N_i(\log(m_i/N_i) + 2) \leq N(\log(m_\sigma/N) + 2)$.

After we create the space required for each zone in LFP-tree, we now explain how we traverse BFP-tree to dynamically insert the nodes in them. The nodes of

the BFP-tree must be traversed in the order they are numbered in Figure 7.1-(a) this is because we can dynamically insert nodes in LFP-tree by filling the zones from left to right. To do this in linear time we need to store at most X_{max} nodeIDs, which we define below. Given an FP-Tree, let X_c be the number of distinct <*parent, child*> tuples where parent itemID $< c$, and child itemID is $\geq c$. Now let X_{max} be the maximum number of tuples at any given itemID, thus X_{max} is bounded by the number of distinct transactions in D. In practice, based on our benchmark datasets $X_{max} \simeq 0.01N$.

Algorithm 4 Conversion Phase

```
1: function CONVERSION(BFP bfp)
2: new LFP();                                          ▷ Initialise A and B
3: new XQueue[σ][0];                                   ▷ XQueue to store nodeIDs.
4: XQueue[0].push(rootNodeID);
5: for(i ← 0; i < σ; ++i)                              ▷ through all zones
6:     for(j ← 0; j <rank_0(select_1(B, i)); ++j)      ▷ through nodes per zone
7:         bfp.getChildren(XQueue[i][j]);
8:         updateXQueue();                             ▷ push nodeIDs of children in XQueue.
9:         XQueue[i].pop();
10:    end for
11:    delete XQueue[i]
12: end for
```

7.6 Implementation details

In this section, we will be giving the implementation details of our approach. Our implementations were developed in C++. The constructor is taking θ which is the support threshold. Then the process starts as we already mentioned, build phase followed by conversion phase and then mine phase.

7.6.1 Build phase: BFP-tree

The items in each transaction are sorted according to their support. All the single items that are $< \theta n$ are pruned, recall that n is the number of transactions.

We read one transaction at a time and we insert it in the BFP-tree. As in all of our implementations we used sdsl-lite library containers `sdsl::int_vector` and `sdsl::bit_vector`. The BFP-tree is very similar to m-Bonsai(r) implementation from Section 5.6.2. The choice of direction in the hash function {left sibling, right sibling, suffix} is by using the numbers $\{0, 1, 2\}$ respectively. The array I is an `sdsl::bit_vector` which allows us to choose the width of the entries based on σ.

pcounts We now discuss the choice of parameters Δ_0 and Δ_1 for the pcount representation. The values of Δ_0 and Δ_1 are currently chosen numerically since there is no mathematical guarantee of how pcounts would behave for each dataset. We see in practice based on the FIMI datasets that there is a very big percentage of nodes with pcount values of $\{0, 1, 2\}$, $> 99.8\%$. Therefore, by choosing $\Delta_0 = 2$ and $\Delta_1 = 5$ we have a small memory footprint of $2.15 - 2.30$ bits per node, based on the FIMI datasets.

7.6.2 Conversion phase: LFP-tree consruction

For the conversion phase, we traverse the BFP-tree as we explained in the Algorithm 4 and build the LFP-tree. We now explain how we initialize the LFP-tree to be able to insert the nodes without rebuilding. We recall that we create a different Elias-Fano encoding (Theorem 1) for each zone z_i where m_i is the maximum integer to be represented (or maximum prefix sum, Section 2.1.1) and N_i is the number of nodes for each zone. As with the proof of Theorem 1, every z_i is split in two parts top_i and bot_i. The top part of each zone is an `sdsl::bit_vector` of size $2N_i$ bits. The bot part is an `sdsl::int_vector` as it requires N_i fixed size entries of size $\lceil \log(m_i/N_i) \rceil$ bits each. The way we traverse the BFP-tree allows us to append nodes from left to right into each zone. Therefore, we append the values in the offset positions of top and bot parts of each zone.

As we explained in the proof of Theorem 1, we can get the j-th value of an Elias Fano encoding by extracting and combining the jth value of both top and bot parts.

