Implicit Data Structures, Sorting, and Text Indexing

Jesper Sindahl Nielsen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Implicit Data Structures, Sorting, and Text Indexing

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Jesper Sindahl Nielsen
July 31, 2015

Abstract

This thesis on data structures is in three parts. The first part deals with two fundamental space efficient data structures: finger search trees and priority queues. The data structures are *implicit*, i.e. they only consist of n input elements stored in an array of length n. We consider the problem in the *strict* implicit model which allows no information to be stored except the array and its length. Furthermore the n elements are comparable and indivisible, i.e. we cannot inspect their bits, but we can compare any pair of elements. A finger search tree is a data structure that allows for efficient lookups of the elements stored. We present a strict implicit dynamic finger search strucutre with operations Search, Change-Finger, Insert, and Delete, with times $\mathcal{O}(\log t)$, $\mathcal{O}(n^\varepsilon)$, $\mathcal{O}(\log n)$, $\mathcal{O}(\log n)$, respectively, where t is the rank distance between the current *finger* and the query element. We also prove this structure is optimal in the *strict* implicit model. Next we present two strictly implicit priority queues supporting Insert and ExtractMin in times $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$. The first priority queue has amortized bounds, and the second structure's bounds are worst case, however the first structure also has $\mathcal{O}(1)$ moves amortized for the operations.

The second part of the thesis deals with another fundamental problem: sorting integers. In this problem the model is a word-RAM with word size $w = \Omega(\log^2 n \log\log n)$ bits, and our input is n integers of w bits each. We give a randomized algorithm that sorts such integers in expected $\mathcal{O}(n)$ time. In arriving at our result we also present a randomized algorithm for sorting smaller integers that are packed in words. Letting b be the number of integers per word, we give a packed sorting algorithm running in time $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$.

The topic of the third part is text indexing. The problems considered are term proximity in succinct space, two pattern document retrieval problems and wild card indexing. In all of the problems we are given a collection of documents with total length n. For term proximity we must store the documents using the information theoretic lower bound space (succinct). The query is then a pattern and a value k, and the answer is the top-k documents matching the pattern. The top-k is determined by the Term Proximity scoring function, where the score of a document is the distance between the closest pair of occurrences of the query pattern in the document (lower is better). For this problem we show it is possible to answer queries in $\mathcal{O}(|P| + k\,\mathrm{polylog}(n))$ time, where $|P|$ is the pattern length, and n the total length of the documents. In the two pattern problem queries are two patterns, and we must return all documents matching both patterns (Two-Pattern – 2P), or matching one pattern but not the other (Forbidden Pattern – FP). For these problems we give a solution with space $\mathcal{O}(n)$ words and query time $\mathcal{O}(\sqrt{nk}\log^{1/2+\varepsilon} n)$. We also reduce boolean matrix multiplication to both 2P and FP, giving evidence that high query times are likely necessary. Furthermore we give concrete lower bounds for 2P and FP in the pointer machine model that prove near optimality of all known data structures. In the Wild Card Indexing (WCI) problem queries are patterns with up to κ wild cards, where a wild card matches any character. We give pointer machine lower bounds for WCI, proving near optimality of known solutions.

Resumé

Denne afhandling omhandler datastrukturer og består af tre dele. Den første del er om to grundlæggende pladseffektive datastrukturer: fingersøgningstræer og prioritetskøer. Datastrukturerne er implicit, dvs. består af de n input elementer gemt i en tabel af længde n. Vi studerer problemerne i den *stærke* implicitte model, hvor det kun er tilladt at gemme tabellen og tallet n. Ydermere er det kun muligt at sammenligne alle par af de n elementer, og ellers er elementerne udelelige, dvs. vi kan ikke tilgå deres bits. Et fingersøgningstræ er en datastruktur, hvor man effektivt kan søge efter de opbevarede elementer. Vi beskriver en stærkt implicit dynamisk fingersøgningsstruktur med operationerne SEARCH, CHANGEFINGER, INSERT og DELETE, som tager henholdsvis $\mathcal{O}(\log t)$, $\mathcal{O}(n^\varepsilon)$, $\mathcal{O}(\log n)$ og $\mathcal{O}(\log n)$ tid. Her er t rangafstanden mellem et specielt element, *fingeren*, og det efterspurgte element blandt de n elementer. Vi beviser også, at disse tider er optimale for strengt implicitte fingersøgningsstrukturer. Bagefter præsenterer vi to stærkt implicitte prioritetskøer, der understøtter INSERT og EXTRACTMIN i $\mathcal{O}(1)$ og $\mathcal{O}(\log n)$ tid. Den første prioritetskø har amortiserede grænser, og den anden har værste-falds-grænser (worst case), tilgengæld har den første kun $\mathcal{O}(1)$ flytninger amortiseret per operation.

Den anden del fokuserer på en anden grundlæggende problemstilling: sortering af heltal. Vi studerer problemet i Random Access Machine modellen hvor antallet af bits per ord (ordstørelsen) er $w = \Omega(\log^2 n \log\log n)$, og inputtet er n heltal hver med w bits. Vi giver en randomiseret algoritme, der sorterer heltallene i forventet linær tid. Undervejs udvikler vi en randomiseret algoritme til at sortere mindre heltal, der er pakket ind i ord. Lad b være antallet af heltal pakket i hvert ord, så tager algoritmen for at sortere pakkede heltal $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$ tid.

Emnet i tredje del er tekstindeksering. Problemstillingerne der betragtes er Udtrykstæthed med koncis plads, To-Mønstret dokumenthentning og dokumenthentning med jokere i forespørgslen. I alle problemstillingerne har vi en samling af tekstdokumenter med totallængde n. For udtrykstæthed må datastrukturen kun bruge den informationsteoretiske nedre grænse i plads (koncis/succinct). Forespørgslen er en tekststreng P og en værdi k, og svaret er de top-k dokumenter, der har P som delstreng, med de bedste vurderingstal. De top-k dokumenter afgøres udfra tæthedskriteriet: et dokuments vurderingstal er afstanden mellem de to tætteste forekomster af P (kortere afstand er bedre). Vi viser, at det er muligt at besvare den slags forespørgsler i $\mathcal{O}(|P| + k \operatorname{polylog} n)$ tid, hvor $|P|$ er længden P. I To Mønstre problemerne er forespørgsler to tekststrenge P_1 og P_2. I den ene variant skal vi returnere alle dokumenter, der indeholder både P_1 og P_2, i den anden variant skal vi returnere alle dokumenter, der indeholder P_1 men ikke P_2. For disse problemer giver vi en datastruktur med $\mathcal{O}(n)$ plads og $\mathcal{O}(\sqrt{nk}\log^{1/2+\varepsilon} n)$ forespørgselstid. Vi reducerer desuden boolsk matrix multiplikation til begge problemer, hvilket er belæg for at forespørgslerne må tage lang tid. Ydermere giver vi pointermaskine nedre grænser for To Mønstre problemerne, der viser at alle kendte datastrukturer er næsten optimale. I joker problemet er forespørgsler tekststrenge med jokere, og resultatet er alle dokumenter, hvor forespørgslen forekommer, når man lader jokere være et hvilket som helst bogstav. For dette problem viser vi også nedre grænser i pointermaskine modellen, der påviser, at de kendte datastrukturer er næsten optimale.

Preface

I have always been fond of programming, even from an early age where I played around with web pages, which later turned into server side scripting, and landed me a job at a local web development company. In the first year of my undergraduate I was introduced to algorithms and data structures, which soon became my primary interest. I met Mark Greve who arranged programming competitions and practice sessions, which were mostly about solving problems reminiscent of the ones we encountered at the algorithms and data structure exams. I enjoyed these programming contests, and as my abilities grew I became more and more interested in the theory of algorithms and data structures. When I started my graduate studies, one of the first courses I took was Computational Geometry with Gerth Brodal teaching it. He came to learn of my interest in algorithmics and soon suggested I pursue a PhD degree with him as my advisor, and this thesis is the end result.

My PhD position has been partially at Aarhus University and partially at the State and University Library. The time at university has been mostly spent focusing on the theory, where as at the library we have focused on the practical side. At the library we have developed several tools for quality assurance in their digital audio archives, which are frequently used by their digital archiving teams.

During my PhD studies I have been a co-author on 6 published papers, and 2 papers under submission. In this thesis I have only included 6 of these papers (one unpublished), but the remaining two still deserve to be mentioned. The first I will mention is part of the work I did at the State and University Library with quality assurance of their digital sound archives. The paper is about a tool used for checking if a migration from one audio file format to another succeeded. Here succeeded means that the content of the migrated file sounds the same as the original. This is particularly useful for libraries since there are standards on how to store digital sound archives and typically they receive audio formats different from their standard.

> **1** Bolette Ammitzbøll Jurik and Jesper Sindahl Nielsen. Audio quality assurance: An application of cross correlation. In *International Conference on Preservation of Digital Objects (iPRES)*, pages 196–201, 2012

The second paper is on covering points with curves and lines. We give an

algorithm for the Line Cover problem, and more generally Curve Cover [6]. In the Curve Cover problem we are given a set of n points in $\mathbb{R}^d$ and the task is to find the minimum number of curves needed to cover all the points. The curves are from a specific class (e.g. lines, circles, ellipses, parabolas) where any pair of curves have at most s intersections and d degrees of freedom. We give an algorithm that runs in time $O^\star\left(\left(\frac{k}{\log k}\right)^k\right)$ where k is the minimum number of curves needed and the $O^\star$ hides polynomial factors in n and k.

2 Peyman Afshani, Edvin Berglin, Ingo van Duijn, and Jesper Sindahl Nielsen. Applications of incidence bounds in point covering problems. 2015. (In submission)

Each chapter of the thesis is based on papers already published or in submission. To establish the connection between chapters and the papers we now list the papers each chapter is based on.

Chapter 2

3 Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Finger search in the implicit model. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 527–536, 2012

4 Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Strictly implicit priority queues: On the number of moves and worst-case time. In *Algorithms and Data Structures Workshop (WADS)*, 2015

Chapter 3

5 Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected linear time sorting for word size $\Omega(\log^2 n \log\log n)$. In *Scandinavian Workshop on Algorithm Theory (SWAT), Proceedings*, pages 26–37, 2014

Chapter 4

6 Kasper Green Larsen, J. Ian Munro, Jesper Sindahl Nielsen, and Sharma V. Thankachan. On hardness of several string indexing problems. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 242–251, 2014

7 J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014

8 Peyman Afshani and Jesper Sindahl Nielsen. Data structure lower bounds for set intersection and document indexing problems. 2015 (manuscript, in submission)

Acknowledgments

First I would like to thank my advisor Gerth Stølting Brodal for accepting me as PhD student. Gerth's door is always open and we have had many interesting talks on research, education, and life in general. Secondly, I would like to thank Lars Arge for providing the means required to maintain an excellent environment of great researchers. Even though research is the primary force, the environment here is also friendly, good-humoured, and delightful. I would like to thank Jakob Truelsen for working with me on the first publication, reimplementing the (in?)famous programming language RASMUS, and hosting programming competitions. In a similar vein, my thanks go to Mark Greve who introduced me to programming competitions that gave the original spark for algorithmic curiousity. I also thank Mathias Rav for taking over the job of preparing and hosting the programming competitions at Aarhus University, now that my time here is coming to an end. During the last part of my PhD I have been working a lot with Peyman Afshani and I have learned much during this time, thank you. Peyman also arranged for a group of the MADALGO PhD students to visit Otfriend Cheong at KAIST in Daejon, South Korea, for a week. This was a truly pleasent trip and I highly appreciate being part of Otfried's and Peyman's venture.

The great atmosphere at MADALGO is in large part thanks to the PhD students here and I am grateful to have been a part this group. We have had a wide variety of social activities such as LAN parties, board game nights, movie nights, Friday bars (a frequent occurrence), and concerts. I would like to in particular thank Ingo van Duijn and Edvin Berglin for their great sense of humour that can turn any gathering into a party. I would also like to thank my previous office mate Casper Kejlberg-Rasmussen for many interesting conversations and for the fun LAN parties.

From the State and University Library I would foremost like to thank Bjarne Andersen for finding excellent and interesting topics to work on. I would also like to thank the team I primarily worked with: Bolette Ammitzbøll Jurik, Asger Askov-Blekinge, and Per Møldrup-Dalum. We have been to conferences, meetings, and social events that I have all enjoyed.

I spent half a year at the University of Waterloo, Canada, hosted by Ian Munro and Alex López-Ortiz. When I arrived at Waterloo several people helped me get settled in and quickly became my friends, thanks to Daniela Maftuleac, Martin Derka, Shahin Kamali and Edward Lee. We had fun times in the Grad House, where we often shared a pitcher of beer (or more) on Fridays, so thank you Ian and Alex for inviting me to Waterloo. During my stay in Waterloo I met Sharma Thankachan, a postdoc who helped me delve deeper into the world of strings and information retrieval. Sharma also invited me to Louisianna, where we visited his previous research group, and of course we spent some time in both Baton Rouge and New Orleans where we had a lot of fun!

Finally I would like to thank my friends and family. First my parents, Inge and Jan, for encouraging me to follow my interests and believing in me. I would also like to thank my brother, Jørn, for nerdy discussions on almost any topic and in general great company.

Jesper Sindahl Nielsen,
Aarhus, July 31, 2015.

Contents

Chapter 1

Introduction

Computer Science is a very data-centric science, and in this sense the danish term "datalogi" (the study (or science) of data), is much more accurate. The availability of data has increased drastically in recent years, everytime we interact with any kind of electronic system (which is almost all the time) more data is generated. But what exactly is data? As computer scientists we often abstract away exactly where the data comes from, and deal with how to process it instead. For instance we might want to compute a function on some data, such as finding the median of a set of integers, or sorting a set of reals. In such a setting we do not care whether those values are temperature measurements, financial statistics, or birth years. For the fundamental problems (e.g. finding the median and sorting), we want general solutions that works for any type of data in order to not reinvent the wheel time and time again, and clearly there is little differences in sorting a set temperatures and sorting a set of birth years.

When we have data, we need to store the data such that we can efficiently manipulate it. However, we also want to store it efficiently, meaning we wish to not waste space. In the area of data structures we develop schemes to store data along with procedures for accessing it. We call the procedures *algorithms* and the storage scheme a *data structure*. There are various types of data structures and algorithms depending on what we want to compute. We generally have two categories of data: *static* data and *dynamic* data.

Static data is never changed once it has been collected or given. For static data structures (i.e. data structures never receive updates to the data) we typically have two associated algorithms: a *query* algorithm and a *building* algorithm. The job of the query algorithm is to answer questions about the data. The building (or preprocessing) algorithm puts the data in the proper order so that the query algorithm is correct and efficient, i.e. it builds the data structure. Examples of static data are old newspapers and dictionaries. For old newspapers we might be interested in retrieving all articles written by a particular author in a certain period. Alternatively we might be interested in

all newspaper articles with a headline containing one word but not containing another word (we will return to this problem in Chapter 4).

Dynamic data changes over time which means our algorithms should support such operations as well. We usually have two types of updates to our data, *insertions* for new arriving data, and *deletions* for data we no longer wish to maintain. One example of dynamic data is road networks, where we insert new roads when they are built and delete roads that have been closed down. Another example is to maintain a list of vendors selling the same item, and we want to always know who has the cheapest. Over time the vendors may change their price, which would correspond to an update. For such a task we would use a *priority queue* (we will return to priority queues in Chapter 2).

Many of the basic algorithms and data structures have been known for years and are now part of the standard libraries of most modern programming languages. Likewise, many of the search structures developed are implemented in databases to allow fast lookups. Exactly because of these reasons, any improvement on the fundamental data structures can have a huge impact, since it could potentially make *nearly all* software faster.

We measure how good a static data structure is in terms of space usage versus query time. The trivial data structure is to simply store the input elements as is, and when answering queries we inspect every element. Letting n be the number of elements in our data structure, this requires (at least for the problems considered in this thesis) linear, or $\mathcal{O}(n)$, query time and linear space. For some problems this is actually optimal in some sense, but for many other problems it is not. An example is to find all newspaper articles containing a particular phrase. We can find the articles in time proportional to the length of the phrase and to the number of articles containing the phrase. This is optimal, because we must read the query and we must list the output. For a number of problems there are many solutions that all hit different trade-offs between space usage and query times. One problem where it is possible to achieve any trade-off is the *Forbidden Pattern* problem, where we must find all newspaper articles containing one given phrase but not containing another given phrase. In the Forbidden Pattern problem it is actually possible to achieve any query time $Q(n)$, and space usage $S(n)$ approximately $O\left(\frac{n^2}{Q(n)}\right)$ (ignoring subpolynomial factors). This means as we allow for slower queries we can use less space, or vice versa. The interesting part here is, that we can actually prove, that we cannot do better! More on this in Chapter 4.

So far we have briefly discussed what data structures are. We typically call them *upper bounds*, since a data structure with certain space and time bounds proves that a problem can be solved using at most that amount of time and space. When we want to prove that a data structure is optimal, we prove that any data structure must *at least* use a certain amount of space or time to solve the problem correctly. A statement about the least amount of resources

required to solve a problem is called a *lower bound*. Intuitively it should be difficult to prove *lower bounds* for data structures, since it requires us to reason about all possible data structures, also the ones we have not yet discovered, and make a general statement about them. To prove lower bounds we need to set the rules and define what an algorithm or data structure is allowed to do. These definitions are captured in a *model*. In this thesis we study data structures in various models: the *implicit model*, the *RAM* (Random Access Machine) model and the *pointer machine* model. Each model has its own set of rules, that allows us to reason about their efficiency and correctness. In each chapter there will be a description of the particular models used.

A prevalent theme of this thesis is space efficiency of data structures. There are many reasons to care about space efficiency of data structures and utilizing every bit in the best way possible, we now give a brief motivation. As data becomes more and more available and we collect more and more of it, we need to put more focus on space efficiency, since we might simply run out of memory or space due to the large quantities of data. When we create data structures based on a model, the model is an abstraction of reality that allows us to prove properties and give guarantees about the running times of the algorithms. However, if we wanted to model a computer completely it would be nearly impossible to prove anything interesting due to its many layers and complexities. This is why so many different models exist, they each focus on a particular cost for the algorithms. The word-RAM model resembles the computer to a high extent, in the sense that we have words consiting of a number of bits (typically more than $c \log n$ so we can encode any number between 0 and n^c), and we can read one word at a time, and perform any instruction a regular CPU can execute on a register. This model is quite reasonable, and anything we can compute in the word-RAM model we can also implement and run on our computers. However, the cost of an algorithm in the RAM model is the number of operations we execute and the number of words we read (each costs 1 unit of time). That is only one of the relevant measures when analyzing running times of algorithms. Another very important measure is the number of cache misses we perform during the execution of an algorithm. If an algorithm analyzed in the RAM model has running time $\mathcal{O}(n)$, that might be very good. However it may suffer in other ways because it might rely too heavily on random accesses, i.e. every time a cell is read it may incur a cache miss. Then we suddenly have $\mathcal{O}(n)$ cache misses, where if we could just scan the input from one end to the other, we would only incur a cache miss every B reads since a cache miss makes the computer fetch the B following records, giving $\mathcal{O}(n/B)$ cache misses. The models that deal with the transfer of records from various levels of cache or external memory to internal memory are the *Cache-Oblivous* model and the *External Memory* model (or *I/O* model).

We will not spend much time on these two models, but here they serve a purpose: to illustrate the usefulness of space efficient data structures. In-

ternal memory space efficient data structures and algorithms synergize well with external memory algorithms due to the following reasons. Sometimes the problem to be solved in external memory has a core problem, which might be significantly smaller, but still on the verge of what we can handle internally. If we develop space efficient solutions and minimize the overhead for the internal algorithm, this may very well be enough to actually have an efficient solution in practice. To illustrate this with an example, we consider one assumption that is very useful in practice, which concerns the well-known sweep line paradigm from computational geometry. The general framework is that for a set of points or line segments we need to compute some function, e.g. the intersections of all pairs of line segments. The algorithm works (intuitively) by sweeping a line from top to bottom, inserting line segments into a search structure when they are encountered and removing a line segment when the sweep line reaches its end point. As the line sweeps down, the search structure is used to find neighbouring line segments, checking if they intersect and reorder them when their intersection point has been passed. Each end point of a line segment or point of intersection between two line segments is called an *event*, and we need to process these events in order. One method of processing them in the right order is to insert all end points of line segments into a priority queue, and insert intersection points as we discover them. To move on to the next event we simply extract the maximum value from the priority queue (if we are sweeping top to bottom). It works well in practice to assume that the number of line segments that intersect the sweep line fit in internal memory. It turns out that for inputs that are encountered in practice this assumption is true, though clearly we can design input instances where it is not true: put all line segments such that they all cross the same horizontal line. If the search structure or event handling structure that we use to maintain these lines and event points have too much overhead, it might not fit in internal memory and then this algorithm becomes very slow due to needing I/O operations. Even if we then use an external memory structure for the line segements intersecting the sweep line, it still takes too long compared to having a fast internal structure, simply because I/O operations are so much more expensive. If we want to optimize our algorithm in practice we should take into account all the facets, both internal and external computation. As the example illustrates, sometimes what can make or break an algorithm's efficiency comes down to the space usage of its data structures.

In the rest of this thesis we have three chapters, each covering a slightly different area. We start in Chapter 2 with space efficient fundamental data structures developed in the implicit model. The data structures are search trees with the finger search property, Sections 2.3 and 2.5 and priority queues in Sections 2.7 and 2.8. Since the structures are implicit they are optimal in regards of space. Their query times are also theoretically optimal. We note it is difficult to get a good working implementation of the dynamic finger search tree presented in Section 2.2, since it relies on a very complicated implicit

dictionary as a black box. The static version however is straight forward to implement, though it is mostly a theoretical study of the limits of the implicit model and better implementations with low query overhead are available. In Chapter 3 we present a linear time algorithm for sorting integers when we have a RAM with a large word size. The algorithm pushes the boundaries for which word sizes we can sort in linear time, and it is an improvement on a more than 15 year old result. However, the question of whether we can sort in linear time for *all* word sizes remains open. Concretely the question is, can we sort integers in linear time on a RAM with $w = \omega(\log n)$ and $w = o(\log^2 n \log\log n)$?

Text indexing is the subject of the last chapter where we study a variety of text indexing problems and prove both upper and lower bounds. We start in Sections 4.1-4.4 by giving an introduction to the area of document retrieval, and highlight a few interesting problems and their history as well as some preliminaries. In Section 4.5 we give an upper bound for the *Term Proximity* document retrieval problem. Term proximity is a scoring function where a document's score is the distance between the closest pair of occurrences of a pattern. The task is then to report the top-k documents for a given pattern and k. We then move on to give an upper bound for the *Forbidden Pattern* problem in Section 4.6, which is achieved through a reduction to what is known as the *Common Colors* problem. Before we move on to prove lower bounds, we give evidence that the Forbidden Pattern (and that family of problems) is hard by giving a reduction from the Boolean Matrix Multiplication. This is also known as a "conditional lower bounds", more on that in Section 4.7. Finally we move on to prove lower bounds in the pointer machine model for a number of document indexing problems. Firstly, in Section 4.8 we present two different lower bounds for the *Wild Card Indexing* problem, where we are given a collection of documents, and queries are texts with "wild card" symbols that match any character. Afterwards we turn our attention to the Two Pattern problems, such as the Forbidden Pattern problem, and prove almost tight trade-offs between space usage and query times. The Two Pattern lower bounds are considered both for the reporting version and the counting version in Sections 4.9-4.10. Finally in Section 4.11 we discuss the implications of our lower bounds for other core problems in theoretical computer science, in particular the theory of algorithms and data structures.

Chapter 2

Fundamental Implicit Data Structures

In this chapter we deal with fundamental data structures developed in the *implicit model* which focuses on space efficiency. The implicit model is also sometimes called *in-place*, typically used when talking about algorithms rather than data structures.

In this chapter two fundamental data structuring problems are studied in the implicit model. The first one is about search trees with the finger search property. The second problem is priority queues with few element moves per operation. For these two data structuring problems the elements are drawn from a totally ordered universe. We first give a description of the implicit model in Section 2.1. In Sections 2.2-2.5 we present our solutions for the finger search trees and we present our implicit priority queue results in Sections 2.6-2.8.

2.1 The Implicit Model

The implicit model is defined as follows. During operations we are only allowed to use $\mathcal{O}(1)$ registers. Each register stores either an element or $\Theta(\log n)$ bits (the word size of the computer). The only allowed operations on elements are comparisons and swaps (i.e. two elements switch their position). Any operations usually found in a RAM are allowed on registers and n in $\mathcal{O}(1)$ time. The cost of an algorithm (also known as its running time) is the sum of the number of comparisons, swaps, and RAM operations performed on registers. Between operations we are only allowed to explicitly store the elements and the number of elements n. There are variations on the limits of exactly what we are allowed to store. Some papers store nothing except n and the elements (see [20, 61]) while others allow $\mathcal{O}(1)$ additional words between operations [58,

This chapter includes the two papers on implicit data structures [25, 26]

60, 95]. We denote the version with $\mathcal{O}(1)$ additional words the *weak* implicit model and the version with no additional words the *strict* implicit model. In both models the elements are stored in an array of length n. Performing insertions and deletions then grows or shrinks the array by one, i.e. the array increases to have $n+1$ entries or decreases to have $n-1$ entries. That means in order to delete an element, the element to be deleted when the procedure returns should be located at position n in the array. Similarly for an insertion the new element appears at position $n+1$.

Note that since (almost) no additional information can be stored between operations all information must be encoded in the permutation of the elements. We study the strict implicit model, i.e. no information is retained between operations and only n and the elements are stored in some permutation. It is interesting to study this version of the implicit model, since there is no exact agreement in the litterature as to which model is "the implicit model". Particularly it is interesting to study the limits of the strict implicit model versus the weak implicit model.

It is very common for data structures to store pointers, but note that we cannot store pointers in this model as we are only allowed to have the input elements and n. Since our elements are drawn from a totally ordered universe we can encode a 0/1-bit with two distinct elements from the universe by their order. That is, with an ordered (in terms of their position in memory) pair of distinct elements (x, y), the pair encodes 1 if $x < y$ and 0 otherwise. E.g. pointers can be encoded with $2 \log n$ elements and decoded in $\mathcal{O}(\log n)$ time.

The most widely known implicit data structure is the binary heap of Williams [121]. The binary heap structure consists of an array of length n, storing the elements, and no information is stored between operations, except for the array and the value n. This is therefore an example of a strictly implicit data structure. We return to a more thorough discussion of priority queues and heaps in Section 2.6

2.2 Finger Search

In this section we consider the problem of creating an implicit dictionary [58] that supports finger search. A dictionary is a data structure storing a set of elements with distinct comparable keys such that an element can be located efficiently given its key. It may also support predecessor and successor queries where given a query k it must return the element with the greatest key less than k or the element with smallest key greater than k, respectively. A dynamic dictionary also supports insertion the and deletion of elements. A dictionary has the finger search property if the time for searching is dependent on the rank distance t between a specific element f, called the *finger*, and the query key k. In the static case $\mathcal{O}(\log t)$ search can be achieved by exponential search on a sorted array of elements starting at the finger.

We show that for any strict implicit dictionary supporting finger searches in $q(t) = \Omega(\log t)$ time, the time to move the finger to another element is $\Omega(q^{-1}(\log n))$, where t is the rank distance between the query element and the finger. We present an optimal implicit static structure matching this lower bound. We furthermore present a near optimal strict implicit dynamic structure supporting SEARCH, CHANGE-FINGER, INSERT, and DELETE in times $\mathcal{O}(q(t))$, $\mathcal{O}(q^{-1}(\log n)\log n)$, $\mathcal{O}(\log n)$, and $\mathcal{O}(\log n)$, respectively, for any $q(t) = \Omega(\log t)$. Finally we show that the search operation must take $\Omega(\log n)$ time for the special case where the finger is always changed to the element returned by the last query.

Dynamic finger search data structures have been widely studied outside the implicit model, e.g. some of the famous dynamic structures that support finger searches are splay trees, randomized skip lists and level linked (2-4)-trees. These all support finger search in $\mathcal{O}(\log t)$ time, respectively in the amortized, expected and worst case sense. For an overview of data structures that support finger search see [27]. We consider two variants of finger search structures. The first variant is the *finger search dictionary* where the SEARCH operation also changes the finger to the returned element. The second variant is the *change finger dictionary* where the CHANGE-FINGER operation is separate from the search operation.

Note that the static sorted array solution does not fit into the strict implicit model, since we are not allowed to use additional space to store the index of f between operations. We show that for a static dictionary in the strict model, if we want a search time of $\mathcal{O}(\log t)$, then CHANGE-FINGER must take time $\Omega(n^{\varepsilon})$, while in the weak model a sorted array achieves $\mathcal{O}(1)$ CHANGE-FINGER time.

Much effort has gone into finding a worst case optimal implicit dictionary. Among the first [94] gave a dictionary supporting insert, delete and search in $\mathcal{O}(\log^2 n)$ time. In [60] an implicit B-tree is presented, and finally in [58] a worst case optimal and cache oblivious dictionary is presented. To prove our dynamic upper bounds we use the movable implicit dictionary presented in [23], supporting INSERT, DELETE, PREDECESSOR, SUCCESSOR, MOVE-LEFT and MOVE-RIGHT. The operation MOVE-RIGHT moves the dictionary laid out in cells i through j to $i+1$ through $j+1$ and MOVE-LEFT moves the dictionary the other direction.

The running time of the SEARCH operation is hereafter denoted by $q(t,n)$. Throughout the chapter we require that $q(t,n)$ is non decreasing in both t and n, $q(t,n) \geq \log t$, and that $q(0,n) < \log \frac{n}{2}$. We define $Z_q(n) = \min\{t \in \mathbb{N} \mid q(t,n) \geq \log \frac{n}{2}\}$, i.e. $Z_q(n)$ is the smallest rank distance t, such that $q(t,n) > \log \frac{n}{2}$. Note that $Z_q(n) \leq \frac{n}{2}$ (since by assumption $q(t,n) \geq \log t$), and if q is a function of only t, then Z_q is essentially equivalent to $q^{-1}(\log \frac{n}{2})$. As an example $q(t,n) = \frac{1}{\varepsilon}\log t$, gives $Z_q(n) = \lceil (\frac{n}{2})^{\varepsilon} \rceil$, for $0 < \varepsilon \leq 1$. We require that for a given q, $Z_q(n)$ can be evaluated in constant time, and that $Z_q(n+1) - Z_q(n)$ is bounded by a fixed constant for all n.

We will use set notation on a data structure when appropriate, e.g. $|X|$ will denote the number of elements in the structure X and $e \in X$ will denote that the element e is in the structure X. Given two data structures or sets X and Y, we say that $X \prec Y \Leftrightarrow \forall (x, y) \in X \times Y : x < y$. We use $d(e_1, e_2)$ to denote the absolute rank distance between two elements, that is the difference of the index of e_1 and e_2 in the sorted key order of all elements in the structure. At any time f will denote the current finger element and t the rank distance between f and the current search key.

Our results In Section 2.3 we present a static *change-finger implicit dictionary* supporting PREDECESSOR in time $\mathcal{O}(q(t, n))$, and CHANGE-FINGER in time $\mathcal{O}(Z_q(n) + \log n)$, for any function $q(t, n)$. Note that by choosing $q(t, n) = \frac{1}{\varepsilon} \log t$, we get a SEARCH time of $\mathcal{O}(\log t)$ and a change finger time of $\mathcal{O}(n^\varepsilon)$ for any $0 < \varepsilon \leq 1$.

In Section 2.4 we prove our lower bounds. First we prove (Lemma 1) that for any algorithm A on a strict implicit data structure of size n that runs in time at most τ, whose arguments are keys or elements from the structure, there exists a set $\mathcal{X}_{\mathtt{A},n}$ of at most $\mathcal{O}(2^\tau)$ array entries, such that A touches only array entries from $\mathcal{X}_{\mathtt{A},n}$, no matter the arguments to A or the content of the data structure. We use this to show that for any *change-finger implicit dictionary* with a search time of $q(t, n)$, CHANGE-FINGER will take time $\Omega(Z_q(n) + \log n)$ for some t (Theorem 1). We prove that for any *change-finger implicit dictionary* SEARCH will take time at least $\log t$ (Theorem 2). A similar argument applies for PREDECESSOR and SUCCESSOR. This means that the requirement $q(t, n) \geq \log t$ is necessary. We show that for any *finger-search implicit dictionary* SEARCH must take at least $\log n$ time as a function of both t and n, i.e. it is impossible to create any meaningful finger-search dictionary in the strict implicit model (Theorem 3).

By Theorem 1 and Theorem 2 the static data structure presented in Section 2.3 is optimal w.r.t. SEARCH and CHANGE-FINGER time trade-off, for any function $q(t, n)$ as defined above. In the special case where the restriction $q(0, n) < \log \frac{n}{2}$ does not hold [58] provides the optimal trade-off.

Finally in Section 2.5 we outline a construction for creating a dynamic *change-finger implicit dictionary*, supporting INSERT and DELETE in time $\mathcal{O}(\log n)$, PREDECESSOR and SUCCESSOR in time $\mathcal{O}(q(t, n))$ and CHANGE-FINGER in time $\mathcal{O}(Z_q(n) \log n)$. Note that by setting $q(t, n) = \frac{2}{\varepsilon} \log t$, we get a SEARCH time of $\mathcal{O}(\log t)$ and a CHANGE-FINGER time of $\mathcal{O}(n^{\varepsilon/2} \log n) = \mathcal{O}(n^\varepsilon)$ for any $0 < \varepsilon \leq 1$, which is asymptotically optimal in the strict model. It remains an open problem if one can get better bounds in the dynamic case by using $\mathcal{O}(1)$ additional words.

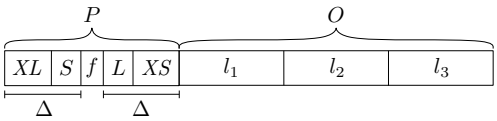


Figure 2.1: Memory layout of the static dictonary.

2.3 Static Finger Search Structure

In this section we present a simple *change-finger implicit dictionary*, achieving an optimal trade-off between the time for SEARCH and CHANGER-FINGER.

Given some function $q(t, n)$, as defined in Section 2.2, we are aiming for a SEARCH time of $\mathcal{O}(q(t, n))$. Let $\Delta = Z_q(n)$. Note that we are allowed to use $\mathcal{O}(\log n)$ time searching for elements with rank-distance $t \geq \Delta$ from the finger, since $q(t, n) = \Omega(\log n)$ for $t \geq \Delta$.

Intuitively, we start with a sorted list of elements. We cut the $2\Delta + 1$ elements closest to f (f being in the center), from this list, and swap them with the first $2\Delta + 1$ elements, such that the finger element is at position $\Delta+1$. The elements that were cut out form the *proximity structure* P, the rest of the elements are in the *overflow structure* O (see Figure 2.1). A SEARCH for x is performed by first doing an exponential search for x in the proximity structure, and if x is not found there, by doing binary searches for it in the remaining sorted sequences.

The proximity structure consists of sorted lists $XS \prec S \prec \{f\} \prec L \prec XL$. The list S contains the up to Δ elements smaller than f that are closest to f w.r.t. rank distance. The list L contains the up to Δ elements closest to f, but larger than f. Both are sorted in ascending order. XL contains a possibly empty sorted sequence of elements larger than elements from L, and XS contains a possibly empty sorted sequence of elements smaller than elements from S. Here $|XL|+|S| = \Delta = |L|+|XS|$, $|S| = \min\{\Delta, \text{rank}(f)-1\}$

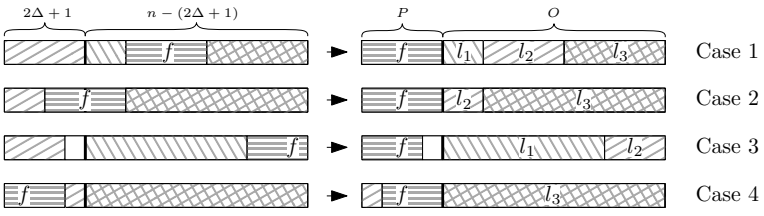


Figure 2.2: Cases for the CHANGE-FINGER operation. The left side is the sorted array. In all cases the horizontally marked segment contains the new finger element and must be moved to the beginning. In the final two cases, there are not enough elements around f so P is padded with what was already there. The emphasized bar in the array is the $2\Delta + 1$ break point between the proximity structure and the overflow structure.

and $|L| = \min\{\Delta, n - \text{rank}(f)\}$. The overflow structure consists of three sorted sequences $l_2 \prec l_1 \prec \{f\} \prec l_3$, each possibly empty.

To perform a CHANGE-FINGER operation, we first revert the array back to one sorted list and the index of f is found by doing a binary search. Once f is found there are 4 cases to consider, as illustrated in Figure 2.2. Note that in each case, at most $2|P|$ elements have to be moved. Furthermore the elements can be moved such that at most $\mathcal{O}(|P|)$ swaps are needed. In particular case 2 and 4 can be solved by a constant number of list reversals.

For reverting to a sorted array and for doing SEARCH, we need to compute the lengths of all sorted sequences. These lengths uniquely determine the case used for construction, and the construction can thus be undone. To find $|S|$ a binary search for the split point between XL and S, is done within the first Δ elements of P. This is possible since $S \prec \{f\} \prec XL$. Similarly $|L|$ and $|XS|$ can be found. The separation between l_2 and l_3, can be found by doing a binary search for f in O, since $l_1 \cup l_2 \prec \{f\} \prec l_3$. Finally if $|l_3| < |O|$, the separation between l_1 and l_2 can be found by a binary search, comparing candidates against the largest element from l_2, since $l_2 \prec l_1$.

When performing the SEARCH operation for some key k, we first determine if $k < f$. If this is the case, an exponential search for k in S is performed. We can detect if we have crossed the boundary to XL, since $S \prec \{f\} \prec XL$. If the element is found it can be returned. If $k > f$ we do an identical search in L. Otherwise the element is neither located in S nor L, and therefore $d(k, f) > \Delta$. All lengths are then reconstructed as above. If $k > f$ a binary search is performed in XL and l_3. Otherwise $k < f$ and binary searches are performed in XS, l_1, and l_2.

Analysis The CHANGE-FINGER operation first computes the lengths of all lists in $\mathcal{O}(\log n)$ time. The case used for constructing the current layout is then identified and reversed in $\mathcal{O}(\Delta)$ time. We locate the new finger f' by binary search in $\mathcal{O}(\log n)$ time and afterwards the $\mathcal{O}(\Delta)$ elements closest to f' are moved to P. We get $\mathcal{O}(\Delta + \log n)$ time for CHANGE-FINGER.

For searches there are two cases to consider. If $t \leq \Delta$, it will be located by the exponential search in P in $\mathcal{O}(\log t) = \mathcal{O}(q(t, n))$ time, since by assumption $q(t, n) \geq \log t$. Otherwise the lengths of the sorted sequences will be recovered in $\mathcal{O}(\log n)$ time, and a constant number of binary searches will be performed in $\mathcal{O}(\log n)$ time total. Since $t \geq \Delta \Rightarrow q(t, n) \geq \log \frac{n}{2}$, we again get a search time of $\mathcal{O}(q(t, n))$.

2.4 Finger Search Lower Bounds

To prove our lower bounds we use an abstracted version of the strict implicit model. The strict model requires that *nothing* but the elements and the number of elements are stored between operations, and that during computation

elements can only be used for comparison. With these assumptions a decision tree can be formed for a given n, where nodes correspond to element comparisons and reads while leaves contain the answers. Note that in the weak model a node could probe a cell containing an integer, giving it a degree of n, which prevents any of our lower bound arguments.

Lemma 1. *Let* `A` *be an operation on an implicit data structure of length n, running in worst case τ time, that takes any number of keys as arguments. Then there exists a set $\mathcal{X}_{\mathtt{A},n}$ of size 2^τ, such that executing* `A` *with any arguments will touch only cells from $\mathcal{X}_{\mathtt{A},n}$ no matter the content of the data structure.*

Proof. Before reading any elements from the data structure, A can reach only a single state which gives rise to a root in a decision tree. When A is in some node s, the next execution step may read some cell in the data structure, and transition into another fixed node, or A may compare two previously read elements or arguments, and given the result of this comparison transition into one of two distinct nodes. It follows that the total number of nodes A can enter within its τ steps is $\sum_{i=0}^{\tau-1} 2^i < 2^\tau$. Now each node can access at most one cell, so it follows that at most 2^τ different cells can be probed by any execution of A within τ steps. □

Observe that no matter how many times an operation that takes at most τ time is performed, the operation will only be able to reach the same set of cells, since the decision tree is the same for all invocations (as long as n does not change).

Theorem 1. *For any* change-finger implicit dictionary *with a search time of $q(t,n)$ as defined in Section 2.2,* CHANGE-FINGER *requires $\Omega(Z_q(n) + \log n)$ time.*

Proof. Let $e_1 \dots e_n$ be a set of elements in sorted order with respect to the keys $k_1 \dots k_n$. Let $t = Z_q(n) - 1$. By definition $q(t+1, n) \geq \log \frac{n}{2} > q(t, n)$. Consider the following sequence of operations:

for $i = 0 \dots \frac{n}{t} - 1$:
  CHANGE-FINGER(k_{it+1})
  for $j = 1 \dots t$: SEARCH(k_{it+j})

Since the rank distance of any query element is at most t from the current finger and q is non-decreasing each search operation takes time at most $q(t, n)$. By Lemma 1 there exists a set $\mathcal{X}$ of size $2^{q(t,n)}$ such that all queries only touch cells in $\mathcal{X}$. We note that $|\mathcal{X}| \leq 2^{q(t,n)} \leq 2^{\log(n/2)} = \frac{n}{2}$.

Since all n elements were returned by the query set, the CHANGE-FINGER operations, must have copied at least $n - |\mathcal{X}| \geq \frac{n}{2}$ elements into $\mathcal{X}$. We performed $\frac{n}{t}$ CHANGE-FINGER operations, thus on average the CHANGE-FINGER operations must have moved at least $\frac{t}{2} = \Omega(Z_q(n))$ elements into $\mathcal{X}$.