The j-th value in the top part can be taken using the select_1 operation in $O(1)$ time. Each top_i is supported by `sdsl::select` and each bot_i part is just an `int_vector` that can access the j-th value in $O(1)$ time. Then we concatenate top and bot part to get the node number of that node. After this it is straight forward as the node number of the parent of the j-th node in z_i is $z_i[j] - A[i]$. And the itemID of the parent can be computed using $\mathsf{rank}_1(B, z_i[j] - A[i])$.

7.6.3 Mine phase: LFP-tree projection.

The LFP-tree uses the project operation as follows. Firstly, we take all the nodes with the same itemID. Each node has a different prefix path towards the root. We use the getParent operation to follow the prefix path. While we do this we temporarily store in a list the itemIDs of each node in the path. In addition, we sum up the Δ-counts as we move upwards to get ℓ which is the count of the node we initially started with. Then, we reverse the list with the itemIDs and insert each item in a new FP-tree (Goethals implementation [26]) where each node will increment its count by ℓ. Note that the list of itemIDs is used in the same way as a single transaction in the build phase. The Goethals implementation [26], is based on the classic implementation proposed by Han et al. [34]. It uses a TST as we explained in Section 6.3. The new FP-tree prunes the infrequent items according to the updated counts and recursively projects new even smaller FP-trees. At the end of each project operation we return the frequent itemsets. To do this for all itemsets we usually start from the least frequent item and recursively repeat the process for all items.

7.7 Experimental evaluation

7.7.1 Datasets

We perform experiments to study the performance of our data structures. Our goal was to precisely calculate the performance of our approach in terms of memory

Datasets	File Size	Node number	σ	Transactions	Avg. trans. length
Chess	342 KB	38609	75	3196	37
Mushroom	570 KB	27348	119	8124	23
Connect	9 MB	359291	129	67557	43
Pumsb	17 MB	1125375	1734	49046	73
Accidents	36 MB	4242317	290	340183	33
Retail	4 MB	653217	8919	87789	10
Webdocs	1.4 GB	231232676	59717	1690527	163

Table 7.1: File properties of FIMI datasets

usage using real life datasets. The datasets used follow the standard FIMI format [25], where each line is a different transaction and items are in the form of integers and are separated by space. The datasets have different characteristics, as shown in Table 7.1, including σ, average transaction length, number of transactions

7.7.2 Experimental setup

The code was compiled using g++ 4.7.3 with optimization level 6. Our implementations are using the sdsl-lite library [29] containers as in previous chapters. The machine used for the experimental analysis is an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz with 3MB L2 cache, running Ubuntu 12.04.5 LTS Linux. To measure the resident memory (RES), `/proc/self/stat` was used. For the speed tests we measured wall clock time using `std::chrono::duration_cast`.

7.8 Benchmarks

7.8.1 Space usage

In this section, we run the experiments on the FIMI datasets. We go through build mine and conversion phase. Therefore, we can compare the space usage for PFP-growth, CFP-growth and CFP-growth(x64). CFP-growth(x64) is the Schlegel et al. approach for our conservative estimate of a full 64-bit implementation of CFP-tree.

Build phase For the experiment shown in Figure 7.2, we compare the BFP-tree against both CFP-tree and CFP-tree(x64). As expected BFP-tree space usage is more consistent and more predictable than the CFP approaches. The memory usage of BFP-tree is dependent on the alphabet size; this is the reason why BFP-tree uses more memory per node on Webdocs compared to Chess dataset. On the other hand, the space usage of the CFP-tree approaches is unpredictable as it ranges from 18 bits per node to 62 bits per node.