For the $\log n$ term in the lower bound, we consider the sequence of operations CHANGE-FINGER(k_i) followed by SEARCH(k_i) for i between 1 and n. Since the rank distance of any search is 0 and $q(0,n) < \log \frac{n}{2}$ (by assumption), we know from Lemma 1 that there exists a set $\mathcal{X}_s$ of size at most $2^{\log(n/2)}$, such that SEARCH only touches cells from $\mathcal{X}_s$. Assume that CHANGE-FINGER runs in time $c(n)$, then from Lemma 1 we get a set $\mathcal{X}_c$ of size at most $2^{c(n)}$ such that CHANGE-FINGER only touches cells from $\mathcal{X}_c$. Since every element is returned, the cell initially containing the element must be touched by either CHANGE-FINGER or SEARCH at some point, thus $|\mathcal{X}_c| + |\mathcal{X}_s| \geq n$. We see that $2^{c(n)} \geq |\mathcal{X}_c| \geq n - |\mathcal{X}_s| \geq n - 2^{\log(n/2)} = 2^{\log(n/2)}$, i.e. $c(n) \geq \log \frac{n}{2}$. □

Theorem 2. *For a* change-finger implicit dictionary *with* SEARCH *time* $q'(t,n)$*, where* q' *is non-decreasing in both* t *and* n*, it holds that* $q'(t,n) \geq \log t$*.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. Let $t \leq n$ be given. First perform CHANGE-FINGER(k_1), then for i between 1 and t perform SEARCH(k_i). From Lemma 1 we know there exists a set $\mathcal{X}$ of size at most $2^{q'(t,n)}$, such that any of the SEARCH operations touch only cells from $\mathcal{X}$ (since any element searched for has rank distance at most t from the finger). The SEARCH operations return t distinct elements so $t \leq |\mathcal{X}| \leq 2^{q'(t,n)}$, and $q'(t,n) \geq \log t$. □

Theorem 3. *For* finger-search implicit dictionary*, the* FINGER-SEARCH *operation requires at least* $g(t,n) \geq \log n$ *time for any rank distance* $t > 0$ *where* $g(t,n)$ *is non decreasing in both* t *and* n*.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. First perform FINGER-SEARCH(k_1), then perform FINGER-SEARCH(k_i) for i between 1 and n. Now for all queries except the first, the rank distance $t \leq 1$ and by Lemma 1 there exists a set of memory cells $\mathcal{X}$ of size $2^{g(1,n)}$ such that all these queries only touch cells in $\mathcal{X}$. Since all elements are returned by the queries we have $|\mathcal{X}| = n$, so $g(1,n) \geq \log n$, since this holds for $t = 1$ it holds for all t. □

We can conclude that it is not possible to achieve any form of meaningful finger-search in the strict implicit model. The static *change-finger implicit dictionary* from Section 2.3 is by Theorem 1 optimal within a constant factor, with respect to the SEARCH to CHANGE-FINGER time trade off, assuming the running time of CHANGE-FINGER depends only on the size of the structure.

2.5 Dynamic Finger Search Structure

For any function $q(t,n)$, as defined in Section 2.2, we present a dynamic *change-finger implicit dictionary* that supports CHANGE-FINGER, SEARCH, INSERT and DELETE in $\mathcal{O}(\Delta \log n), \mathcal{O}(q(t,n)), \mathcal{O}(\log n)$ and $\mathcal{O}(\log n)$ time respec-

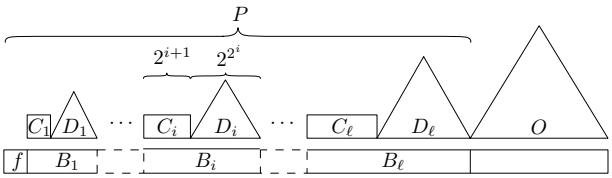


Figure 2.3: Memory layout.

tively, where $\Delta = Z_q(n)$ and n is the number of elements when the operation was started.

The data structure consists of two parts: a *proximity structure* P which contains the elements near f and an *overflow structure* O which contains elements further from f w.r.t. rank distance. We partition P into several smaller structures $B_1, \ldots, B_\ell$. Elements in B_i are closer to f than elements in B_{i+1}. The overflow structure O is an *implicit movable dictionary* [23] that supports MOVE-LEFT and MOVE-RIGHT as described in the Section 2.2. See Figure 2.3 for the layout of the data structure. During a CHANGE-FINGER operation the proximity structure is rebuilt such that $B_1, \ldots, B_\ell$ correspond to the new finger, and the remaining elements are put in O.

The total size of P is $2\Delta+1$. The i'th block B_i consists of a counter C_i and an implicit movable dictionary D_i. The counter C_i contains a pair encoded number c_i, where c_i is the number of elements in D_i smaller than f. The sizes within B_i are $|C_i| = 2^{i+1}$ and $|D_i| = 2^{2^i}$, except in the final block B_ℓ where they might be smaller (B_ℓ might be empty). In particular we define:

$$\ell = \min\left\{\ell' \in \mathbb{N} \,\middle|\, \sum_{i=0}^{\ell'} \left(2^{i+1} + 2^{2^i}\right) > 2\Delta\right\}.$$

We will maintain the following invariants for the structure:

I.1 $\forall i < j, e_1 \in B_i, e_2 \in B_j : d(f, e_1) < d(f, e_2)$

I.2 $\forall e_1 \in B_1 \cup \cdots \cup B_\ell,\ e_2 \in O : d(f, e_1) \leq d(f, e_2)$

I.3 $|P| = 2\Delta + 1$

I.4 $|C_i| \leq 2^{i+1}$

I.5 $|D_i| > 0 \Rightarrow |C_i| = 2^{i+1}$

I.6 $|D_\ell| < 2^{2^\ell}$ and $\forall i < \ell : |D_i| = 2^{2^i}$

I.7 $|D_i| > 0 \Rightarrow c_i = |\{e \in D_i \mid e < f\}|$

We observe that the above invariants imply:

O.1 $\forall i < \ell : |B_i| = 2^{i+1} + 2^{2^i}$ (From I.5 and I.6)

O.2 $|B_\ell| < 2^{\ell+1} + 2^{2^\ell}$ (From I.4 and I.6)

O.3 $d(e, f) \leq 2^{2^k-1} \leq \Delta \Rightarrow e \in B_j$ for some $j \leq k$ (From I.1 – I.6)

2.5.1 Block operations

The following operations operate on a single block and are internal helper functions for the operations described in Section 2.5.2.

BLOCK_DELETE(k, B_i): Removes the element e with key k from the block B_i. This element *must* be located in B_i. First we scan C_i to find e. If it is not found it must be in D_i, so we DELETE it from D_i. If $e < f$ we decrement c_i. In the case where $e \in C_i$ and D_i is nonempty, an arbitrary element g is deleted from D_i and if $g < f$ we decrement c_i. We then overwrite e with g, and fix C_i to encode the new number c_i. In the final case where $e \in C_i$ and D_i is empty, we overwrite e with the last element from C_i.

BLOCK_INSERT(e, B_i): Inserts e into block B_i. If $|C_i| < 2^{i+1}$, e is inserted into C_i and we return. Else we insert e into D_i. If D_i was empty we set $c_i = 0$. In either case if $e < f$ we increment c_i.

BLOCK_SEARCH(k, B_i): Searches for an element e with key k in the block B_i. We scan C_i for e, if it is found we return it. Otherwise if D_i is nonempty we perform a SEARCH on it, to find e and we return it. If the element is not found `nil` is returned.

BLOCK_PREDECESSOR(k, B_i): Finds the predecessor element for the key k in B_i. Do a linear scan through C_i and find the element l_1 with largest key less than k. Afterwards do a predecessor search for key k on D_i, call the result l_2. Return $\max(l_1, l_2)$, or that no element in B_i has key less than k.

2.5.2 Operations

In order to maintain correct sizes of P and O as the entire structure expands or contracts a REBALANCE operation is called at the end of every INSERT and DELETE operation. This is an internal operation that does not require I.3 to be valid before invocation.

REBALANCE(): Balance B_ℓ such that the number of elements in P less than f is as close to the number of elements greater than f as possible. We start by evaluating $\Delta = Z_q(n)$, the new desired proximity size. Let s be the number of elements in B_ℓ less than f which can be computed as $c_\ell + |\{e \in C_\ell \mid e < f\}|$. While $2\Delta + 1 > |P|$ we move elements from O to P. We move the predecessor of f from O to B_ℓ if $O \prec \{f\} \vee (s < \frac{|B_\ell|}{2} \wedge \neg(\{f\} \prec O))$ and otherwise we move the successor of f to O. While $2\Delta + 1 < |P|$ we move elements from B_ℓ to O. We move the largest element from B_ℓ to O if $s < \frac{B_\ell}{2}$. Otherwise we move the smallest element.

CHANGE-FINGER(k): To change the finger of the structure to k, we first insert every element of $B_\ell \ldots B_1$ into O. We then remove the element e with

key k from O, and place it at index 1 as the new f, and finish by performing REBALANCE.

INSERT(e): Assume $e > f$. The case $e < f$ can be handled similarly. Find the first block B_i where e is smaller than the largest element l_i from B_i (which can be found using a predecessor search) or $l_i < f$. Now if $l_i > f$ for all blocks $j \geq i$, BLOCK_DELETE the largest element and BLOCK_INSERT it into B_{j+1}. In the other case where $l_i < f$ for all blocks $j \geq i$, BLOCK_DELETE the smallest element and BLOCK_INSERT it into B_{j+1}. The final element that does not have a block to go into, will be put into O, then we put e into B_i. In the special case where e did not fit in any block, we insert e into O. In all cases we perform REBALANCE.

DELETE(k): We perform a BLOCK_SEARCH on all blocks and a SEARCH in O to find out which structure the element e with key k is located in. If it is in O we just DELETE it from O. Otherwise assume $k < f$ (the case $k > f$ can be handled similarly), and assume that e is in B_i, then BLOCK_DELETE e from B_i. For each $j > i$ we BLOCK_DELETE the predecessor of f in B_j, and insert it into B_{j-1} (in the case where there is no predecessor, we BLOCK_DELETE the successor of f instead). We also delete the predecessor of f from O and insert it in B_ℓ. The special case where $k = f$, is handled similarly to $k < f$, we note that after this the predecessor of f will be the new finger element. In all cases we perform a REBALANCE.

SEARCH(k), PREDECESSOR(k) and SUCCESSOR(k), all follow the same general pattern. For each block B_i starting from B_1, we compute the largest and the smallest element in the block. If k is between these two elements we return the result of BLOCK_SEARCH, BLOCK_PREDECESSOR or BLOCK_SUCCESSOR respectively on B_i, otherwise we continue with the next block. In case k is not within the bounds of any block, we return the result of SEARCH(k), PREDECESSOR(k) or SUCCESSOR(k) respectively on O.

2.5.3 Analysis

By the invariants, we see that every C_i and D_i except the last, have fixed size. Since O is a movable dictionary it can be moved right or left as this final C_i or D_i expands or contracts. Thus the structure can be maintained in a contiguous memory layout.

The correctness of the operations follows from the fact that I.1 and I.2, imply that elements in B_j or O are further away from f than elements from B_i where $i < j$. We now argue that SEARCH runs in time $\mathcal{O}(q(t, n))$. Let e be the element we are searching for. If e is located in some B_i then at least half the elements in B_{i-1} will be between f and e by I.1. We know from O.1 that $t = d(f, e) \geq \frac{|B_{i-1}|}{2} \geq 2^{2^{i-1}-1}$. The time spent searching is $\mathcal{O}(\sum_{j=1}^{i} \log |B_j|) = \mathcal{O}(2^i) = \mathcal{O}(\log t) = \mathcal{O}(q(t, n))$. If on the other hand e is in O, then by I.3 there are $2\Delta + 1$ elements in P, of these at least half are between f and e by I.2, so $t \geq \Delta$, and the time used for searching is $\mathcal{O}(\log n + \sum_{j=1}^{k} \log |B_j|) =$

$\mathcal{O}(\log n) = \mathcal{O}(q(t,n))$. The last equality follows by the definition of Z_q. The same arguments work for PREDECESSOR and SUCCESSOR.

Before the CHANGE-FINGER operation the number of elements in the proximity structure by I.3 is $2\Delta + 1$. During the operation all these elements are inserted into O, and the same number of elements are extracted again by REBALANCE. Each of these operations are just INSERT or DELETE on a movable dictionary or a block taking time $\mathcal{O}(\log n)$. In total we use time $\mathcal{O}(\Delta \log n)$.

Finally to see that both INSERT and DELETE run in $\mathcal{O}(\log n)$ time, notice that in the proximity structure doing a constant number of queries in every block is asymptotically bounded by the time to do the queries in the last block. This is because their sizes increase double-exponentially. Since the size of the last block is bounded by n we can guarantee $\mathcal{O}(\log n)$ time for doing a constant number of queries on every block (this includes predecessor/successor queries). In the worst case, we need to insert an element in the first block of the proximity structure, and "bubble" elements all the way through the proximity structure and finally insert an element in the overflow structure. This will take $\mathcal{O}(\log n)$ time. At this point we might have to rebalance the structure, but this merely requires deleting and inserting a constant number of elements from one structure to the other, since we assumed $Z_q(n)$ and $Z_q(n+1)$ differ by at most a constant. Deletion works in a similar manner.

Conclusion We have now established both static and dynamic search trees with the finger search property exist in the strict implicit model. We also established optimality of the two structures when the desired query time is close to $\mathcal{O}(\log t)$. The dynamic structure is based on a scheme where we have a number of dicionaries with increasing size. The size of the $(i+1)$-th dictionary is the square of the i-th, i.e. $|D_{i+1}| = |D_i|^2$. This is a rapidly progressing series, and for an implementation one would have very few dictionaries (starting with a dictionary of size 2, the 7th dictionary would have up to 2^{64} elements). One way to slightly increase the number of dictionaries, is to let the $(i+1)$-th structure have size $|D_{i+1}| = |D_i|^{1+\varepsilon}$, for some $\varepsilon > 0$. This scheme is potentially better for elements close to the finger, but worse for elements that are further away.

2.6 Priority Queues

In 1964 Williams presented "Algorithm 232" [121], commonly known as the binary heap. The binary heap is a priority queue data structure storing a dynamic set of n elements from a totally ordered universe, supporting the insertion of an element (INSERT) and the deletion of the minimum element (EXTRACTMIN) in worst-case $\mathcal{O}(\log n)$ time.

The binary heap is a complete binary tree structure where each node stores an element and the tree satisfies *heap order*, i.e., the element at a non-root

node is larger than or equal to the element at the parent node. Binary heaps can be generalized to d-ary heaps [78], where the degree of each node is d rather than two. This implies $\mathcal{O}(\log_d n)$ and $\mathcal{O}(d \log_d n)$ time for INSERT and EXTRACTMIN, respectively, using $\mathcal{O}(\log_d n)$ moves for both operations.

Due to the $\Omega(n \log n)$ lower bound on comparison based sorting, either INSERT or EXTRACTMIN must take $\Omega(\log n)$ time, but not necessarily both. Carlson *et al.* [31] presented an implicit priority queue with worst-case $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ time INSERT and EXTRACTMIN operations, respectively. However, the structure is not strictly implicit since it needs to store $\mathcal{O}(1)$ additional words. Harvey and Zatloukal [105] presented a strictly implicit priority structure achieving the same bounds, but amortized. Prior to the work in this section no strictly implicit priority queue with matching worst-case time bounds was known.

There are many applications of priority queues. The most notable examples are and finding minimum spanning trees (MST) and Dijkstra's algorithm for finding the shortest paths from one vertex to all other vertices in a weighted (non-negative) graph (Single Source Shortest Paths). Priority queues are also often used for scheduling or similar greedy algorithms.

Recently Larkin, Sen, & Tarjan, 2014, implemented many of the known priority queues and compared them experimentally [84]. One of their discoveries is that implicit priority queues are almost always better than pointer based structures. Larkin et al., 2014, also found out that for several applications (such as sorting and Dijkstra's algorithm) the fastest times are achieved by using an implicit 4-ary heap.

A measurement often studied in implicit data structures and in-place algorithms is the number of element *moves* performed during the execution of a procedure (separate from the other costs). The number of moves is defined as the number of writes to the array storing elements, i.e. swapping two elements costs 2 moves. Franceschini showed how to sort n elements in-place using $\mathcal{O}(n \log n)$ comparisons and $\mathcal{O}(n)$ moves [57], and Franceschini and Munro [59] presented implicit dictionaries with amortized $\mathcal{O}(\log n)$ time updates with amortized $\mathcal{O}(1)$ moves per update. The latter immediately implies an implicit priority queue with amortized $\mathcal{O}(\log n)$ time INSERT and EXTRACTMIN operations performing amortized $\mathcal{O}(1)$ moves per operation. For a more thorough survey of previous priority queue results, see [21].

Our Results We present two strictly implicit priority queues. The first structure (Section 2.7) limits the number of moves to $\mathcal{O}(1)$ per operation with amortized $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ time INSERT and EXTRACTMIN operations, respectively. However the bounds are all amortized and it remains an open problem to achieve these bounds in the worst case for strictly implicit priority queues. We note that this structure implies a different way of sorting in-place with $\mathcal{O}(n \log n)$ comparisons and $\mathcal{O}(n)$ moves. The second structure

Table 2.1: Selected previous and new results for implicit priority queues. The bounds are asymptotic, and $\star$ are amortized bounds.

	Insert	Extract-Min	Moves	Strict	Identical elements
Williams [121]	$\log n$	$\log n$	$\log n$	yes	yes
Carlsson *et al.* [31]	1	$\log n$	$\log n$	no	yes
Edelkamp *et al.* [52]	1	$\log n$	$\log n$	no	yes
Harvey and Zatloukal [105]	$\star\, 1$	$\star \log n$	$\star \log n$	yes	yes
Franceschini and Munro [59]	$\star \log n$	$\star \log n$	$\star\, 1$	yes	no
Section 2.7	$\star\, 1$	$\star \log n$	$\star\, 1$	yes	yes
Section 2.8	1	$\log n$	$\log n$	yes	no

(Section 2.8) improves over [31, 105] by achieving Insert and ExtractMin operations with worst-case $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ time (and moves), respectively. The structure in Section 2.8 assumes all elements to be distinct where as the structure in Section 2.7 can also be extended to support identical elements (see Section 2.7.4). See Table 2.1 for an overview of new and previous results.

2.7 A Priority Queue with Amortized $\mathcal{O}(1)$ Moves

In this section we describe a strictly implicit priority queue supporting amortized $\mathcal{O}(1)$ time Insert and amortized $\mathcal{O}(\log n)$ time ExtractMin. Both operations perform amortized $\mathcal{O}(1)$ moves. In Sections 2.7.1-2.7.3 we assume elements are distinct. In Section 2.7.4 we describe how to handle identical elements.

Overview The basic idea of our priority queue is the following (the details are presented in Section 2.7.1). The structure consists of four components: an insertion buffer B of size $\mathcal{O}(\log^3 n)$; m insertion heaps $I_1, I_2, \ldots, I_m$ each of size $\Theta(\log^3 n)$, where $m = \mathcal{O}(n/\log^3 n)$; a singles structure T, of size $\mathcal{O}(n)$; and a binary heap Q, storing $\{1, 2, \ldots, m\}$ (integers encoded by pairs of elements) with the ordering $i \leq j$ if and only if $\min I_i \leq \min I_j$. Each I_i and B is a $\log n$-ary heap of size $\mathcal{O}(\log^3 n)$. The table below summarizes the performance of each component:

	Insert		ExtractMin	
Structure	Time	Moves	Time	Moves
B, I_i	1	1	$\log n$	1
Q	$\log^2 n$	$\log^2 n$	$\log^2 n$	$\log^2 n$
T	$\log n$	1	$\log n$	1

It should be noted that the implicit dictionary of Franceschini and Munro [59] could be used for T, but we will give a more direct solution since we only need the restricted ExtractMin operation for deletions.

The Insert operation inserts new elements into B. If the size of B becomes $\Theta(\log^3 n)$, then m is incremented by one, B becomes I_m, m is inserted into Q, and B becomes a new empty $\log n$-ary heap. An ExtractMin operation first identifies the minimum element in B, Q and T. If the overall minimum element e is in B or T, e is removed from B or T. If the minimum element e resided in I_i, where i is stored at the root of Q, then e and $\log^2 n$ further smallest elements are extracted from I_i (if I_i is not empty) and all except e inserted into T (T has cheap operations whereas Q does not, thus the expensive operation on Q is amortized over inexpensive ones in T), and i is deleted from and reinserted into Q with respect to the new minimum element in I_i. Finally e is returned.

For the analysis we see that Insert takes $\mathcal{O}(1)$ time and moves, except when converting B to a new I_m and inserting m into Q. The $\mathcal{O}(\log^2 n)$ time and moves for this conversion is amortized over the insertions into B, which becomes amortized $\mathcal{O}(1)$, since $|B| = \Omega(\log^2 n)$. For ExtractMin we observe that an expensive deletion from Q only happens once for every $\log^2 n$-th element from I_i (the remaining ones from I_i are moved to T and deleted from T), and finally if there have been d ExtractMin operations, then at most $d + m\log^2 n$ elements have been inserted into T, with a total cost of $\mathcal{O}((d + m\log^2 n)\log n) = \mathcal{O}(n + d\log n)$, since $m = \mathcal{O}(n/\log^3 n)$.

2.7.1 The implicit structure

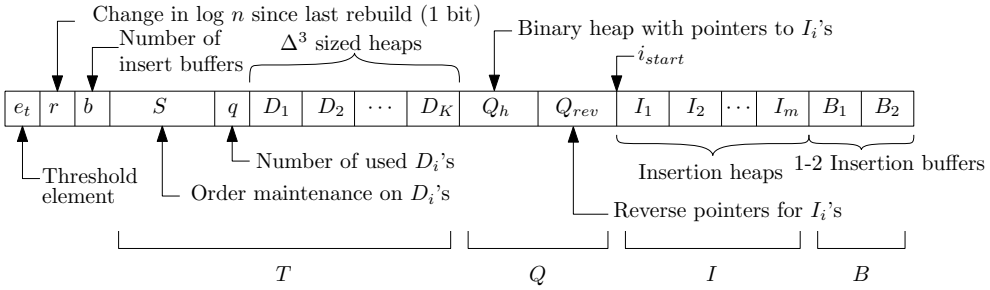


Figure 2.4: The different structures and their layout in memory.

We now give the details of our representation (see Figure 2.4). We select one element e_t as our *threshold element*, and denote elements greater than e_t as *dummy elements*. The current number of elements in the priority queue is denoted n. We fix an integer N that is an approximation of n, where $N \leq n < 4N$ and $N = 2^j$ for some j. Instead of storing N, we store a bit $r = \lfloor \log n \rfloor - \log N$, encoded by two dummy elements. We can then compute

N as $N = 2^{\lfloor \log n \rfloor - r}$, where $\lfloor \log n \rfloor$ is the position of the most significant bit in the binary representation of n (which we assume is computable in constant time). The value r is easily maintained: When $\lfloor \log n \rfloor$ changes, r changes accordingly. We let $\Delta = \log(4N) = \lfloor \log n \rfloor + 2 - r$, i.e., Δ bits is sufficient to store an integer in the range $0..n$. We let $M = \lceil 4N/\Delta^3 \rceil$.

We maintain the invariant that the size of the insertion buffer B satisfies $1 \leq |B| \leq 2\Delta^3$, and that B is split into two parts B_1 and B_2, each being Δ-ary heaps (B_2 possibly empty), where $|B_1| = \min\{|B|, \Delta^3\}$ and $|B_2| = |B| - |B_1|$. We use two buffers to prevent expensive operation sequences that alternate inserting and deleting the same element. We store a bit b indicating if B_2 is nonempty, i.e., $b = 1$ if and only if $|B_2| \neq 0$. The bit b is encoded using two dummy elements. The structures $I_1, I_2, \ldots, I_m$ are Δ-ary heaps storing Δ^3 elements. The binary heap Q is stored using two arrays Q_h and Q_{rev} each of a fixed size $M \geq m$ and storing integers in the range $1..m$. Each value in both arrays is encoded using 2Δ dummy elements, i.e., Q is stored using $4M\Delta$ dummy elements. The first m entries of Q_h store the binary heap, whereas Q_{rev} acts as reverse pointers, i.e., if $Q_h[j] = i$ then $Q_{rev}[i] = j$. All operations on a regular binary heap take $\mathcal{O}(\log n)$ time, but since each "read"/"write" from/to Q needs to decode/encode an integer the time increases by a factor 2Δ. It follows that Q supports Insert and ExtractMin in $\mathcal{O}(\log^2 n)$ time, and FindMin in $\mathcal{O}(\log n)$ time.

We now describe T and we need the following density maintenance result.

Lemma 2 ([22])**.** *There is a dynamic data structure storing n comparable elements in an array of length $(1+\varepsilon)n$, supporting* Insert *and* ExtractMin *in amortized $\mathcal{O}(\log^2 n)$ time and* FindPredecessor *in worst case $\mathcal{O}(\log n)$ time.* FindPredecessor *does not modify the array.*

Corollary 1. *There is an implicit data structure storing n (key, index) pairs, while supporting* Insert *and* ExtractMin *in amortized $\mathcal{O}(\log^3 n)$ time and moves, and* FindPredecessor *in $\mathcal{O}(\log n)$ time in an array of length $\Delta(2+\varepsilon)n$.*

Proof. We use the structure from Lemma 2 to store pairs of a key and an index, where the index is encoded using 2Δ dummy elements. All additional space is filled with dummy elements. However comparisons are only made on keys and not indexes, which means we retain $\mathcal{O}(\log n)$ time for FindMin. Since the stored elements are now an $\mathcal{O}(\Delta) = \Theta(\log n)$ factor larger, the time for update operations becomes an $\mathcal{O}(\log n)$ factor slower giving amortized $\mathcal{O}(\log^3 n)$ time for Insert and ExtractMin. □

The singles structure T intuitively consists of a sorted list of the elements stored in T partitioned into buckets $D_1, \ldots, D_q$ of size at most Δ^3, where the minimum element e from bucket D_i is stored in a structure S from Corollary 1

as the pair (e, i). Each D_i is stored as a Δ-ary heap of size Δ^3, where empty slots are filled with dummy elements. Recall implicit heaps are complete trees, which means all dummy elements in D_i are stored consecutively after the last non-dummy element. In S we consider pairs (e, i) where $e > e_t$ to be empty spaces.

More specifically, the structure T consists of: q, S, $D_1, D_2, \ldots, D_K$, where $K = \lceil \frac{N}{16\Delta^3} \rceil \geq q$ is the number of D_i's available. The structure S uses $\left\lceil \frac{N}{4\Delta^2} \right\rceil$ elements and q uses 2Δ elements to encode a pointer. Each D_i uses Δ^3 elements.

The D_i's and S relate as follows. The number of D_i's is at most the maximum number of items that can be stored in S. Let $(e, i) \in S$, then $\forall x \in D_i : e < x$, and furthermore for any $(e', i') \in S$ with $e < e'$ we have $\forall x \in D_i : x < e'$. These invariants do not apply to dummy elements. Since D_i is a Δ-ary heap with Δ^3 elements we get $\mathcal{O}(\log_\Delta \Delta^3) = \mathcal{O}(1)$ time for Insert and $\mathcal{O}(\Delta \log_\Delta \Delta^3) = \mathcal{O}(\Delta)$ for ExtractMin on a D_i.

2.7.2 Operations

For both Insert and ExtractMin we need to know N, Δ, and whether there are one or two insert buffers as well as their sizes. First r is decoded and we compute $\Delta = 2 + \text{msb}(n) - r$, where $\text{msb}(n)$ is the position of the most significant bit in the binary representation of n (indexed from zero). From this we compute $N = 2^{\Delta-2}$, $K = \lceil N/(16\Delta^3) \rceil$, and $M = \lceil 4N/\Delta^3 \rceil$. By decoding b we get the number of insert buffers. To find the sizes of B_1 and B_2 we compute the value i_{start} which is the index of the first element in I_1. The size of B_1 is computed as follows. If $(n - i_{start}) \bmod \Delta^3 = 0$ then $|B_1| = \Delta^3$. If B_2 exists then B_1 starts at $n - 2\Delta^3$ and otherwise B_1 starts at $n - \Delta^3$. If B_2 exists and $(n - i_{start}) \bmod \Delta^3 = 0$ then $|B_2| = \Delta^3$, otherwise $|B_2| = (n - i_{start}) \bmod \Delta^3$. Once all of this information is computed the actual operation can start. If $n = N + 1$ and an ExtractMin operation is called, then the ExtractMin procedure is executed and afterwards the structure is rebuilt as described in the paragraph below. Similarly if $n = 4N - 1$ before an Insert operation the new element is appended and the data structure is rebuilt.

Insert If $|B_1| < \Delta^3$ the new element is inserted in B_1 by the standard insertion algorithm for Δ-ary heaps. If $|B_1| = \Delta^3$ and $|B_2| = 0$ and a new element is inserted the two elements in b are swapped to indicate that B_2 now exists. When $|B_1| = |B_2| = \Delta^3$ and a new element is inserted, B_1 becomes I_{m+1}, B_2 becomes B_1, $m + 1$ is inserted in Q (possibly requiring $\mathcal{O}(\log n)$ values in Q_h and Q_{rev} to be updated in $\mathcal{O}(\log^2 n)$ time). Finally the new element becomes B_2.

ExtractMin Searches for the minimum element e are performed in B_1, B_2, S, and Q. If e is in B_1 or B_2 it is deleted, the last element in the array is swapped with the now empty slot and the usual bubbling for heaps is performed. If B_2

disappears as a result, the bit b is updated accordingly. If B_1 disappears as a result, I_m becomes B_1, and m is removed from Q.

If e is in I_i then i is deleted from Q, e is extracted from I_i, and the last element in the array is inserted in I_i. The Δ^2 smallest elements in I_i are extracted and inserted into the *singles structure*: for each element a search in S is performed to find the range it belongs to, i.e. D_j, the structure it is to be inserted in. Then it is inserted in D_j (replacing a dummy element that is put in I_i, found by binary search). If $|D_j| = \Delta^3$ and $q = K$ the priority queue is rebuilt. Otherwise if $|D_j| = \Delta^3$, D_j is split in two by finding the median y of D_j using a linear time selection algorithm [30]. Elements $\geq y$ in D_j are swapped with the first $\Delta^3/2$ elements in D_q then D_j and D_q are made into Δ-ary heaps by repeated insertion. Then y is extracted from D_q and (y, q) is inserted in S. The dummy element pushed out of S by y is inserted in D_q. Finally q is incremented and we reinsert i into Q. Note that it does not matter if any of the elements in I_i are dummy elements, the invariants are still maintained.

If $(e, i) \in S$, the last element of the array is inserted into the singles structure, which pushes out a dummy element z. The minimum element y of D_i is extracted and z inserted instead. We replace e by y in S. If y is a dummy element, we update S as if (y, i) was removed. Finally e is returned. Note this might make B_1 or B_2 disappear as a result and the steps above are executed if needed.

Rebuilding We let the new $N = n'/2$, where n' is n rounded to the nearest power of two. Using a linear time selection algorithm [30], find the element with rank $n - i_{start}$, this element is the new threshold element e_t, and it is put in the first position of the array. Following e_t are all the elements greater than e_t and they are followed by all the elements comparing less than e_t. We make sure to have at least $\Delta^3/2$ elements in B_1 and at most $\Delta^3/2$ elements in B_2 which dictates whether b encodes 0 or 1. The value q is initialized to 1. All the D_i structures are considered empty since they only contain dummy elements. The pointers in Q_h and Q_{rev} are all reset to the value 0. All the I_i structures as well as B_1 (and possibly B_2) are made into Δ-ary heaps with the usual heap construction algorithm. For each I_j structure the Δ^2 smallest elements are inserted in the singles structure as described in the ExtractMin procedure, and j is inserted into Q. The structure now satisfies all the invariants.

2.7.3 Analysis

In this subsection we give the analysis that leads to the following theorem.

Theorem 4. *There is a strictly implicit priority queue supporting* Insert *in amortized $\mathcal{O}(1)$ time,* ExtractMin *in amortized $\mathcal{O}(\log n)$ time. Both operations perform amortized $\mathcal{O}(1)$ moves.*

INSERT While $|B| < 2\Delta^3$, each insertion takes $\mathcal{O}(1)$ time. When an insertion happens and $|B| = 2\Delta^3$, the insertion into Q requires $\mathcal{O}(\log^2 n)$ time and moves. During a sequence of s insertions, this can at most happen $\lceil s/\Delta^3 \rceil$ times, since $|B|$ can only increase for values above Δ^3 by insertions, and each insertion at most causes $|B|$ to increase by one. The total cost for s insertions is $\mathcal{O}(s + s/\Delta^3 \cdot \log^2 n) = \mathcal{O}(s)$, i.e., amortized constant per insertion.

EXTRACTMIN We first analyze the cost of updating the singles structure. Each operation on a D_i takes time $\mathcal{O}(\Delta)$ and performs $\mathcal{O}(1)$ moves. Locating an appropriate bucket using S takes $\mathcal{O}(\log n)$ time and no moves. At least $\Omega(\Delta^3)$ operations must be performed on a bucket to trigger an expensive bucket split or bucket elimination in S. Since updating S takes $\mathcal{O}(\log^3 n)$ time, the amortized cost for updating S is $\mathcal{O}(1)$ moves per insertion and extraction from the singles structure. In total the operations on the singles structure require amortized $\mathcal{O}(\log n)$ times and amortized $\mathcal{O}(1)$ moves. For EXTRACTMIN the searches performed all take $\mathcal{O}(\log n)$ comparisons and no moves. If B_1 disappears as a result of an extraction we know at least $\Omega(\Delta^3)$ extractions have occurred because a rebuild ensures $|B_1| \geq \Delta^3/2$. These extractions pay for extracting I_m from Q_h which takes $\mathcal{O}(\log^2 n)$ time and moves, amortized this gives $\mathcal{O}(1/\log n)$ additional time and moves. If the extracted element was in I_i for some i, then Δ^2 insertions occur in the singles structure each taking $\mathcal{O}(\log n)$ time and $\mathcal{O}(1)$ moves amortized. If that happens either $\Omega(\Delta^3)$ insertions or Δ^2 extractions have occurred: Suppose no elements from I_i have been inserted in the singles structure, then the reason there is a pointer to I_i in Q_h is due to $\Omega(\Delta^3)$ insertions. When inserting elements in the singles structure from I_i the number of elements inserted is Δ^2 and these must first be deleted. From this discussion it is evident that we have saved up $\Omega(\Delta^2)$ moves and $\Omega(\Delta^3)$ time, which pay for the expensive extraction. Finally if the minimum element was in S, then an extraction on a Δ-ary heap is performed which takes $\mathcal{O}(\Delta)$ time and $\mathcal{O}(1)$ moves, since its height is $\mathcal{O}(1)$.

Rebuilding The cost of rebuilding is $\mathcal{O}(n)$, due to a selection and building heaps with $\mathcal{O}(1)$ height. There are three reasons a rebuild might occur: (i) n became $4N$, (ii) n became $N - 1$, or (iii) An insertion into T would cause $q > K$. By the choice of N during a rebuild it is guaranteed that in the first and second case at least $\Omega(N)$ insertions or extractions occurred since the last rebuild, and we have thus saved up at least $\Omega(N)$ time and moves. For the last case we know that each extraction incurs $\mathcal{O}(1)$ insertions in the singles structure in an amortized sense. Since the singles structure accommodates $\Omega(N)$ elements and a rebuild ensures the singles structure has $o(n)$ non dummy elements (Lemma 3), at least $\Omega(N)$ extractions have occurred which pay for the rebuild.

Lemma 3. *Immediately after a rebuild $o(n)$ elements in the singles structure are non-dummy elements*

Proof. There are at most n/Δ^3 of the I_i structures and Δ^2 elements are inserted in the singles structure from each I_i, thus at most $n/\Delta = o(n)$ non-dummy elements reside in the singles structure after a rebuild. □

The paragraphs above establish Theorem 4.

2.7.4 Handling Identical Elements

The primary difficulty in handling identical elements is that we lose the ability to encode bits. The primary goal of this section is to do so anyway. The idea is to let the items stored in the priority queue be pairs of distinct elements where the key of an item is the lesser element in the pair. In the case where it is not possible to make a sufficient number of pairs of distinct elements, almost all elements are equal and this is an easy case to handle. Note that many pairs (or all for that matter) can contain the same elements, but each pair can now encode a bit, which is sufficient for our purposes.

The structure is almost the same as before, however we put a few more things in the picture. As mentioned we need to use *pairs of distinct* elements, so we create a mechanism to produce these. Furthermore we need to do some book keeping such as storing a pointer and being able to compute whether there are enough pairs of distinct elements to actually have a meaningful structure. The changes to the memory layout is illustrated in Figure 2.5.

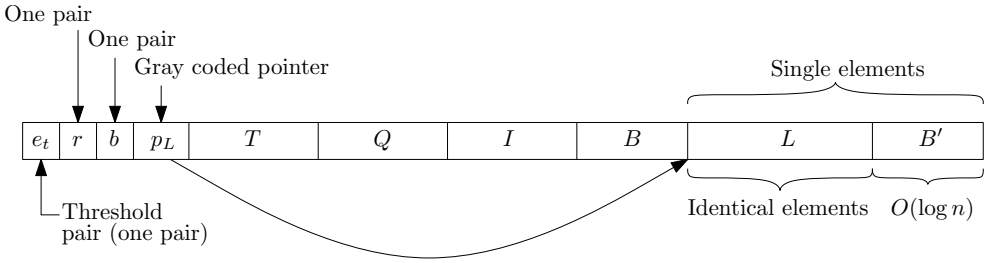


Figure 2.5: The different structures and their layout in memory.

Modifications The areas L and B' in memory are used to produce pairs of distinct elements. The area p_L is a Gray coded pointer[66] with $\Theta(\log n)$ pairs, pointing to the beginning of L. The rest of the structure is essentially the same as before, except instead of storing elements, we now store pairs $e = (e_1, e_2)$ and the key of the pair is $e_k = \min\{e_1, e_2\}$. All comparisons between items are thus made with the key of the pair. We will refer to the priority queue from Section 2.7 as PQ.

Gray, F.: Pulse code communications. U.S. Patent (2632058) (1953)

There are a few minor modifications to PQ. Recall that we needed to simulate *empty spaces* inside T (specifically in S, see Figure 2.4). The way we simulated empty spaces was by having elements that compared greater than e_t. Now e_t is actually a pair, where the minimum element is the threshold element. It might be the case that there are many items comparing equal to e_t, which means some would be used to simulate empty spaces and others would be actual elements in PQ and some would be used to encode pointers. This means we need to be able to differentiate these types that might all compare equal to e_t. First observe that items used for pointers are always located in positions that are distinguishable from items placed in positions used as *actual* items. Thus we do not need to worry about confusing those two. Similarly, the "empty" spaces in T are also located in positions that are distinguishable from pointers. Now we only need to be able to differentiate "empty" spaces and occupied spaces where the keys both compare equal to e_t. Letting items (i.e. pairs) used as empty spaces encode 1, and the "occupied" spaces encode 0, empty spaces and occupied spaces become differentiable as well. Encoding that bit is possible, since they are not used for encoding anything else.

Since many elements could now be identical we need to decide whether there are enough distinct elements to have a meaningful structure. As an invariant we have that if the two elements in the pair $e_t = (e_{t,1}, e_{t,2})$ are equal then there are not enough elements to make $\Omega(\log n)$ pairs of distinct elements. The $\mathcal{O}(\log n)$ elements that are different from the majority are then stored at the end of the array. After every $\log n$th insertion it is easy to check if there are now sufficient elements to make $\geq c \log n$ pairs for some appropriately large and fixed c. When that happens, the structure in Figure 2.5 is formed, and e_t must now contain two distinct elements, with the lesser being the threshold key. Note also, that while $e_{t,1} = e_{t,2}$ an ExtractMin procedure simply needs to scan the last $< c \log n$ elements and possibly make one swap to return the minimum and fill the empty index.

Insert The structure B' is a list of single elements which functions as an insertion buffer, that is elements are simply appended to B' when inserted. Whenever $n \bmod \log n = 0$ a procedure making pairs is run: At this point we have time to decode p_L, and up to $\mathcal{O}(\log n)$ new pairs can be made using L and B'. To make pairs B' is read, all elements in B' that are equal to elements in L, are put after L, the rest of the elements in B' are used to create pairs using one element from L and one element from B'. If there are more elements in B', they can be used to make pairs on their own. These pairs are then inserted into PQ. To make room for the newly inserted pairs, L might have to move right and we might have to update p_L. Since p_L is a Gray coded pointer, we only need as many bit changes as there are pairs inserted in PQ, ensuring $\mathcal{O}(1)$ amortized moves. Note that the size of PQ is now the value of p_L, which means all computations involving n for PQ should use p_L instead.

ExtractMin To extract the minimum a search for the minimum is performed in PQ, B' and L. If the minimum is in PQ, it is extracted and the other element in the pair is put at the end of B'. Now there are two empty positions before L, so the last two elements of L are put there, and the last two elements of B' are put in those positions. Note p_L also needs to be decremented. If the minimum is in B', it is swapped with the element at position n, and returned. If the minimum is in L, the last element of L is swapped with the element at position n, and it is returned.

Analysis Firstly observe that if we can prove the producing of pairs uses amortized $\mathcal{O}(1)$ moves for Insert and ExtractMin and $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ time respectively, then the rest of the analysis from Section 2.7.3 carries through. We first analyze Insert and then ExtractMin.

For Insert there are two variations: either append elements to B' or clean up B' and insert into PQ. Cleaning up B' and inserting into PQ is expensive and we amortize it over the cheap operations. Each operation that just appends to B' costs $\mathcal{O}(1)$ time and moves. Cleaning up B' requires decoding p_L, scanning B' and inserting $\mathcal{O}(\log n)$ elements in PQ. Note that between two clean-ups either $\mathcal{O}(\log n)$ elements have been inserted or there has been at least one ExtractMin, so we charge the time there. Since each insertion into PQ takes $\mathcal{O}(1)$ time and moves amortized we get the same bound when performing those insertions. The cost of reading p_L is $\mathcal{O}(\log n)$, but since we are guaranteed that either $\Omega(\log n)$ insertions have occurred or at least one ExtractMin operation we can amortize the reading time.