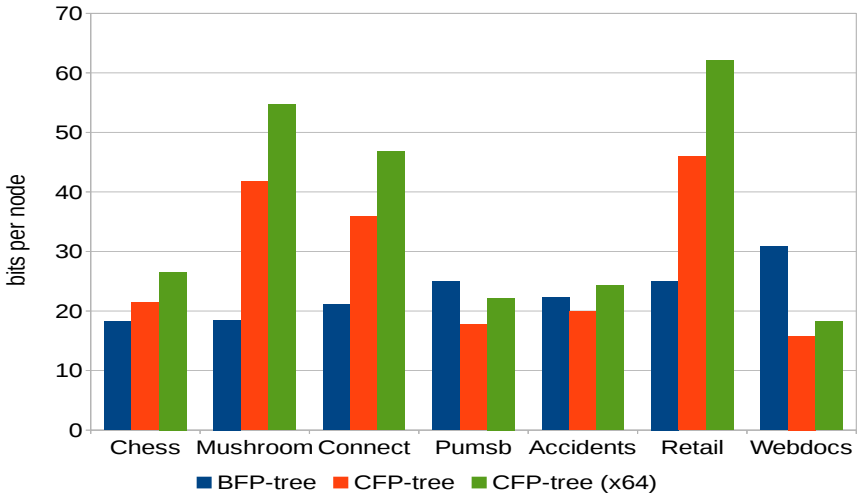


Figure 7.2: We compare the memory usage between BFP-tree, CFP-tree and CFP-tree ($\times 64$).

Mine phase In Figure 7.3, we compare the memory usage for the mine phase data structures. The CFP-array approach is more stable relative to the CFP-tree. The CFP-array is a 64-bit implementation, therefore we just compare the LFP-tree with the normal CFP-Array. As shown in Fig. 7.3, the space usage of the LFP-tree is consistently performing $2-3$ times better than the CFP-Array.

Conversion phase (peak memory) For this experiment, we measure the memory usage during the conversion phase while both build and mine phase data structures are stored in memory. As we can see Figure 7.4, the space usage of PFP-growth is predictable, and consistently better than CFP-growth (upto 2.5 times). The CFP-growth is affected by: (1) the CFP-tree being unpredictable, (2) the relatively high

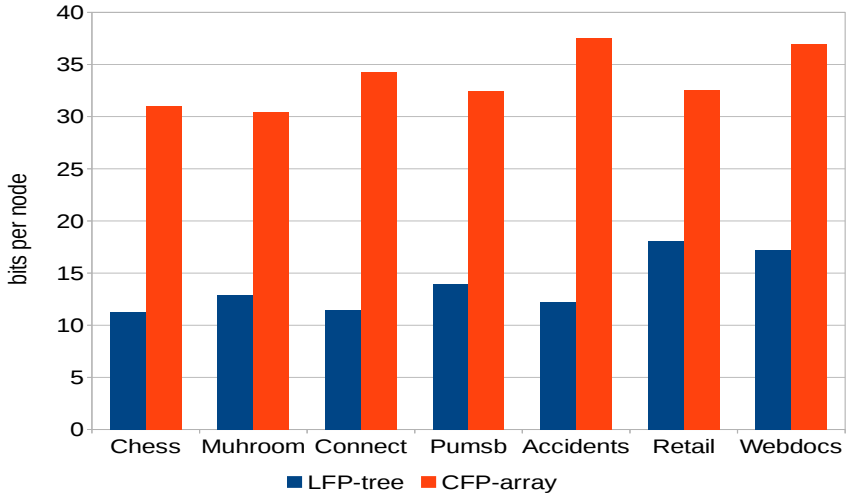


Figure 7.3: We compare the memory usage between LFP-tree and CFP-array.

memory usage of the CFP-array and (3) the 64-bit pointers which may lead to a range of $50 - 90$ bits per node.