2.8 A Priority Queue with Worst Case Bounds

In this section we present a strictly implicit priority queue supporting Insert in worst-case $\mathcal{O}(1)$ time and ExtractMin in worst-case $\mathcal{O}(\log n)$ time (and moves). The data structure requires all elements to be distinct. The main concept used is a variation on binomial trees. The priority queue is a forest of $\mathcal{O}(\log n)$ such trees. We start with a discussion of the variant we call *relaxed binomial trees*, then we describe how to maintain a forest of these trees in an amortized sense, and finally we give the deamortization.

2.8.1 Relaxed binomial tree

Binomial trees are defined inductively: A single node is a binomial tree of size one and the node is also the root. A binomial tree of size 2^{i+1} is made by *linking* two binomial trees T_1 and T_2 both of size 2^i, such that one root becomes the rightmost child of the other root. We lay out in memory a binomial tree of size 2^i by a preorder traversal of the tree where children are visited in order of increasing size, i.e. $c_0, c_1, \ldots, c_{i-1}$. This layout is also described in [31]. See Figure 2.6 for an illustration of the layout. In a *relaxed binomial tree* (RBT)

each nodes stores an element, satisfying the following order: Let p be a node with i children, and let c_j be a child of p. Let T_{c_j} denote the set of elements in the subtree rooted at c_j. We have the invariant that the element c_ℓ is less than either all elements in T_{c_ℓ} or less than all elements in $\bigcup_{j<\ell} T_{c_j}$ (see Figure 2.6). In particular we have the requirement that the root must store the smallest element in the tree. In each node we store a flag indicating in which direction the ordering is satisfied. Note that linking two adjacent RBTs of equal size can be done in $\mathcal{O}(1)$ time: compare the keys of the two roots, if the lesser is to the right, swap the two nodes and finally update the flags to reflect the changes as just described.

For an unrelated technical purpose we also need to store whether a node is the root of a RBT. This information is encoded using three elements per node (allowing $3! = 6$ permutations, and we only need to differentiate between three states per node: "root", "minimum of its own subtree", or "minimum among strictly smaller subtrees").

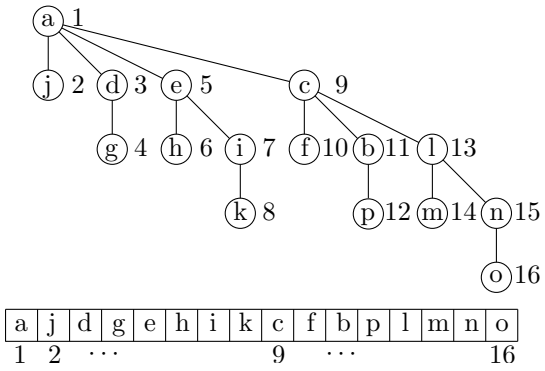


Figure 2.6: An example of an RBT on 16 elements (a,b,...,o). The layout in memory of an RBT and a regular binomial tree is the same. Note here that node 9 has element c and is not the minimum of its subtree because node 11 has element b, but c is the minimum among the subtrees rooted at nodes 2, 3, and 5 (c_0, c_1, and c_2). Note also that node 5 is the minimum of its subtree but not the minimum among the trees rooted at nodes 2 and 3, which means only one state is valid. Finally node 3 is the minimum of both its own subtree and the subtree rooted at node 2, which means both states are valid for that node.

To extract the minimum element of an RBT it is replaced by another element. The reason for replacing is that the forest of RBTs is implicitly maintained in an array and elements are removed from the right end, meaning only an element from the last RBT is removed. If the last RBT is of size 1, it is trivial to remove the element. If it is larger, then we *decompose* it. We first describe how to perform a DECOMPOSE operation which changes an RBT of size 2^i into i structures $T_{i-1}, \ldots, T_1, T_0$, where $|T_j| = 2^j$. Then we describe

how to perform ReplaceMin which takes one argument, a new element, and extracts the minimum element from an RBT and inserts the argument in the same structure.

A Decompose procedure is essentially reversing insertions. We describe a tail recursive procedure taking as argument a node r. If the structure is of size one, we are done. If the structure is of size 2^i the $(i-1)$th child, c_{i-1}, of r is inspected, if it is not the minimum of its own subtree, the element of c_{i-1} and r are swapped. The $(i-1)$th child should now encode "root", that way we have two trees of size 2^{i-1} and we recurse on the subtree to the right in the memory layout. This procedure terminates in $\mathcal{O}(i)$ steps and gives $i+1$ structures of sizes $2^{i-1}, 2^{i-2}, \ldots, 2, 1$, and 1 laid out in decreasing order of size (note there are two structures of size 1). This enables easy removal of a single element.

The ReplaceMin operation works similarly to the Decompose, where instead of always recursing on the right, we recurse where the minimum element is the root. When the recursion ends, the minimum element is now in a structure of size 1, which is deleted and replaced by the new element. The decomposition is then reversed by linking the RBTs using the Link procedure. Note it is possible to keep track of which side was recursed on at every level with $\mathcal{O}(\log n)$ extra bits, i.e. $\mathcal{O}(1)$ words. The operation takes $\mathcal{O}(\log n)$ steps and correctness follows by the Decompose and Link procedures. This concludes the description of RBTs and yields the following theorem.

Theorem 5. *On an RBT with $3 \cdot 2^i$ elements,* Link *and* FindMin *can be supported in $\mathcal{O}(1)$ time and* Decompose *and* ReplaceMin *in $\mathcal{O}(i)$ time.*

2.8.2 How to maintain a forest

As mentioned our priority queue is a forest of the relaxed binomial trees from Theorem 5. An easy amortized solution is to store one structure of size $3 \cdot 2^j$ for every set bit j in the binary representation of $\lfloor n/3 \rfloor$. During an insertion this could cause $\mathcal{O}(\log n)$ Link operations, but by a similar argument to that of binary counting, this yields $\mathcal{O}(1)$ amortized insertion time. We are aiming for a worst case constant time solution so we maintain the invariant that there are at most 5 structures of size 2^i for $i = 0, 1, \ldots, \lfloor \log n \rfloor$. This enables us to postpone some of the Link operations to appropriate times. We are storing $\mathcal{O}(\log n)$ RBTs, but we do not store which sizes we have, this information must be decodable in constant time since we do not allow storing additional words. Recall that we need 3 elements per node in an RBT, thus in the following we let n be the number of elements and $N = \lfloor n/3 \rfloor$ be the number of nodes. We say a node is in node position k if the three elements in it are in positions $3k-2$, $3k-1$, and $3k$. This means there is a buffer of $0, 1$, or 2 elements at the end of the array. When a third element is inserted, the elements in the buffer become an RBT with a single node and the buffer is now empty. If an Insert operation does not create a new node, the new element is simply appended

to the buffer. We are not storing the structure of the forest (i.e. how many RBTs of size 2^j exists for each j), since that would require additional space. To be able to navigate the forest we need the following two lemmas.

Lemma 4. *There is a structure of size 2^i at node positions $k, k+1, \ldots, k+2^i-1$ if and only if the node at position k encodes "root", the node at position $k+2^i$ encodes "root" and the node at position $k+2^{i-1}$ encodes "not root".*

Proof. It is trivially true that the three mentioned nodes encode "root", "root" and "not root" if an RBT with 2^i nodes is present in those locations.

We first observe there cannot be a structure of size 2^{i-1} starting at position k, since that would force the node at position $k+2^{i-1}$ to encode "root". Also all structures between k and N must have less than 2^i elements, since both nodes at positions k and $k+2^i$ encode "root". We now break the analysis in a few cases and the lemma follows from a proof by contradiction. Suppose there is a structure of size 2^{i-2} starting at k, then for the same reason as before there cannot be another one of size 2^{i-2}. Similarly, there can at most be one structure of size 2^{i-3} following that structure. Now we can bound the total number of nodes from position k onwards in the structure as: $2^{i-2} + 2^{i-3} + 5\sum_{j=0}^{i-4} 2^j = 2^i - 5 < 2^i$, which is a contradiction. So there cannot be a structure of size 2^{i-2} starting at position k. Note there can at most be three structures of size 2^{i-3} starting at position k, and we can again bound the total number of nodes as: $3 \cdot 2^{i-3} + 5\sum_{j=0}^{i-4} 2^j = 2^i - 5 < 2^i$, again a contradiction. □

Lemma 5. *If there is an RBT with 2^i nodes the root is in position $N - 2^i k - x + 1$ for $k = 1, 2, 3, 4$ or 5 and $x = N \bmod 2^i$.*

Proof. There are at most $5 \cdot 2^i - 5$ nodes in structures of size $\leq 2^{i-1}$. All structures of size $\geq 2^i$ contribute 0 to x, thus the number of nodes in structures with $\leq 2^{i-1}$ nodes must be x counting modulo 2^i. This gives exactly the five possibilities for where the first tree of size 2^i can be. □

We now describe how to perform an ExtractMin. First, if there is no buffer ($n \bmod 3 = 0$) then Decompose is executed on the smallest structure. We apply Lemma 5 iteratively for $i = 0$ to $\lfloor \log N \rfloor$ and use Lemma 4 to find structures of size 2^i. If there is a structure we call the FindMin procedure (i.e. inspect the element of the root node) and remember which structure the minimum element resides in. If the minimum element is in the buffer, it is deleted and the rightmost element is put in the empty position. If there is no buffer, we are guaranteed due to the first step that there is a structure with 1 node, which is now the buffer. On the structure with the minimum element ReplaceMin is called with the rightmost element of the array. The running time is $\mathcal{O}(\log n)$ for finding all the structures, $\mathcal{O}(\log n)$ for decomposing the smallest structure and $\mathcal{O}(\log n)$ for the ReplaceMin procedure, in total we get $\mathcal{O}(\log n)$ for ExtractMin.

The INSERT procedure is simpler but the correctness proof is somewhat involved. A new element is inserted in the buffer, if the buffer becomes a node, then the *least significant bit* i of N is computed. If at least two structures of size 2^i exist (found using the two lemmas above), then they are linked and become one structure of size 2^{i+1}.

Lemma 6. *The* INSERT *and* EXTRACTMIN *procedures maintain that at most five structures of size* 2^i *exist for all* $i \leq \lfloor \log n \rfloor$.

Proof. Let $N_{\leq i}$ be the total number of nodes in structures of size $\leq 2^i$. Then the following is an invariant for $i = 0, 1, \ldots, \lfloor \log N \rfloor$.

$$N_{\leq i} + (2^{i+1} - ((N + 2^i) \bmod 2^{i+1})) \leq 6 \cdot 2^i - 1$$

The invariant states that $N_{\leq i}$ plus the number of inserts until we try to link two trees of size 2^i is at most $6 \cdot 2^i - 1$. Suppose that a new node is inserted and i is not the least significant bit of N then $N_{\leq i}$ increases by one and so does $(N + 2^i) \bmod 2^{i+1}$, which means the invariant is maintained. Suppose that i is the least significant bit in N (i.e. we try to link structures of size 2^i) and there are at least two structures of size 2^i, then the insertion makes $N_{\leq i}$ decrease by $2 \cdot 2^i - 1 = 2^{i+1} - 1$ and $2^{i+1} - (N + 2^i \bmod 2^{i+1}))$ increases by $2^{i+1} - 1$, since $(N + 2^i) \bmod 2^{i+1}$ becomes zero, which means the invariant is maintained. Now suppose there is at most one structure of size 2^i and i is the least significant bit of N. We know by the invariant that $N_{\leq i-1} + (2^i - (N + 2^{i-1} \bmod 2^i)) \leq 6 \cdot 2^{i-1} - 1$ which implies $N_{\leq i-1} \leq 6 \cdot 2^{i-1} - 1 - 2^i + 2^{i-1} = 5 \cdot 2^{i-1} - 1$. Since we assumed there is at most one structure of size 2^i we get that $N_{\leq i} \leq 2^i + N_{\leq i-1} \leq 2^i + 5 \cdot 2^{i-1} - 1 = 3.5 \cdot 2^i - 1$. Since $N \bmod 2^{i+1} = 2^i$ (i is the least significant bit of N) we have $N_{\leq i} + (2^{i+1} - (N + 2^i \bmod 2^{i+1})) \leq 3.5 \cdot 2^i - 1 + 2^{i+1} = 5.5 \cdot 2^i - 1 < 6 \cdot 2^i - 1$.

The invariant is also maintained when deleting: for each i where $N_i > 0$ before the EXTRACTMIN, N_i decreases by one. For all i the second term increases by at most one, and possibly decreases by $2^{i+1}-1$. Thus the invariant is maintained for all i where $N_i > 0$ before the procedure. If $N_i = 0$ before an EXTRACTMIN, we get $N_j = 2^{j+1} - 1$ for $j \leq i$. Since the second term can at most contribute 2^{j+1}, we get $N_j + (2^{j+1} - ((N + 2^j) \bmod 2^{j+1})) \leq 2^{j+1} - 1 + 2^{j+1} \leq 6 \cdot 2^j - 1$, thus the invariant is maintained. □

Correctness and running times of the procedures have now been established.

Conclusion With the two priority queues presented we now conclude the chapter on implicit data structures. We note that the priority queue with $\mathcal{O}(1)$ amortized moves is difficult to implement, and probably have some significant constant factors overhead. However, the worst case efficient priority queue is fairly easy to implement and has relatively low constant factors, its primary

drawback is the lack of support for identical elements. We do have a working implementation of the worst case efficient priority queue, and future work includes testing if it performs well against other priority queues. One point that might be slow is when testing if there is a structure of size 2^i, since that may incur several cache misses that are expensive.

Chapter 3

Interlude: Sorting

Sorting is one of the most fundamental problems in computer science and has been studied widely in many different computational models. Sorting n integers in the word-RAM model is a fundamental problem and a long-standing open problem is whether integer sorting is possible in linear time when the word size is $\omega(\log n)$. Recall that radix sort takes $\mathcal{O}(n)$ time when the word size is $\mathcal{O}(\log n)$. The radix sorting procedure with words size $w = \mathcal{O}(\log n)$ intuitively works by writing each integer in base $\lfloor n \rfloor$, and then stably sorting the numbers in $\mathcal{O}(1)$ rounds, first round by the least significant digit and the last round by the most significant digit.

In this chapter we give an algorithm for sorting integers in expected linear time when the word size is $\Omega(\log^2 n \log\log n)$. Previously expected linear time sorting was only possible for word size $\Omega(\log^{2+\varepsilon} n)$. Part of our construction is a new packed sorting algorithm that sorts n integers of w/b-bits packed in $\mathcal{O}(n/b)$ words, where b is the number of integers packed in a word of size w bits. The packed sorting algorithm runs in expected $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$ time.

In the comparison based setting both the worst case and average case complexity of sorting n elements is $\Theta(n \log n)$ comparisons. This $\mathcal{O}(n \log n)$ bound is achieved by several algorithms, the most well-known are Mergesort, Randomized Quicksort and Heapsort. The lower bound is proved using decision trees, see e.g. [48], and is also valid in the average case. Note in Section 2.7 we presented an implicit priority queues that enables us to sort n comparable items in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ moves.

In the word-RAM model with word size $w = \Theta(\log n)$ we can sort n w-bit integers in $\mathcal{O}(n)$ time using radix sort. The exact bound for sorting n integers of w bits each using radix sort is $\Theta(n\frac{w}{\log n})$. A fundamental open problem is if we can still sort in linear time when the word size is $\omega(\log n)$ bits. The RAM dictionary of van Emde Boas [118] allows us to sort in $\mathcal{O}(n \log w)$ time. Unfortunately the space usage by the van Emde Boas structure cannot

This chapter is based on the paper Expected Linear Time Dorting for Word Size $\Omega(\log^2 n \log\log n)$[16]

be bounded better than $\mathcal{O}(2^w)$. The space usage can be reduced to $\mathcal{O}(n)$ by using the Y-fast trie of Willard [120], but the time bound for sorting becomes expected. For polylogarithmic word sizes, i.e. $w = \log^{\mathcal{O}(1)} n$, this gives sorting in time $\mathcal{O}(n \log\log n)$. Kirkpatrick and Reisch gave an algorithm achieving $\mathcal{O}(n \log \frac{w}{\log n})$ [81], which also gives $\mathcal{O}(n \log\log n)$ for $w = \log^{\mathcal{O}(1)} n$. Andersson et al. [10] showed how to sort in expected $\mathcal{O}(n)$ time for word size $w = \Omega(\log^{2+\varepsilon} n)$ for any $\varepsilon > 0$. The result is achieved by exploiting word parallelism on "signatures" of the input elements packed into words, such that a RAM instruction can perform several element operations in parallel in constant time. Han and Thorup [69] achieved running time $\mathcal{O}(n\sqrt{\log(w/\log n)})$, implying the best known bound of $\mathcal{O}(n\sqrt{\log\log n})$ for sorting integers that is independent of the word size. Thorup established that maintaining RAM priority queues and RAM sorting are equivalent problems by proving that if we can sort in time $\mathcal{O}(n \cdot f(n))$ then there is a priority queue using $\mathcal{O}(f(n))$ time per operation [117].

Our results. We consider for which word sizes we can sort n w-bit integers in the word-RAM model in expected linear time. We improve the previous best word size of $\Omega(\log^{2+\varepsilon} n)$ [10] to $\Omega(\log^2 n \log\log n)$. Word-level parallelism is used extensively and we rely on a new packed sorting algorithm (see Section 3.4) in intermediate steps. The principal idea for the packed sorting algorithm is an implementation of the randomized Shell-sort of Goodrich [64] using the parallelism in the RAM model. The bottleneck in our construction is $\mathcal{O}(\log\log n)$ levels of packed sorting of $\mathcal{O}(n)$ elements each of $\Theta(\log n)$ bits, where each sorting requires time $\mathcal{O}(n\frac{\log^2 n}{w})$. For $w = \Omega(\log^2 n \log\log n)$, the overall time becomes $\mathcal{O}(n)$.

This chapter is structured as follows: Section 3.1 contains a high level description of the ideas and concepts used by our algorithm. In Section 3.2 we summarize the RAM operations adopted from [10] that are needed to implement the algorithm outlined in Section 3.1. In Section 3.3 we give the details of implementing the algorithm on a RAM and in Section 3.4 we present the packed sorting algorithm. Finally, in Section 3.5 we discuss how to adapt our algorithm to work with an arbitrary word size.

3.1 Algorithm

In this section we give a high level description of the algorithm. The input is n words $x_1, x_2, \ldots, x_n$, each containing a w-bit integer from $U = \{0, 1, \ldots, 2^w{-}1\}$. We assume the elements are distinct. Otherwise we can ensure this by hashing the elements into buckets in expected $\mathcal{O}(n)$ time and only sorting a reduced input with one element from each bucket. The algorithm uses a Monte Carlo procedure, which sorts the input with high probability. While the output is

not sorted, we repeatedly rerun the Monte Carlo algorithm, turning the main sorting algorithm into a Las Vegas algorithm.

The Monte Carlo algorithm is a recursive procedure using geometrically decreasing time in the recursion, ensuring $\mathcal{O}(n)$ time overall. We view the algorithm as building a Patricia trie over the input words by gradually refining the Patricia trie in the following sense: on the outermost recursion level characters are considered to be w bits long, on the next level $w/2$ bits, then $w/4$ bits and so on. The main idea is to avoid considering all the bits of an element to decide its rank. To avoid looking at every bit of the bit string e at every level of the recursion, we either consider the MSH(e) (Most Significant Half, i.e. the $\frac{|e|}{2}$ most significant bits of e) or LSH(e) (Least Significant Half) when moving one level down in the recursion (similar to the recursion in van Emde Boas trees).

The input to the ith recursion is a list $(id_1, e_1), (id_2, e_2), \ldots, (id_m, e_m)$ of length m, where $n \leq m \leq 2n-1$, id_j is a $\log n$ bit id and e_j is a $w/2^i$ bit element. At most n elements have equal id. The output is a list of ranks $\pi_1, \pi_2, \ldots, \pi_m$, where the j'th output is the rank of e_j among elements with id identical to id_j using $\log n$ bits. There are $m(\log n + \frac{w}{2^i})$ bits of input to the ith level of recursion and $m \log n$ bits are returned from the ith level. On the outermost recursion level we take the input $x_1, x_2, \ldots, x_n$ and produce the list $(1, x_1), (1, x_2), \ldots, (1, x_n)$, solve this problem, and use the ranks $\pi_1, \pi_2, \ldots, \pi_n$ returned to permute the input in sorted order in $\mathcal{O}(n)$ time.

To describe the recursion we need the following definitions.

Definition 1 ([53])**.** *The* Patricia trie *consists of all the branching nodes and leaves of the corresponding compacted trie as well as their connecting edges. All the edges in the Patricia trie are labeled only by the first character of the corresponding edge in the compacted trie.*

Definition 2. *The Patricia trie of $x_1, x_2, \ldots, x_n$ of* detail *i, denoted T^i, is the* Patricia trie *of $x_1, \ldots, x_n$ when considered over the alphabet $\Sigma^i = \{0,1\}^{w/2^i}$.*

The input to the ith recursion satisfies the following invariants, provided the algorithm has not made any errors so far:

i. The number of bits in an element is $|e| = \frac{w}{2^i}$.

ii. There is a bijection from id's to non leaf nodes in T^i.

iii. The pair (id, e) is in the input if and only if there is an edge from a node $v \in T^i$ corresponding to id to a child labeled by a string in which $e \in \Sigma^i$ is the first character.

That the maximum number of elements at any level in the recursion is at most $2n-1$ follows because a Patricia trie on n strings has at most $2n-1$ edges.

The recursion. The base case of the recursion is when $|e| = \mathcal{O}(\frac{w}{\log n})$ bits, i.e. we can pack $\Omega(\log n)$ elements into a single word, where we use the packed sorting algorithm from Section 3.4 to sort (id_j, e_j, j) pairs lexicographically by (id, e) in time $\mathcal{O}(\frac{n}{\log n}(\log n + (\log\log n)^2)) = \mathcal{O}(n)$. Then we generate the ranks π_j and return them in the correct order by packed sorting pairs (j, π_j) by j.

When preparing the input for a recursive call we need to halve the number of bits the elements use. To maintain the second invariant we need to find all the branching nodes of T^{i+1} to create a unique id for each of them. Finally for each edge going out of a branching node v in T^{i+1} we need to make the pair (id, e), where id is v's id and e is the first character (in Σ^{i+1}) on an edge below v. Compared to level i, level $i+1$ may have two kinds of branching nodes: *inherited nodes* and *new nodes*, as detailed below (Figure 3.1).

In Figure 3.1 we see T^i and T^{i+1} on 5 binary strings. In T^i characters are 4 bits and in T^{i+1} they are 2 bits. Observe that node a is not going to be a branching node when characters are 2 bits because "00" are the first bits on both edges below it. Thus the "00" bits below a should not appear in the next recursion – this is captured by Invariant iii. A similar situation happens at the node b, however since there are *two different* 2-bit strings below it, we get the inherited node b_{inh}. At the node c we see that the order among its edges is determined by the first two bits, thus the last two bits can be discarded. Note there are 7 elements in the ith recursion and 8 in the next – the number of elements may increase in each recursion, but the maximum amount is bounded by $2n - 2$.

By invariant ii) every id corresponds to a node v in T^i. If we find all elements that share the same id, then we have all the outgoing edges of v. We refine an edge labeled e out of v to have the two characters MSH(e)LSH(e) both of $w/2^{i+1}$ bits. Some edges might then share their MSH. The node v will appear in level $i+1$ if and only if at least two outgoing edges do not share MSH – these are the *inherited nodes*. Thus we need only count the number of unique MSHs out of v to decide if v is also a node in level $i+1$. The edges out of v at level $i+1$ will be the unique MSH characters (in Σ^{i+1}) on the edges down from v at level i.

If at least two edges out of v share the same first character c (MSH), but not the second, then there is a branching node following c – these are the *new nodes*. We find all new nodes by detecting for each MSH character $c \in \Sigma^{i+1}$ going out of v if there are two or more edges with c as their first character. If so, we have a branching node following c and the labels of the edges are the LSHs. At this point everything for the recursion is prepared.

We receive for each id/node of T^{i+1} the ranks of all elements (labels on the outgoing edges) from the recursion. A *relative* rank for an element at level i is created by concatenating the rank of MSH(e) from level $i+1$ with the rank of LSH(e) from level $i+1$. All edges branching out of a new node needs to receive the rank of their MSH (first character). If the MSH was not used for

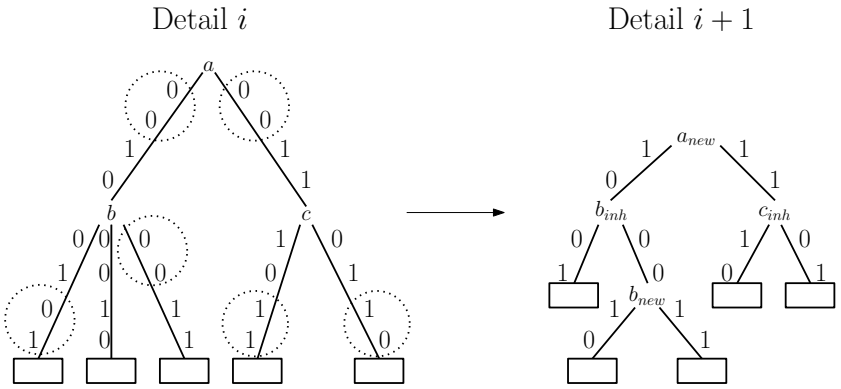


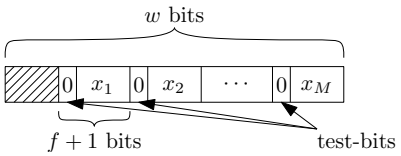
Figure 3.1: Example of how nodes are introduced and how they disappear from detail i to $i+1$. The bits that are marked by a dotted circle are omitted in the recursion.

the recursion, it means it did not distinguish any edges, and we can put an arbitrary value as the rank (we use 0). The same is true for the LSHs. Since each relative rank consists of $2 \log n$ bits we can sort them fast using packed sorting (Section 3.4) and finally the actual ranks can be returned based on that.

3.2 Tools

This section is a summary of standard word-parallel algorithms used by our sorting algorithm; for an extensive treatment see [82]. In particular the prefix sum and word packing algorithms can be derived from [86]. For those familiar with "bit tricks" this section can be skipped.

We adopt the notation and techniques used in [10]. A w-bit word can be interpreted as a single integer in the range $0, \ldots, 2^w - 1$ or the interpretation can be parameterized by (M, f). A word under the (M, f) interpretation uses the rightmost $M(f+1)$ bits as M fields using $f+1$ bits each and the most significant bit in each field is called the *test bit* and is 0 by default.



We write $X = (x_1, x_2, \ldots, x_M)$ where x_i uses f bits, meaning the word X has the integer x_1 encoded in its leftmost field, x_2 in the next and so on. If $x_i \in \{0, 1\}$ for all i we may also interpret them as boolean values where 0 is false and 1 is true. This representation allows us to do "bit tricks".

Comparisons. Given a word $X = (x_1, x_2, \ldots, x_M)$ under the (M, f) interpretation, we wish to test $x_i > 0$ for $1 \leq i \leq M$, i.e. we want a word $Z = [X > 0] = (z_1, z_2, \ldots, z_M)$, in the (M, f) interpretation, such that $z_i = 1$ (true) if $x_i > 0$ and $z_i = 0$ (false) otherwise. Let $k_{M,f}$ be the word where the number k is encoded in each field where $0 \leq k < 2^f$. Create the word $0_{M,f}$ and set all test bits to 1. Evaluate $\neg(0_{M,f} - X)$, the ith test bit is 1 if and only if $x_i > 0$. By masking away everything but the test bit and shifting right by f bits we have the desired output. We can also implement more advanced comparisons, such as comparing $[X \leq Y]$ by setting all test bits to 1 in Y and 0 in X and subtracting the word X from Y. The test bits now equal the result of comparing $x_i \leq y_i$.

Hashing. We will use a family of hash functions that can hash n elements in some range $0, \ldots, m-1$ with $m > n^c$ to $0, \ldots n^c - 1$. Furthermore a family of hash functions that are injective on a set with high probability when chosen uniformly at random, can be found in [51]. Hashing is roughly just multiplication by a random odd integer and keeping the most significant bits. The integer is at most f bits. If we just multiply this on a word in (M, f) interpretation one field might overflow to the next field, which is undesirable. To implement hashing on a word in (M, f) representation we first mask out all even fields, do the hashing, then do the same for odd fields. The details can be found in [10]. In [51] it is proved that if we choose a function h_a uniformly at random from the family $H_{k,\ell} = \{h_a \mid 0 < a < 2^k, \text{ and } a \text{ is odd}\}$ where $h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell}$ for $0 \leq x < 2^k$ then $\Pr[h_a(x) = h_a(y)] \leq \frac{1}{2^{\ell-1}}$ for distinct x, y from a set of size n. Thus choosing $\ell = c \log n + 1$ gives collision probability $\leq 1/n^c$. The probability that the function is not injective on n elements is upper bounded by the union bound on all pairs of elements as $\Pr[\exists x, y : x \neq y \wedge h_a(x) = h_a(y)] \leq \frac{n^2}{n^c}$.

Prefix sum. Let $A = (a_1, \ldots, a_M)$ be the input with $M = b$, $f = w/b$ and $a_i \in \{0, 1\}$. In the output $B = (b_1, \ldots, b_M)$, $b_i = 0$ if $a_i = 0$ and $b_i = \sum_{j=1}^{i-1} a_j$ otherwise. We describe an $\mathcal{O}(\log b)$ time algorithm to accomplish this task. The invariant is that in the jth iteration a_i has been added to its 2^j immediately right adjacent fields. Compute B_1, which is A shifted right by f bits and added to itself: $B_1 = A + (A \downarrow f)$. Let $B_i = (B_{i-1} \downarrow 2^{i-1} f) + B_{i-1}$. This continues for $\log b$ steps. Then we keep all fields i from $B_{\log b}$ where $a_i = 1$, subtract 1 from all of these fields and return it.

Packing words. We are given a word $X = (x_1, \ldots, x_M)$ in $(M, f) = (b, w/b)$ representation. Some of the fields are zero fields, i.e. a field only containing bits set to 0. We want to produce a "packed word", such that reading from left to right there are no zero fields, followed only by zero fields. The fields

We use $\uparrow$ and $\downarrow$ as the shift operations where $x \uparrow y$ is $x \cdot 2^y$ and $x \downarrow y$ is $\lfloor x \operatorname{div} 2^y \rfloor$.

that are nonzero in the input must be in the output and in the same order. This problem is solved by Andersson et al. [10, Lemma 6.4]

Expanding. Given a word with fields using b' bits we need to expand each field to using b bits i.e., given $X = (x_1, \ldots, x_k)$ where $|x_i| = b'$ we want $Y = (y_1, \ldots, y_k)$ such that $y_i = x_i$ but $|y_i| = b$. We assume there are enough zero fields in the input word such that the output is only one word. The general idea is to just do packing backwards. The idea is to write under each field the number of bits it needs to be shifted right, this requires at most $\mathcal{O}(\log b)$ bits per field. We now move items based on the binary representation. First we move those who have the highest bit set, then we continue with those that have the second highest bit set and so on. The proof that this works is the same as for the packing algorithm.

Creating index. We have a list of n elements of w/b bits each, packed in an array of words $X_1, X_2, \ldots, X_{n/b}$, where each word is in $(b, w/b)$ representation and $w/b \geq \lceil \log n \rceil$. Furthermore, the rightmost $\lceil \log n \rceil$ bits in every field are 0. The index of an element is the number of elements preceding it and we want to put the index in the rightmost bits of each field. First we will spend $\mathcal{O}(b)$ time to create the word $A = (1, 2, 3, \ldots, b)$ using the rightmost bits of the fields. We also create the word $B = (b, b, \ldots, b)$. Now we run through the input words, update $X_i = X_i + A$, then update $A = A + B$. The time is $\mathcal{O}(n/b + b)$, which in our case always is $\mathcal{O}(n/b)$, since we always have $b = \mathcal{O}(\log n \log \log n)$.

3.3 Algorithm – RAM details

In this section we describe how to execute each step of the algorithm outlined in Section 3.1. We first we describe how to construct T^{i+1} from T^i, i.e. advance one level in the recursion. Then we describe how to use the output of the recursion for T^{i+1} to get the ranks of the input elements for level i. Finally the analysis of the algorithm is given.

The input to the ith recursion is a list of pairs: (id, e) using $\log n + \frac{w}{2^i}$ bits each and satisfying the invariants stated in Section 3.1. The list is packed tightly in words, i.e. if we have m input elements they occupy $\mathcal{O}\left(\frac{m \cdot (\log n + w/2^i)}{w}\right)$ words. The returned ranks are also packed in words, i.e. they occupy $\mathcal{O}(\frac{m \cdot \log n}{w})$ words. The main challenge of this section is to be able to compute the necessary operations, even when the input elements and output ranks are packed in words. For convenience and simplicity we assume in the following that tuples are not split between words.

Finding branching nodes. We need to find the branching nodes (*inherited* and *new*) of T^{i+1} given T^i. For each character e_j in the input list (i.e. T^i) we create the tuple (id_j, H_j, j) where id_j corresponds to the node e_j branches out

of, $H_j = h(\text{MSH}(e_j))$ is the hash function applied to the MSH of e_j, and j is the index of e_j in the input list. The list L consists of all these tuples and L is sorted. We assume the hash function is injective on the set of input MSHs, which it is with high probability if $|H_j| \geq 4 \log n$ (see the analysis below). If the hash function is not injective, this step may result in an error which we will realize at the end of the algorithm, which was discussed in Section 3.1. The following is largely about manipulating the order of the elements in L, such that we can create the recursive sub problem, i.e. T^{i+1}.

To find *Inherited nodes* we find all the edges out of nodes that are in both T^i and T^{i+1} and pair them with unique identifiers for their corresponding nodes in T^{i+1}. Consider a fixed node a which is a branching node in T^i – this corresponds to an id in L. There is a node a_{inh} in T^{i+1} if a and its edges satisfy the following condition: When considering the labels of the edges from a to its children over the alphabet Σ^{i+1} instead of Σ^i, there are at least two edges from a to its children that do not share their first character. When working with the list L the node a and its edges correspond to the tuples where the id is the id that corresponds to a. This means we need to compute for each id in L whether there are at least 2 unique MSHs, and if so we need to extract precisely all the unique MSHs for that id.

The list L is sorted by (id_j, H_j, j), which means all edges out of a particular node are adjacent in L, and all edges that share their MSH are adjacent in L (with high probability), because they have the same hash value which is distinct from the hash value of all other MSHs (with high probability). We select the MSHs corresponding to the first occurrence of each unique hash value with a particular id for the recursion (given that it is needed). To decide if a tuple contains a first unique hash value, we need only consider the previous tuple: did it have a different hash value from the current, or did it have a different id? To decide if $\text{MSH}(e_j)$ should be extracted from the corresponding tuple we also need to compute whether there are at least two unique hash values with id id_j. This tells us we need to compute two things for every tuple (id_j, H_j, j) in L:

1. Is j the first index such that $(id_{j-1} = id_j \wedge H_{j-1} \neq H_j) \vee id_{j-1} \neq id_j$?
2. Is there an i such that $id_i = id_j$ and $H_i \neq H_j$?

To accomplish the first task we do parallel comparison of id_j and H_j with id_{j-1} and H_{j-1} on L and L shifted left by one tuple length (using the word-level parallel comparisons described in Section 3.2). The second task is tedious but conceptually simple to test: count the number of unique hash values for each id, and test for each id if there are at least two unique hash values.

The details of accomplishing the two tasks are as follows (keep in mind that elements of the lists are bit-strings). Let B be a list of length $|L|$ and consider each element in B as being the same length as a tuple in L. Encode 1 in element j of B if and only if $id_{j-1} \neq id_j$ in L. Next we create a list C with

the same element size as B. There will be a 1 in element j of C if and only if $H_j \neq H_{j-1} \wedge id_j = id_{j-1}$ (this is what we needed to compute for task 1). The second task is now to count how many 1s there are in C between two ones in B. Let CC be the prefix sum on C (described in Section 3.2) and keep only the values where there is a corresponding 1 in B, all other elements become 0 (simple masking). Now we need to compute the difference between each non-zero value and the next non-zero value in CC – but these are varying lengths apart, how do we subtract them? The solution is to pack the list CC (see Section 3.2) such that the values become adjacent. Now we compute the difference, and by maintaining some information from the packing we can unpack the differences to the same positions that the original values had. Now we can finally test for the first tuple in each *id* if there are at least two different hash values with that *id*. That is, we now have a list D with a 1 in position j if j is the first position of an *id* in L and there are at least two unique MSHs with that *id*. In addition to completing the two tasks we can also compute the unique identifiers for the inherited nodes in T^{i+1} by performing a prefix sum on D.

Finding the *new nodes* is simpler than finding the *inherited nodes*. The only case where an LSH should be extracted is when two or more characters out of a node share MSH, in which case all the LSHs with that MSH define the outgoing edges of a *new node*. Observe that if two characters share MSH then their LSHs must differ, due to the assumption of distinct elements propagating through the recursion. To find the relevant LSHs we consider the sorted list L. Each *new node* is identified by a pair (id, MSH) where $(id_j, h(\text{MSH}(e_j)), \cdot)$ appears at least twice in L, i.e. two or more tuples with the same *id* and hash of MSH. For each *new node* we find the leftmost such tuple j in L.

Technically we scan through L and evaluate $(H_{j-1} \neq H_j \vee id_{j-1} \neq id_j) \wedge (H_{j+1} = H_j \wedge id_{j+1} = id_j)$. If this evaluates to true then j is a *new node* in T^{i+1}. Using a prefix sum we create and assign all *id*s for *new nodes* and their edges. In order to test if LSH_j should be in the recursion we evaluate $(H_{j-1} = H_j \wedge id_{j-1} = id_j) \vee (H_{j+1} = H_j \wedge id_{j+1} = id_j)$. This evaluates to true only if the LSH should be extracted for the recursion because we assume distinct elements.

Using results from the recursion. We created the input to the recursion by first extracting all MSHs, packing them and afterwards extracting all LSHs and then packing them. Finally concatenate the two packed arrays. Now we simply have to reverse this process, first for the MSHs, then the LSHs. Technically after the recursive call the array consisting of tuples $(j, rank_{\text{MSH}_j}, rank_{\text{LSH}_j}, H_j, id_j, rank_{new})$ is filled out. Some of the fields are just additional fields to the array L. The three ranks use $\log n$ bits each and are initialized to 0. First $rank_{\text{MSH}_j}$ is filled out and afterwards $rank_{\text{LSH}_j}$. The same procedure is used for both.

For retrieving $rank_{\text{MSH}_i}$, we know how many MSHs were extracted for the recursion, so we separate the ranks of MSHs and LSHs and now only consider MSHs ranks. We first expand the MSH ranks as described in Section 3.2 such that each rank uses the same number of bits as an entire tuple. Recall that the MSHs were packed and we now need to unpack them. If we saved information on how we packed elements, we can also unpack them. The information we need to retain is how many elements each word contributed and for each element in a word its initial position in that word. Note that for each unique H_j we only used one MSH for the recursion, thus we need to propagate its rank to all other elements with the same hash and *id*. Fortunately the hash values are adjacent, and by noting where the hash values change we can do an operation similar to a prefix sum to copy the ranks appropriately.

Returning. As this point the only field not filled out is $rank_{new}$. To fill it out we sort the list by the concatenation of $rank_{\text{MSH}_j}$ and $rank_{\text{LSH}_j}$. In this sorted list we put the current position of the elements in $rank_{new}$ (see Section 3.2 on creating index). The integer in $rank_{new}$ is currently not the correct rank, but by subtracting the first $rank_{new}$ in an *id* from the other $rank_{new}$s with that *id* we get the correct rank. Then we sort by j, mask away everything except $rank_{new}$, pack the array and return. We are guaranteed the ranks from the recursion use $\log n$ bits each, which means the concatenation uses $2 \log n$ bits so we can sort the array efficiently.

Analysis. We argue that the algorithm is correct and runs in linear time.

Lemma 7. *Let n be the number of integers we need to sort then the maximum number of elements in any level of the recursion is $2n - 1$.*

Proof. This follows immediately from the invariants. □

Theorem 6. *The main algorithm runs in $\mathcal{O}(n)$ time.*

Proof. At level i of the recursion $|e| = \frac{w}{2^i}$. After $\log \log n$ levels we switch to the base case where there are $b = 2^{\log \log n} = \log n$ elements per word. The time used in the base case is $\mathcal{O}(\frac{n}{b}(\log^2 b + \log n)) = \mathcal{O}(\frac{n}{\log n}((\log \log n)^2 + \log n)) = \mathcal{O}(n)$.

At level i of the recursion we have $b = 2^i$ elements per word and the time to work with each of the $\mathcal{O}(\frac{n}{b})$ words using the methods of Section 3.2 is $\mathcal{O}(\log b)$. The packed sorting at each level sorts elements with $\mathcal{O}(\log n)$ bits, i.e. $\mathcal{O}\left(\frac{w}{\log n}\right)$ elements per word in time $\mathcal{O}\left(\frac{n}{w/\log n}\left(\log^2 \frac{w}{\log n} + \log n\right)\right)$. Plugging in our assumption $w = \Omega(\log^2 n \log \log n)$, we get time $\mathcal{O}\left(\frac{n}{\log \log n}\right)$. For all levels the total time becomes $\sum_{i=0}^{\log \log n}\left(\frac{n}{2^i} i + \frac{n}{\log \log n}\right) = \mathcal{O}(n)$. □

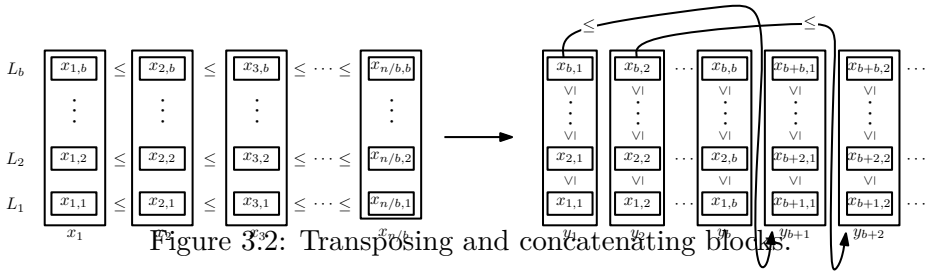


Figure 3.2: Transposing and concatenating blocks.