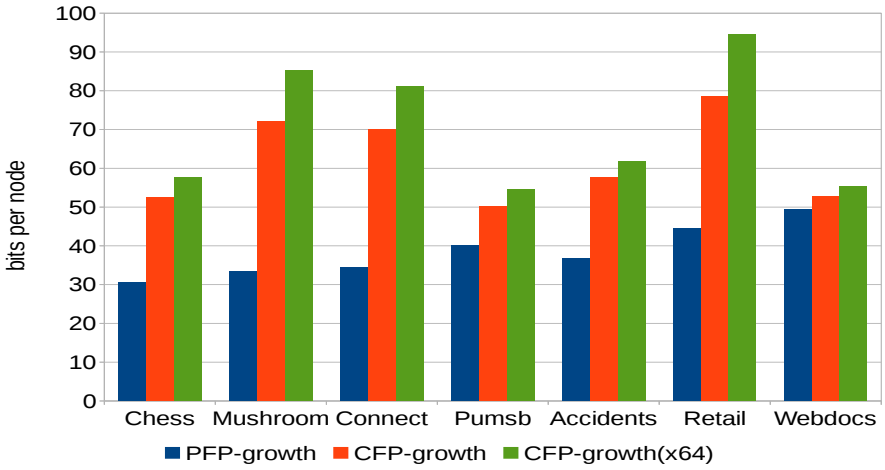


Figure 7.4: We compare the peak memory usage between PFP-growth and CFP-growth.

7.8.2 Runtime speed

In this section we compare the wall clock time required for the build, mine, and conversion phase by the two approaches.

Build phase In this experiment we measure the wall time required to build the FP-tree. As shown in Figure 7.5, we see that there are cases where the BFP-tree performs better than CFP-tree and the other way around. For the small datasets (Chess and Mushroom), the BFP-tree performs around 10 times better than CFP-tree. We can also see that for Retail dataset which is clearly the most bushy dataset we have (average transaction length 10 and $\sigma = 8919$) performs slightly better. On the other hand, on some datasets we notice the benefit coming from Patricia compression as there are less memory accesses due to the chains. For example, Webdocs has long chains (average transaction length = 175), where with only one or two accesses it can pass through a lot of nodes. Therefore, for datasets like Webdocs we can see that CFP-tree performs upto 4 times better.

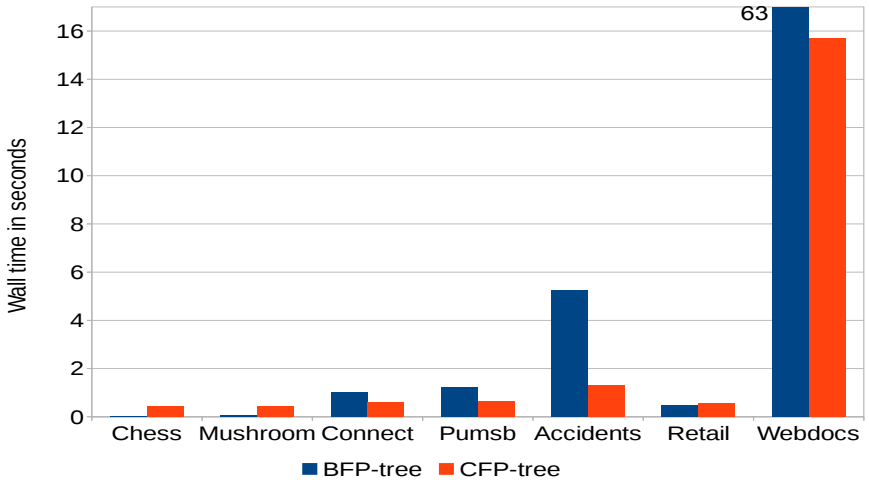
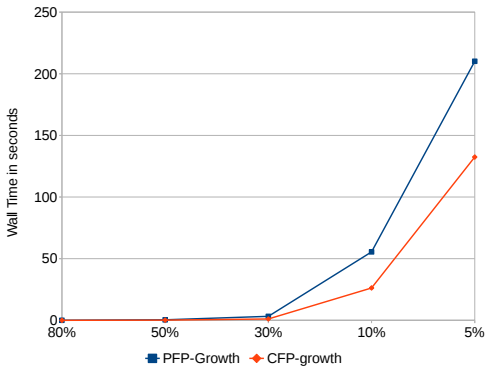
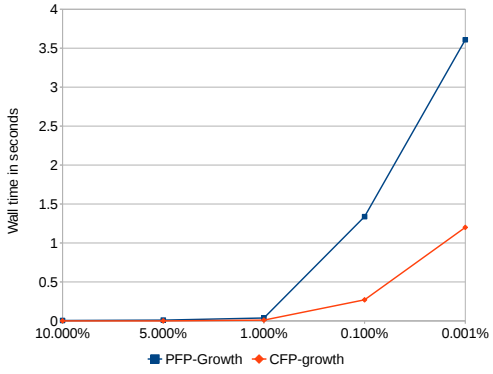


Figure 7.5: We measure the runtime (in seconds) required for the build phase for both BFP-tree and CFP-tree.

Mine phase In this experiment, we selected two datasets from the FIMI repository [25], Accidents and Retail. We measure the wall time required to identify all frequent itemsets on a decreasing support thereshold θ. For the PFP-growth approach, we used the Goethals implementation to project the conditional pattern base data structures. We compare this approach with the CFP-growth implementation. We see that PFP-growth approach is initially (at high θ) very competitive with CFP-growth approach for both data structures. As θ gets lower the gap in time



(a) Mining time in seconds of Accidents dataset on decreasing support threshold(%)



(b) Mining time in seconds of Retail dataset on decreasing support threshold(%)

Figure 7.6: Mining runtime tests

Datasets	PFP-growth	CFP-growth
Chess	0.044	0.038
Pumsb	1.21	0.042
Accidents	5.25	1.121
Retail	0.483	0.04
Webdocs	72.43	11.231

Table 7.2: Conversion time in seconds for PFP-growth compared to the CFP-growth.

increases, the CFP-growth is upto 2 times faster on Accidents and 3 times faster on Retail. We recall that PFP-growth could as well use any approach to project the datasets from the LFP-tree.

Conversion phase During conversion phase both CFP and PFP approaches need to traverse the build phase data structure and at the same time construct the mine phase data structure. In this experiment we measure the time required to traverse the tree in seconds. As shown in Table 7.2, it is clear that the CFP-growth conversion is upto 6.5 times faster than the PFP conversion. Note that the attempted conversion was on 0.001% support threshold. Compared to the mine phase runtime, conversion phase is much smaller (15-40 times) than the mine phase which is not considered very impactful. This is also shown with respect to the results in Fig. 7.6.

7.9 Conclusion

In this chapter, we investigated the FP-Growth algorithm under the compact implementation of two data structures, BFP-Tree and LFP-Tree. The standard solution would be to have one TST data structure for both phases. We have presented a significant space reduction compared to the state of the art implementation, the CFP-growth. Even though CFP-growth shows excellent runtime performance, PFP-growth is more compact, consistent and predictable under any different (property-wise) datasets. In addition, we show excellent space usage during the conversion phase which is the peak memory usage. More precisely, the memory usage is reduced by upto 2.5 times from the CFP-growth and 30 times from the original FP-tree implementation. Even though, the conversion time of PFP-growth is upto 6.5 times slower than CFP-growth, it proves to be insignificant as it is upto 40 times smaller than the mine phase runtime. In Figure 7.6, we show that CFP-growth is $2-3$ times faster during the mine phase. Furthermore, PFP-growth proves to be the ideal test case for all the compact dynamic data structures (modified or not) proposed in this thesis. We show ideas like dynamic insertion of values in Elias-Fano encoding, proposing (as far as we know) a novel succinct data structure, the LFP-tree.

Chapter 8

Conclusion

Nowadays, people are misled to believe that memory is getting cheaper by the years. As a result, one can easily underestimate the value and importance of efficient memory usage or compact memory representation. As it is mentioned in the introduction data is accumulated with high velocity, leading to large volumes of raw unprocessed data or partially processed data. Processing or mining data under these conditions requires more and more memory. It can be argued that memory is cheap but at the same time data is growing so rapidly. During this journey, I was able to understand the importance around the efficiency, and compactness of data structures. Not every company or every person owns clusters of servers and most certainly only high budget companies can afford renting machines in the cloud for processing big data. Consider for instance, processing hundreds of genomes on a researcher's personal computer or companies that are unable to process 5GB XML files on servers of 80GB of RAM. When such problems occur people are limited to alternative processing strategies like processing data a piece at a time and so on. In most cases the quality is impacted which may lead to wrong conclusions.