The probability of doing more than one iteration of the algorithm is the probability that there is a level in the recursion where the randomly chosen hash function was not injective. The hash family can be designed such that the probability of a hash function not being injective when chosen uniformly at random is less than $1/n^2$ [51]. We need to choose $\log\log n$ such functions. The probability that at least one of the functions is not injective is $\mathcal{O}(\log\log n/n^2) < \mathcal{O}(1/n)$. In conclusion the sorting step works with high probability, thus we expect to repeat it $\mathcal{O}(1)$ times.

3.4 Packed sorting

We used a subroutine in the main sorting algorithm for sorting the hash values. In this section we describe how perform that subroutine. We are given n elements of $\frac{w}{b}$ bits packed into $\frac{n}{b}$ words using $(M, f) = (b, w/b)$ representation that we need to sort. Albers and Hagerup [8] describe how to perform a deterministic packed sorting in time $\mathcal{O}(\frac{n}{b}\log n \cdot \log b)$. We describe a simple randomized word-level parallel sorting algorithm running in time $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$. Packed sorting proceeds in four steps described in the following sections. The idea is to implement b sorting networks in parallel using word-level parallelism. In sorting networks one operation is available: compare the elements at positions i and j then swap i and j based on the outcome of the comparison. Denote the ℓth element of word i at any point by $x_{i,\ell}$. First we use the ℓth sorting network to get a sorted list L_ℓ: $x_{1,\ell} \leq x_{2,\ell} \leq \cdots \leq x_{n/b,\ell}$ for $1 \leq \ell \leq b$. Each L_ℓ then occupies field ℓ of every word. Next we reorder the elements such that each of the b sorted lists uses n/b^2 consecutive words, i.e. $x_{i,j} \leq x_{i,j+1}$ and $x_{i,w/b} \leq x_{i+1,1}$, where $n/b^2 \cdot k < i \leq n/b^2 \cdot (k+1)$ and $0 \leq k \leq b-1$ (See Figure 3.2). From that point we can merge the lists using the RAM implementation of bitonic merging (see below). The idea of using sorting networks or *oblivious* sorting algorithms is not new (see e.g. [68]), but since we need to sort in sublinear time (in the number of elements) we use a slightly different approach.

Data-oblivious sorting. A famous result is the AKS deterministic sorting network which uses $\mathcal{O}(n\log n)$ comparisons [7]. Other deterministic $\mathcal{O}(n\log n)$ sorting networks were presented in [8, 65]. However, in our application ran-

domized sorting suffices so we use the simpler randomized Shell-sort by Goodrich [64]. An alternative randomized sorting-network construction was given by Leighton and Plaxton [87].

Randomized Shell-sort sorts any permutation with probability at least $1 - 1/N^c$ ($N = n/b$ is the input size), for any $c \geq 1$. We choose $c = 2$. The probability that b arbitrary lists are sorted is then at least $1 - b/N^c \geq 1 - N^{c-1}$. We check that the sorting was correct for all the lists in time $\mathcal{O}(\frac{n}{b})$. If not, we rerun the oblivious sorting algorithm (with new random choices). Overall the expected running time is $\mathcal{O}(\frac{n}{b} \log \frac{n}{b})$.

The Randomized Shell-sort algorithm works on any adversarial chosen permutation that does not know the random choices of the algorithm. The algorithm uses randomization to generate a sequence of $\Theta(n \log n)$ comparisons (a sorting network) and then applies the sequence of comparisons to the input array. We start the algorithm of Goodrich [64] to get the sorting network. We run it with $N = n/b$ as the input size. When the network compares i and j, we compare words i and j field-wise. That is, the first element of the two words are compared, the second element of the words are compared and so on. Using the result we can implement the swap that follows. After this step we have $x_{1,\ell} \leq x_{2,\ell} \leq \cdots \leq x_{n/b,\ell}$ for all $1 \leq \ell \leq b$.

The property of Goodrich' Shellsort that makes it possible to apply it in parallel is its data obliviousness. In fact any sufficiently fast data oblivious sorting algorithm would work.

Verification step. The verification step proceeds in the following way: we have n/b words and we need to verify that the words are sorted field-wise. That is, to check that $x_{i,\ell} \leq x_{i+1,\ell}$ for all i, ℓ. One packed comparison will be applied on each pair of consecutive words to verify this. If the verification fails, then we redo the oblivious sorting algorithm.

Rearranging the sequences. The rearrangement in Figure 3.2 corresponds to looking at b words as a $b \times b$ matrix (b words with b elements in each) and then transposing this matrix. Thorup [116, Lemma 9] solved this problem in $\mathcal{O}(b \log b)$ time. We transpose every block of b consecutive words. The transposition takes overall time $\mathcal{O}(\frac{n}{b} \log b)$. Finally, we collect in correct order all the words of each run. This takes time $\mathcal{O}(\frac{n}{b})$. Building the ith run for $1 \leq i \leq b$ consists of putting together the ith words of the blocks in the block order. This can be done in a linear scan in $\mathcal{O}(n/b)$ time.

Bitonic merging. The last phase is the bitonic merging. We merge pairs of runs of $\frac{n}{b^2}$ words into runs of $\frac{2n}{b^2}$ words, then runs of $\frac{2n}{b^2}$ words into runs of size $\frac{4n}{b^2}$ and so on, until we get to a single run of n/b words. We need to do $\log b$ rounds, each round taking time $\mathcal{O}(\frac{n}{b} \log b)$ making for a total time of $\mathcal{O}(\frac{n}{b} \log^2 b)$ [8].

3.5 General sorting

In this section we tune the algorithm slightly and state the running time of the tuned algorithm in terms of the word size w. We see that for some word sizes we can beat the $\mathcal{O}(n\sqrt{\log\log n})$ bound. We use the splitting technique of [69, Theorem 7] that given n integers can partition them into sets $X_1, X_2, \ldots X_k$ of at most $\mathcal{O}(\sqrt{n})$ elements each, such that all elements in X_i are less than all elements in X_{i+1} in $\mathcal{O}(n)$ time. Using this we can sort in $\mathcal{O}(n \log \frac{\log n}{\sqrt{w/\log w}})$ time. The algorithm repeatedly splits the set S of inital size n_0 into smaller subsets of size $n_j = \sqrt{n_{j-1}}$ until we get $\log n_j \leq \sqrt{w/\log w}$ where it stops and sorts each subset in linear time using our sorting algorithm. The splitting is performed $\log((\log n)/(\sqrt{w/\log w})) = \frac{1}{2}\log\frac{\log^2 n \log w}{w} = \mathcal{O}(\log\frac{\log^2 n \log\log n}{w})$ times. An interesting example is to sort in time $\mathcal{O}(n\log\log\log n)$ for $w = \frac{\log^2 n}{(\log\log n)^c}$ for any constant c. When $w = \frac{\log^2 n}{2^{\Omega(\sqrt{\log\log n})}}$, the sorting time is $\Omega(n\sqrt{\log\log n})$ which matches the bound of Thorup and Han. In contrast the exact bound by Thorup and Han is $O\left(n\sqrt{\log\frac{w}{\log n}}\right)$ which becomes $\Theta\left(n\sqrt{\log\log n}\right)$ when $w = \Omega\left(\log^{1+\varepsilon} n\right)$ and $w = \log^{\mathcal{O}(1)} n$. This means the asymptotic of Thorup and Han's algorithm rises very quickly to $\mathcal{O}(n\sqrt{\log\log n})$ and then stabilizes. Our algorithm converges towards $\mathcal{O}(n\log\log n)$ as w decreases towards $\Theta(\log n)$.

Chapter 4

Text Indexing

In this chapter we deal with text indexing. We first briefly state the results of this chapter. The exact formulation of all the problems considered are given in the following sections, but we now informally introduce them. We consider three concrete problems: Term Proximity in succinct space, Two-Pattern problems and Wild Card Indexing. In all the problems we have a collection of documents $\mathcal{D} = \{d_1, d_2, \ldots d_D\}$ with a total length of n characters. In the Term Proximity problem queries are a pattern P and a value k, and we must return k documents with the best Term Proximity score. We prove it is possible to create a Term Proximity data structure in succinct space with $\mathcal{O}(|P| + k \operatorname{polylog} n)$ query time. We establish upper and lower bounds for the Two Pattern problems, where queries are two patterns P_1 and P_2. There are a couple of variants: the 2P problem where documents matching both patterns must be returned, and the FP variant where documents matching P_1 but not P_2 must be returned. The presented data structure uses $\mathcal{O}(n)$ words of space, and has a query time of $\mathcal{O}(\sqrt{nk} \log^{1/2+\varepsilon} n)$ where k is the output size. For the Two-Pattern problems we show a lower bound trade-off between query time and space usage for the cases of reporting the documents and for counting the documents. Both of these lower bounds prove that known structures are essentially optimal (up to $n^{o(1)}$ factors). In the Wild Card Indexing problem, we show two lower bounds, that show near optimality of some of the known data structures. The two lower bounds work for different amounts of wild cards in the query pattern.

The rest of this chapter is structured as follows. In Section 4.1 we give a more thorough introduction to the text indexing area, and we continue in Section 4.2 with previous results on the problems we consider. In Section 4.3 we explain more carefully the results presented in this chapter. Before proving all the results we have some preliminaries in Section 4.4, that covers the various tools we use to obtain the upper and lower bounds. The first result we present is the Term Proximity in Section 4.5, afterwards, in Section 4.6, we

This chapter is based on the papers [3, 85, 93].

give our upper bounds for the 2P problem, which are easily adapted for the FP problem. We show how to reduce boolean matrix multiplication to the 2P problem in Section 4.7, which is also easily adapted for the FP problem. Next we move on to the lower bounds, first the pointer machine lower bounds for the Wild Card Indexing problem in Section 4.8, then in Section 4.9 we present the reporting lower bounds for the Two Pattern problems. We end the lower bound results in Section 4.10 with a lower bound for the counting variant of the Two Pattern problems in the semi-group model. It turns out, that the Two-Pattern problems are very related to the Set Intersection problem, and we get similar lower bounds this problem. Finally, we end in Section 4.11 with a discussion on the implications of the established lower bounds, and see that solutions obtained by easy applications of "standard" techniques solve the Two-Pattern problems almost optimally.

4.1 Introduction

Text indexing is a class of very important problems since it has many applications in very diverse fields ranging from biology to cyber security. Generally, the input is a single text or a collection of documents and the goal is to index them such that given a query P, all the documents where P occurs as a substring can be either found efficiently (the *reporting variant*), or counted efficiently (the *searching variant*).

The standard document retrieval problem where the query is just one text pattern was introduced in by Matias et al. [90]. The problem is classical and well-studied and there are linear space solutions with optimal query time [96]. Not surprisingly, there have been various natural extensions of this problem. As an example the counting case asks to find the number of documents containing the query pattern rather than its occurences. Many other and more interesting variations on this problem have also been studied. For an excellent survey on more results and extensions of document retrieval problems see [97].

Ranked document retrieval, that is, returning the documents that are most relevant to a query, is the fundamental task in Information Retrieval (IR) [12, 29]. Muthukrishnan [96] initiated the study of this family of problems in the general scenario where both the documents and the queries are general strings over arbitrary alphabets, which has applications in several areas [97]. In this scenario, we have a collection $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ of D string documents of total length n, drawn from an alphabet $\Sigma = [\sigma] = \{0, 1, \ldots, \sigma - 1\}$, and the query is a pattern $P[1..p]$ over Σ. Muthukrishnan considered a family of problems called *thresholded* document listing: given an additional parameter K, list only the documents where some function $\mathsf{score}(P, i)$ of the occurrences of P in d_i exceeded K. For example, the *document mining* problem aims to return the documents where P appears at least K times. Another variant, the

This chapter is based on the text indexing papers [3, 85, 93].

repeats problem, aims to return the documents where two occurrences of P appear at distance at most K. While document mining has obvious connections with typical term-frequency measures of relevance [12, 29] (i.e. when the number of occurences of a pattern determines the document's importance), the repeats problem is more connected to various problems in bioinformatics [17, 67]. Also notice that the repeats problem is closely related to the term proximity based document retrieval (i.e. when the two closest occurrences of the pattern determines the document's importance) in the Information Retrieval field [28, 113, 122–124]. Muthukrishnan achieved optimal time for both the document mining and repeats problems, with $\mathcal{O}(n)$ space (in words) if K is specified at indexing time and $\mathcal{O}(n \log n)$ if specified at query time. A more natural version of the thresholded problems, as used in IR, is *top-k retrieval*: Given P and k, return k documents with the best $\mathsf{score}(P, d)$ values.

A different variation on text indexing is to find all occurrences in a text that *approximately* match a pattern. This type of generalisation is very important in areas where there is a source of noise/error when receiving the bits. One example where approximate text matching is very useful is audio recognition, since the queries may have various sources of noise. To properly define *approximately match* there has to be a distance function between a position in the text and the query pattern. There are many examples of various distance functions and each requires a different data structure to solve the problem. The distance function could for instance be the *Hamming* distance, where the distance is the number of characters that mismatch. A different measure is the *edit distance*, where the distance between two texts is the number of insertions, deletions, and character substitutions required to turn one text into the other. Another way to *approximately* match is the "don't care" approach, where either the text, the query, or both contain "wild card" or "don't care" symbols. Now when receiving a query pattern, the task is to return all positions that match, and letting "wild cards" match any character. Each distance function finds applications in different areas and they are all important both from a theoretical and a practical stand point.

So far the described problems have dealt with exactly one pattern. A generalisation that turns out to have deep connections to problems in theoretical computer science are the *two-pattern query problems*. One of these types of problems was introduced in 2001 by Ferragina et al. [54] and since then it has attracted lots of attention. In the two-pattern problem, each query is composed of two patterns and a document matches the query if both patterns occur in the document (2P problem). One can also define the *Forbidden Pattern* problem (FP problem) [56] where the a document matches the query if it contains one pattern (the *positive* pattern) but not the other one (the *forbidden* or *negative* pattern). For symmetry, we also consider the *Two Forbidden Pattern* (2FP) problem where none of the patterns are allowed to match the document. For each problem one can also consider a *searching variant* or a *reporting variant*. Furthermore, in the searching variant, we can consider the

documents to be either weighted or unweighted; the weighted version is especially attractive in situations where the weight represents the "importance" of each document (e.g. the PageRank value of a document). We will work with the weighted sets where a document d_i is assigned a weight $w(d_i)$ from a semi-group G (more details and motivations to follow) and we call this the *semi-group variant*.

In studying the 2P problems we also delve into a problem that we call *set intersection queries* (SI). In this problem, the input is m sets, $S_1, \cdots, S_m$ of total size n, that are subsets of a universe $\mathcal{U}$. Each query is a pair of indices i and j. The reporting variant ask for all the elements in $S_i \cap S_j$. In the searching variant, the input also includes a weight function $w : \mathcal{U} \to G$ where G is a semi-group. The query asks for $\sum_{x \in S_i \cap S_j} w(x)$. Finally, in the *decision variant* we simply want to know whether $S_i \cap S_j = \emptyset$. The set intersection queries have appeared in many different formulations and variants (e.g., see [1, 45, 46, 83, 109, 110]). The most prominent conjecture related to set intersection is that answering the decision variant with constant query time requires $\Omega(n^{2-o(1)})$ space (see [109] for more details).

4.2 Previous Results

Ranked document retrieval Recall in the Ranked Document Retrieval problem we are given a set of documents $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ each over an alphabet $\Sigma = [\sigma] = \{0, 1, \ldots, \sigma - 1\}$ as input. The queries are pairs (P, k) where P is a pattern over Σ and k is an integer. The task is then, for a known scoring function $\mathsf{score}(P, i)$, to return k documents with the best score. The total length of the documents $\Sigma_{i=1}^{D} |d_i| = n$ and we are interested in solutions that use low space in terms of n.

Hon et al. [70, 75] gave a general framework to solve top-k problems for a wide variety of $\mathsf{score}(P, i)$ functions, which takes $\mathcal{O}(n)$ space, allows k to be specified at query time, and solves queries in $\mathcal{O}(p + k \log k)$ time. Navarro and Nekrich reduced the time to $\mathcal{O}(p + k)$ [99], and finally Shah et al. achieved time $\mathcal{O}(k)$ given the locus of P in the generalized suffix tree of $\mathcal{D}$ [114].

The problem is far from closed, however. Even the $\mathcal{O}(n)$ space (i.e., $\mathcal{O}(n \log n)$ bits) is excessive compared to the size of the text collection itself ($n \log \sigma$ bits), and in data-intensive scenarios it often renders all these solutions impractical by a wide margin. Hon et al. also introduced a general framework for *succinct indexes* [70], which use $o(n)$ bits[1] on top of a *compressed suffix array* (CSA) [98], which represents $\mathcal{D}$ in a way that also provides pattern-matching functionalities on it, all within space close to that of the *compressed* collection ($|\mathsf{CSA}|$ bits). A CSA finds the suffix array interval of $P[1..p]$ in time $t_{\mathsf{s}}(p)$ and retrieves any cell of the suffix array or its inverse

[1] If $D = o(n)$, which we assume for simplicity in this paper. Otherwise it is $D \log(n/D) + \mathcal{O}(D) + o(n) = \mathcal{O}(n)$ bits.

in time t_{SA}. Hon et al. achieved $\mathcal{O}(t_{\mathsf{s}}(p) + k\, t_{\mathsf{SA}} \log^{3+\varepsilon} n)$ query time, using $\mathcal{O}(n/\log^{\varepsilon} n)$ bits. Subsequent work (see [97]) improved the initial result up to $\mathcal{O}(t_{\mathsf{s}}(p) + k\, t_{\mathsf{SA}} \log^2 k \log^{\varepsilon} n)$ [101, 103], and also considered *compact* indexes, which may use up to $o(n \log n)$ bits on top of the CSA. For example, these achieve $\mathcal{O}(t_{\mathsf{s}}(p) + k\, t_{\mathsf{SA}} \log k \log^{\varepsilon} n)$ query time using $n \log \sigma + o(n)$ further bits [74], or $\mathcal{O}(t_{\mathsf{s}}(p) + k \log^* k)$ query time using $n \log D + o(n \log n)$ further bits [102, 103].

Two Pattern Problems We start by formally defining the text indexing problems we consider where the queries are two patterns.

2P Problem Given a set of strings $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ with $\sum_{i=1}^{D} |d_i| = n$, preprocess $\mathcal{D}$ to answer queries: given two strings P_1 and P_2 report all i's where both P_1 and P_2 occur in d_i.

FP Problem Given a set of strings $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ with $\sum_{i=1}^{D} |d_i| = n$, preprocess $\mathcal{D}$ to answer queries: given two strings P^+ and P^- report all i's where P^+ occurs in string d_i and P^- does not occur in string d_i.

2FP Problem Given a set of strings $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ with $\sum_{i=1}^{D} |d_i| = n$, preprocess $\mathcal{D}$ to answer queries: given two strings P_1 and P_2 report all i's neither of P_1 or P_2 occur.

2DSS Problem (shorthand for the *two dimensional substring problem*) Let $\mathcal{D} = \{(d_{1,1}, d_{1,2}), (d_{2,1}, d_{2,2}), \ldots, (d_{D,1}, d_{D,2})\}$, be a set of pairs of strings with $\sum_{i=1}^{D} |d_{i,1}| + |d_{i,2}| = n$. Preprocess $\mathcal{D}$ to answer queries: given two strings P_1 and P_2 report all i's where P_1 occurs in $d_{i,1}$ and P_2 occurs in $d_{i,2}$.

Muthukrishnan presented a data structure for the 2P problem using space $\mathcal{O}(n^{1.5} \log^{\mathcal{O}(1)} n)$ (in words) with $\mathcal{O}(p_1 + p_2 + \sqrt{n} + k)$ time for query processing, where $p_1 = |P_1|$ and $p_2 = |P_2|$ and k is the output size[1] [96]. Later Cohen and Porat [46] presented a space efficient structure of $\mathcal{O}(n \log n)$-space, but with a higher query time of $\mathcal{O}(p_1 + p_2 + \sqrt{nk \log n} \log^2 n)$. The space and the query time of was improved by Hon et al. [71] to $\mathcal{O}(n)$ words and $\mathcal{O}(p_1 + p_2 + \sqrt{nk \log n} \log n)$ time. See [76] for a succinct space solution for this problem as well as an improved linear space structure with query time $\mathcal{O}(p_1 + p_2 + \sqrt{nk \log n} \log \log n)$. The FP problem was introduced by Fischer et al. [56], where they presented an $\mathcal{O}(n^{3/2})$-bit solution with query time $\mathcal{O}(p_1 + p_2 + \sqrt{n} + k)$. Immediately, Hon et al. [73] improved its space occupancy to $\mathcal{O}(n)$ words, but with a higher query time of $\mathcal{O}(p_1 + p_2 + \sqrt{nk \log n} \log^2 n)$. They presented an $\mathcal{O}(n)$-space and $\mathcal{O}(p_1 + p_2 + \sqrt{n} \log \log n)$ query time structure for the counting version of FP (i.e., just report the value k). We remark that the same framework can be adapted to handle the counting version of

[1] Specifically k is the maximum of 1 and the output size.