Compact and succinct data structures are data structures that can provide solutions to such problems. However, sometimes the weaknesses of available implementations could be a "turn off" for users. As we have seen in this thesis, the space savings of such data structures is humongous compared to traditional approaches. Therefore, any weakness that could be pointed out whether that is speed or structure flexibility; with some extra engineering, there is orders of magnitude space available

to overcome such weaknesses and still use less memory than traditional approaches. These data structures not only provide results in terms of memory and speed but are also confined with compelling ideas that could be used as solutions to a variety of problems in computer science.

Summary. In this thesis, we proposed a number of compact dynamic data structures. We introduced the problem for the implementation of compact dynamic rewritable (CDRW) arrays, and gave 3 different implementations of CDRW arrays: Base, Base IL and DFR. Part of the implementation of the CDRW arrays, a key data structure was the compact hash table (CHT). Not only have we (re)-confirmed that CHT approach is very fast and space-efficient on modern architectures, we implemented the delete operation which was not described in [11]. The delete operation allowed us to "realloc" bit-strings in the CDRW array. Along with some heuristic optimizations the CDRW arrays show excellent performance for both space and time.

Furthermore, we took a closer look to Bonsai, which is a compact dynamic representation of tries. We proposed improved variants of Bonsai implementation, m-Bonsai(γ) and m-Bonsai(r). m-Bonsai(γ) is sufficiently more space efficient than Bonsai but slower in terms of speed. m-Bonsai(r) is consistently better for both space and speed than original Bonsai and slightly worst in space than mBonsai(γ). We gave new solutions including how to traverse and extend the tree in linear time.

Finally, we give an efficient solution for the FP-Growth algorithm. We proposed PFP-Growth, which is using the m-Bonsai and CDRW arrays. In addition, we implement a novel SDS called LFP-tree. The LFP-tree represents a tree data structure that is able to efficiently enumerate all nodes with the same itemID/satellite data. We are not aware of any SDS for tries that supports this operation efficiently (in time that is linear in the number of nodes enumerated). PFP-growth performance is competitive in terms of speed and achieves significant reductions to the memory usage, compared to space efficient solutions like CFP-growth.

Future work. The main drawback of CHT is that it is of fixed size. To be able to resize the CHT we would need to rehash the entire hash table. This restricts the CHT to uses where we know the number of elements to be hashed. It would be interesting direction to be able to implement the CHT data structure such that it can extend/shrink without affecting the amortized time for the insert and delete operations. We believe that this thesis would benefit substantially by an efficient implementation of an extensible CHT. CDRW-arrays would be truly dynamic and be used on more applications.

Moving on to the CDRW arrays. The asymptotic results are not the main thrust of the CDRW arrays which indicates that there is room to study this problem in further detail. For example, there is no obvious reason why a theoretical solution cannot be found that uses $S + o(S)$ bits of space and supports operations in $O(1)$ time. An interesting future direction is to tailor CDRW arrays to particular distributions of the values from the source, or better yet to design CDRW arrays that adapt to the distribution to optimize space and time.

Finally, we describe potential improvements on the PFP-growth. The most obvious step is the use of Δ-items for the BFP-tree. Dynamizing the LFP-tree would be a very interesting future direction as it has excellent performance in terms of memory and is a unique SDS that can enumerate all nodes with the same itemID in time that is linear with the number of nodes enumerated.

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[2] Jun-Ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Software Eng.*, 15(9):1066–1077, 1989.

[3] Niranka Banerjee, Sankardeep Chakraborty, and Venkatesh Raman. Improved space efficient algorithms for BFS, DFS and applications. *CoRR*, abs/1606.04718, 2016.

[4] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[5] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009.

[6] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[7] Jon Bentley and Bob Sedgewick. Ternary search trees. `http://www.drdobbs.com/database/ternary-search-trees/184410528`, 1998.

[8] Daniel K. Blandford and Guy E. Blelloch. Compact dictionaries for variable-length keys and data with applications. *ACM Trans. Algorithms*, 4(2):17:1–17:25, 2008.

[9] John F. Canny and Huasha Zhao. Big data analytics with small footprint: squaring the cloud. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 95–103, 2013.

[10] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[11] John G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

[12] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 241–252. ACM, 2003.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[14] John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper.*, 23(3):277–291, 1993.

[15] René de la Briandais. File searching using variable length keys. In *Proc. Western J. Computer Conf.*, pages 295—-298, 1959.

[16] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Patrascu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 349–361. Springer, 2006.

[17] Martin Dietzfelbinger. On randomness in hash functions (invited talk). In Christoph Dürr and Thomas Wilke, editors, *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, volume 14 of *LIPIcs*, pages 25–28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[18] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[19] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, 21(2):194–203, 1975.

[20] Irv Englander. *The Architecture of Computer Hardware and System Software: An Information Technology Approach*. John Wiley & Sons Software, 4th edition, 2009.

[21] Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[22] R.M. Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971.

[23] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, pages 271–282, 2003.

[24] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[25] Bart Goethals. Frequent itemset mining implementations repository. `http://fimi.ua.ac.be/`.

[26] Bart Goethals. Frequent pattern mining implementations. `http://adrem.ua.ac.be/~goethals/software/`, 2003.

[27] Bart Goethals. Survey on frequent pattern mining. Technical report, 2003.

[28] Bart Goethals and Mohammed Javeed Zaki. Advances in frequent itemset mining implementations: report on FIMI'03. *SIGKDD Explorations*, 6(1):109–117, 2004.

[29] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 326–337, 2014.

[30] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Information Theory*, 12(3):399–401, 1966.

[31] Google. Our story: from the garage to the googleplex. `https://www.google.co.uk/about/our-story/`, 2016.

[32] Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini. Dynamic compressed strings with random access. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 504–515. Springer, 2013.

[33] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '98, pages 366–398, London, UK, UK, 1998. Springer-Verlag.

[34] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[35] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science*, pages 549–554. IEEE Computer Society, 1989.

[36] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. CRAM: compressed random access memory. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *Lecture Notes in Computer Science*, pages 510–521. Springer, 2012.

[37] Subhash Kak. Generalized unary coding. *Circuits, Systems, and Signal Processing*, 35(4):1419–1426, Apr 2016.

[38] Ming Yang Kao. *Encyclopedia of Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[39] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.)*. Addison Wesley Longman, 1998.

[40] Henk Moed. The evolution of big data as a research and scientific topic: Overview of the literature. Special issue, Research Trends, `http://www.researchtrends.com`, 2012.

[41] Christian Worm Mortensen, Rasmus Pagh, and Mihai Patrascu. On dynamic range reporting in one dimension. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 104–111. ACM, 2005.

[42] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[43] Stefan Nilsson and Matti Tikkanen. An experimental study of compression methods for dynamic tries. *Algorithmica*, 33(1):19–33, 2002.

[44] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[45] Andreas Poyias. Xxml: Handling extra-large xml documents. `http://www.cs.le.ac.uk/SiXML/`.

[46] Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. m-bonsai: a practical compact dynamic trie. *CoRR*, abs/1704.05682, 2017.

[47] Andreas Poyias and Rajeev Raman. Improved practical compact dynamic tries. In *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2015.

[48] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[49] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2003.

[50] John Michael Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(2):416–423, 1971.

[51] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Memory-efficient frequent-itemset mining. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 461–472, 2011.

[52] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[53] Pang-Ning Tan et al. Introduction to data mining, 2006.

[54] Naoki Yoshinaga. cedar - c++ implementation of efficiently-updatable double-array trie. http://www.tkl.iis.u-tokyo.ac.jp/ ynaga/cedar/.

[55] Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In Jan Hajic and Junichi Tsujii, editors, *COLING 2014, 25th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, August 23-29, 2014, Dublin, Ireland*, pages 1091–1102. ACL, 2014.

[56] Chengqi Zhang and Shichao Zhang. *Association Rule Mining: Models and Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2002.