2P as well. Also the $\mathcal{O}(\log\log n)$ factor in the query time can be removed by replacing predecessor search queries in their algorithm by range emptiness queries. In summary, we have $\mathcal{O}(n)$-space and $\tilde{\Omega}(\sqrt{n})$ query time solutions for the counting versions of these problems. Later we address the question of whether these are the best possible bounds.

The 2DSS problem was introduced by Ferragina et al. [54]. They presented a number of solutions for the 2DSS problem with space and query times that depend on the "average size" of each document. Their strategy was to reduce it to another problem known as the *common colors query* problem, where the task is to preprocess an array of colors and maintain a data structure, such that whenever two ranges comes as a query, we can output the unique colors which are common to both ranges. Based on their solution for this new problem, they presented an $\mathcal{O}(n^{2-\varepsilon})$ space and $\mathcal{O}(n^{\varepsilon}+k)$ query time solution for 2DSS, where ε is any constant in $(0,1]$. Later Cohen and Porat [46] presented a space efficient solution for the common colors query problem of space $\mathcal{O}(n\log n)$ words and query time $\mathcal{O}(\sqrt{nk\log n}\log^2 n)$. Therefore, the current best data structure for the 2DSS problem occupies $\mathcal{O}(n\log n)$ space and processes a query in $\mathcal{O}(p_1+p_2+\sqrt{nk\log n}\log^2 n)$ time.

The 2P and 2DSS Problems have been independently studied but we note that they are actually equivalent up to constant factors. Suppose we have a solution for 2DSS, and we are given the input for 2P, i.e. a set of strings $\mathcal{D}=\{d_1,d_2,\ldots,d_D\}$. Now build the data structure for Problem 2DSS with the input $\mathcal{D}'=\{(d_1,d_1),(d_2,d_2),\ldots,(d_D,d_D)\}$. The queries remain the same. This reduction has an overhead factor of two.

Similarly, suppose we have a solution for 2P and we are given the input for 2DSS, i.e. $\mathcal{D}=\{(d_{1,1},d_{1,2}),(d_{2,1},d_{2,2}),\ldots,(d_{D,1},d_{D,2})\}$. We make a new alphabet Σ' such that $|\Sigma'|=2|\Sigma|$. Now create the set of strings $\mathcal{D}'=\{d_1,d_2,\ldots,d_D\}$, where $d_i=d_{i,1}d'_{i,2}$ and $d'_{i,2}$ is $d_{i,2}$ where each character s is changed to $s+|\Sigma|$. A query is changed in the same manner: (P_1,P_2) is changed to (P_1,P'_2) where P'_2 is P_2 with each character s replaced by $s+|\Sigma|$. This reduction increases the input length by a factor of at most two (one extra bit per character).

The only lower bound so far says that with query time of $\mathcal{O}(\text{poly}(\log n)+k)$, the space must be $\Omega(n(\log n/\log\log n)^3)$. This is achieved by showing that one can solve 4-dimensional range queries using a two-pattern data structure [56]. However this bound is far away from the upper bounds.

There has been some work on hardness results, however. For the FP problem (reporting variant) Kopelowitz et al. [83] show that assuming there is no $\mathcal{O}(n^{2-o(1)})$ algorithm for the integer 3SUM problem then $P(n)+n^{\frac{1.5+\varepsilon}{2-\varepsilon}}Q(n)\geq n^{\frac{2}{2-\varepsilon}-o(1)}$ for any $0<\varepsilon<1/2$, where $P(n)$ is the preprocessing time and $Q(n)$ is the query time. For the best bound, ε should be set close to $1/2$. For the 2P problem (reporting variant), they show a slightly different trade-off curve of $P(n)+n^{\frac{1+\varepsilon}{2-\varepsilon}}Q(n)\geq n^{4/3-o(1)}$, for any $0<\varepsilon<1/2$ (the best bound is achieved

when ε is close to 0).

Unfortunately, the hardness results tell us nothing about the complexity of the space usage, $S(n)$, versus the query time which is what we are truly interested in for data structure problems. Furthermore, even under the relatively generous assumption[1] that $P(n) = \mathcal{O}(S(n)n^{o(1)})$ the space and query lower bounds obtained from the above results have polynomial gaps compared with the current best data structures.

Set Intersection The interest in set intersection problems has grown considerably in recent years and variants of the set intersection problem have appeared in many different contexts. Cohen and Porat [46] considered the reporting variant of the set intersection queries (due to connection to 2P problem) and presented a data structure that uses linear space and answers queries in $\mathcal{O}(\sqrt{nk})$ time, where k is the output size. They also presented a linear-space data structure for the searching variant that answers queries in $\mathcal{O}(\sqrt{n})$ time. In [80] the authors study set intersection queries because of connections to dynamic graph connectivity problems. They offer very similar bounds to those offered by Cohen and Porat (with a $\sqrt{\log n}$ factor worse space and query times) but they allow updates in $\mathcal{O}(\sqrt{n \log n})$ time. As far as we know, there has not been further progress on building better data structures and in fact, it is commonly believed that all set intersection queries are hard. Explicitly stated conjectures on set intersection problems are used to obtain conditional lower bounds for diverse problems such as distance oracles in graphs [45, 109, 110]. On the other hand, other well-known problems, such as 3SUM, can be used to show conditional lower bounds for variants of set intersection problems [1, 83]. For a few other variants of set intersection problems see [18, 50, 107].

Dietz et al. [50] considered set intersection queries in the semi-group model (a.k.a. *the arithmetic model*) and they presented near optimal dynamic and offline lower bounds. They proved that given a sequence of n updates and q queries one must spend $\Omega(q + n\sqrt{q})$ time (ignoring polylog factors); in the offline version a sequence of n insertions and q queries are used but in the dynamic version, the lower bound applies to a dynamic data structure that allows insertion and deletion of points, as well as set intersection queries.

Wild Card Indexing Cole et al. [47] presented a solution for indexing a text and querying with patterns containing up to κ wild cards. Their data structure uses $\mathcal{O}(n\frac{\log^\kappa n}{\kappa!})$ words of space and answers queries in $\mathcal{O}(P + 2^\kappa \log\log n + t)$, where t is the number of occurrences. Recently this structure was generalized to provide a trade-off with increased query time $\mathcal{O}(P + \beta^j \log\log n + t)$ and reduced space usage $\mathcal{O}(n \log n \log_\beta^{\kappa-1} n)$ for any $2 \leq \beta \leq \sigma$ (σ is the alphabet size), where j is the number of wild cards in the query [19].

[1] There are problems, such as jumbled indexing [33], where the preprocessing time is a polynomial factor larger than the space complexity.

In the same paper an index with $\mathcal{O}(m+t)$ query time and $\mathcal{O}(\sigma^{\kappa^2} n \log^{\kappa} n \log n)$ space usage was also presented. Another result, which is not easily comparable, is an $\mathcal{O}(n)$ space index with query time $\mathcal{O}(P + \alpha)$ [111]. Here α is the number of occurrences of all the subpatterns separated by wild card characters. This can obviously have worst case linear query time, but it could be a more practical approach.

There have been no lower bounds for Wild Card Indexing (WCI) queries. However, the *partial match* problem is a closely related problem for which there are many cell-probe lower bounds. One of the main differences is that there are no hard limits on the number of wild cards in the partial match problems; because of this, the partial match problem has tight connections to higher dimensional nearest neighbour search type problems. In [108] it is proven that a data structure with query time q must use $\Omega(2^{m/q})$ words of space, under the assumption that the word size is $\mathcal{O}(n^{1-\varepsilon}/q)$, where m is the size of the query. In some sense this is the highest lower bound we can hope to prove with current techniques in the cell probe model. For a discussion of the cell probe complexity of the partial match problem we refer to [108].

For a summary of the results regarding document indexing for two patterns and wild cards see Table 4.2. The table also includes the new results we present in the later sections.

Table 4.1: (top) Data structures for the two-pattern (2P), the forbidden pattern (FP), and the set intersection (SI) query problems. (bottom) Lower bounds. Here, $M(m)$ is the time to multiply two $m \times m$ matrices. "Comb-MM" stands for the assumption that there is no "combinatorial" algorithm that multiplies two $m \times m$ matrices in time $\mathcal{O}(n^{3-o(1)})$. I3S refers to the assumption that there is no $\mathcal{O}(n^{2-o(1)})$ algorithm that solves an instance of the 3SUM problem with integer input. "PM" refers to the pointer machine model and "SG" refers to the semi-group model.

Problem	**Space**	**Query Time**	**Notes**
2P (reporting)	$n^{2-\alpha}\log^{\mathcal{O}(1)} n$	$n^{\alpha}+t$	Ferragina et al.'03 [54]
2P (reporting)	$n\log n$	$\sqrt{nt\log n}\log^2 n+t$	Cohen, Porat'10 [46]
2P (reporting)	n	$\sqrt{nt\log n}\log n+t$	Hon et al.'10 [71]
FP (reporting)	$n^{3/2}/\log n$	$\sqrt{n}+t$	Fischer et al.'12 [56]
FP (reporting)	n	$\sqrt{nt\log n}\log^2 n+t$	Hon et al.'12 [73]
2P, FP (reporting)	n	$\sqrt{nt\log n}\log^{\varepsilon} n+t$	**new**
2P, FP (counting)	n	$\sqrt{n}$	**new** (Observation)
2P, FP, SI (reporting)	$\frac{n^2\log^2 n}{Q(n)}$	$Q(n)+t$	**new** ($Q(n)=\Omega(\log n)$)
2P, FP, SI (counting)	$\frac{n^2\log^2 n}{Q^2(n)}$	$Q(n)$	**new** ($Q(n)=\Omega(\log n)$)
WCI	$n\frac{\log^{\kappa} n}{\kappa!}$	$2^{\kappa}\log\log n+t$	Cole et al. '04 [47]
WCI	$n\log n\log_{\beta}^{\kappa-1} n$	$\beta^{\kappa}\log\log n+t$	$2\le\beta\le\sigma$, Bille et al. '04 [19]
WCI	n	$\sum_{i=0}^{\kappa} occ(p_i)+t$	Rahman et al. '07 [111]

Problem	**Lower Bound**	**Notes**
2P, FP (counting)	$P(n)+nQ(n)=\Omega(M(\sqrt{n\log n}))$	**new**
2P, FP (counting)	$P(n)+nQ(n)=\Omega(n^{1.5-o(1)})$	[Comb-MM], **new**
2P (reporting)	$P(n)+n^{1+\varepsilon}Q(n)=\Omega(n^{4/3-o(1)})$	[I3S], [83]
FP (reporting)	$P(n)+n^{3/4+\varepsilon}Q(n)=\Omega(n^{4/3-\varepsilon})$	[I3S], [83]
2P, FP, SI (reporting)	$S(n)Q(n)=\Omega\left(n^{2-o(1)}\right)$	[PM], **new**
2P, FP, SI (reporting)	$Q(n)=(nk)^{\frac{1}{2}-\alpha}$, $S(n)=\Omega\left(n^{\frac{1+6\alpha}{1+2\alpha}-o(1)}\right)$	[PM], **new**
2P, FP, SI (counting)	$S(n)Q^2(n)=\Omega\left(n^{2-o(1)}\right)$	[SG], **new**
WCI (reporting)	$Q(n,\kappa)=\mathcal{O}(2^{\frac{\kappa}{2}})\Rightarrow$ $S(n,m,\kappa)=\Omega\left(n2^{\Theta(k)}n^{\Theta(\frac{1}{\log\kappa})}\right)$	$\kappa\ge 3\sqrt{\log n}$, **new**
WCI (reporting)	$Q(n)=f(n)+\mathcal{O}(m+t)$, $S(n)=\Omega\left(\frac{n}{\kappa}\Theta\left(\frac{\log_{Q(n)} n}{\kappa}\right)^{\kappa-1}\right)$	$\kappa\le\log_{Q(n)} n$, **new**

4.3 Our Results

Ranked Document Retrieval All the frameworks that give succinct and compact indexes discovered so far work *exclusively* for the term frequency (or closely related, e.g., tf-idf) measure of relevance. For the simpler case where documents have a fixed relevance independent of P, succinct indexes achieve $\mathcal{O}(t_{\mathsf{s}}(p)+k\,t_{\mathsf{SA}}\log k\log^{\varepsilon} n)$ query time [15], and compact indexes using $n\log D+o(n\log D)$ bits achieve $\mathcal{O}(t_{\mathsf{s}}(p)+k\log(D/k))$ time [62]. On the other hand, there have been *no succinct nor compact indexes for the term proximity measure of relevance.* The term proximity measure of relevance is defined as follows:

$$\mathsf{tp}(P,\ell)=\min\{\{|i-j|\mid i\neq j\wedge d_\ell[i..i+|P|-1]=d_\ell[j..j+|P|-1]=P\}\cup\{\infty\}\}$$

Where d_ℓ is the ℓ-th document and P is the pattern. We present the first succinct and compact indexes for term proximity. Theorem 7 gives a succinct structure that is competitive with the original succinct term-frequency results [70]. For example, on a recent CSA [14], this time is $\mathcal{O}(p+(k\log k+\log n)\log^{3+\varepsilon} n)$, whereas the original succinct term-frequency solution [70] would require $\mathcal{O}(p+k\log^{4+\varepsilon} n)$ time.

Theorem 7. *Using a* CSA *plus $o(n)$ bits data structure, one can answer top-k term proximity queries in $\mathcal{O}(t_{\mathsf{s}}(p)+(\log^2 n+k(t_{\mathsf{SA}}+\log k\log n))\log^{2+\varepsilon} n)$ time, for any constant $\varepsilon>0$.*

We further show how to extend our result to a scenario where $\mathsf{score}(\cdot,\cdot)$ is a weighted sum of document ranking, term-frequency, and term-proximity with predefined non-negative weights [123].

Two Pattern Problems We present an improved upper bound for the common colors query problem, where the space and query time are $\mathcal{O}(n)$ and $\mathcal{O}(\sqrt{nk}\log^{1/2+\varepsilon} n)$ respectively, where $\varepsilon>0$ is any constant. Therefore, we now have a linear-space and $\mathcal{O}(p_1+p_2+\sqrt{nk}\log^{1/2+\varepsilon} n)$ query time index for the 2DSS problem. Due to the reductions given previously this also gives the same upper bounds for the 2P problem.

The difficulty of obtaining fast data structures using (near) linear space has led many to believe that very efficient solutions are impossible to obtain. We strengthen these beliefs by presenting strong connections between the counting versions of the 2P problem and the boolean matrix multiplication problem. Specifically, we show that multiplying two $\sqrt{n}\times\sqrt{n}$ boolean matrices can be reduced to the problem of indexing $\mathcal{D}$ and answering n counting queries. However, matrix multiplication is a well known hard problem and this connection gives us a hardness result for the pattern matching problems under considerations. We also note that the hardness result also follows for FP since the reduction are easily adapted.

Letting ω be the best exponent for multiplying two $n \times n$ matrices, we show that $P(n)+nQ(n) = \tilde{\Omega}(n^{\omega/2})$ where $P(n)$ and $Q(n)$ are the preprocessing and the query times of the data structure, respectively. Currently $\omega = 2.3728639$, i.e., the fastest matrix multiplcation algorithm runs in $\mathcal{O}(n^{2.3728639})$ time. If one assumes that there is no "combinatorial" matrix multiplication algorithm with better running time than $\mathcal{O}(n^{3-o(1)})$, then the lower bound becomes $P(n) + nQ(n) = \Omega(n^{1.5-o(1)})$. However boolean matrix multiplication can be solved using a non-combinatorial algorithm (such as Strassen's algorithm). Expanding slightly on the result we see that either the preprocessing time must be high or the query time is high. Essentially we get that either preprocessing takes time $\tilde{\Omega}(n^{1.18})$ or a query takes $\tilde{\Omega}(n^{0.18})$ time.

The hardness results obtained by Kopelowitz et al [83] are independent of the bounds obtained here, and they will hold as long as integer 3SUM is hard regardless of the difficulty of the boolean matrix multiplication.

We also move away from hardness results and prove lower bounds directly. The lower bounds we prove are based on the Pointer Machine model and Semi-Group model (both are described in the preliminaries, Section 4.4). For the two pattern problems (and Set Intersection) our lower bounds show that all the known data structures are optimal within $n^{o(1)}$ factors:

Theorem (15)**.** *In the pointer machine model of computation, any data structure that solves the reporting variant of the aforementioned problems with $S(n)$ space and $Q(n) + \mathcal{O}(k)$ query time must have*

$$S(n)Q(n) = \Omega\left(n^{2-o(1)}\right)$$

As a corollary, we also obtain that there is no data structure that uses near linear space and has $\mathcal{O}((nk)^{1/2-\varepsilon} + k)$ query time, for any constant $\varepsilon > 0$. On the other hand, in the semi-group model of computation (see [42] or Section 4.4 for a description of the semi-group model), we prove that any data structure that solves the searching variant of the aforementioned problems with $S(n)$ space and $Q(n)$ query time must have $S(n)Q^2(n) = \Omega(n^2/\log^4 n)$. Furthermore, we show that both of the these trade off curves are almost optimal by providing data structures with space and query time trade-off that match our lower bounds, up to $n^{o(1)}$ factors. We claim no novelty in the upper bounds, and the data structures mostly follow from standard techniques.

Our results have a few surprising consequences. First, we can establish that the reporting case of these problems is strictly more difficult than their searching variant. For example, ignoring $n^{o(1)}$ factors, we can count the number of matching documents using $S_c = \mathcal{O}(n)$ space and with $Q_c = \mathcal{O}(\sqrt{n})$ query time but reporting them in $Q_r = \mathcal{O}(\sqrt{n}+k)$ time requires $S_r = \Omega(n^{3/2})$ space. Our lower bounds show that both of these results are the best possible and thus counting is certainly easier than reporting. Notice that conditional lower bounds that use reductions from offline problems (such as matrix multiplications or 3SUM) cannot distinguish between reporting and counting. For

example, consider the boolean matrix multiplication framework, and for the best outcome, let us add one assumption that $P(n) = S(n)n^{o(1)}$ and another assumption that no efficient "combinatorial" matrix multiplication exists; this will yield that $S_c + nQ_c = \Omega(n^{1.5-o(1)})$ and $S_r + nQ_r = \Omega(n^{1.5-o(1)})$ which both are almost tight bounds (i.e., we cannot make the exponent "1.5" any larger) but they are unable to tell us that for $Q_r = \mathcal{O}(\sqrt{n})$ we must also have $S_r = \Omega(n^{1.5})$. The second surprising outcome is that separation between counting and reporting is a rare phenomenon with often counting being the difficult variant. The fact that here we observe the reverse is quite interesting. Third, we provably establish that getting fast queries always requires close to n^2 space.

Remarks. Proving a lower bound for the decision version of the problems mentioned above is considered a major open problem that needs breakthrough techniques. While we believe our lower bounds are certainly interesting (particularly since they separate counting and reporting variants), they do not make any progress towards resolving this major open problem.

Wild Card Indexing For WCI with κ wild cards, we prove two results, both in the pointer machine model. First, we prove that for $\kappa \geq 3\sqrt{\log n}$ and for a binary alphabet ($\sigma = 2$), if we want fast queries (specifically, if we want query time smaller than $\mathcal{O}(2^{k/2} + m + t)$, where m and t are the pattern length and the output size respectively), then the data structure must consume $\Omega(n2^{\Theta(k)}n^{\Theta(1/\log k)})$ space; For $\kappa = 3\sqrt{\log n}$, this space bound can be written as $\Omega(n2^{\Theta(k^2/\log k)})$ which is very close to a structure offered by Bille et al. [19]. Second, we prove that any pointer-machine data structure that answers WCI queries in time $Q(n) + \mathcal{O}(m + t)$ must use $\Omega\left(\frac{n}{\kappa}\Theta\left(\frac{\log_{Q(n)} n}{\kappa}\right)^{\kappa-1}\right)$ space, as long as $\kappa < \log_{Q(n)} n$. Combined with our first lower bound, these show that all known WCI data structures are almost optimal, at least for a particular range of parameters (for more details, see the discussion in Section 4.8.3).

4.4 Preliminaries

In this section we introduce concepts as well as reiterate previous results that we use as black boxes or building blocks. For readers that are familiar with the standard succinct data structure techniques and standard string indexes this section merely serves as a refresher and a description of notation.

Suffix Trees. The suffix tree [119] of a string T is a compact trie containing all of its suffixes, where the ith leftmost leaf, ℓ_i, represents the ith lexicographically smallest suffix. It is also called the generalized suffix tree of $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$, GST. Each edge in GST is labeled by a string, and

path(x) is the concatenation of the edge labels along the path from the GST root to the node x. Then path(ℓ_i) is the ith lexicographically smallest suffix of T. The highest node x with path(x) prefixed by $P[1..p]$ is the *locus* of P, and is found in time $\mathcal{O}(p)$ from the GST root. The GST uses $\mathcal{O}(n)$ words of space.

Suffix Arrays. The suffix array ([88]) of T, SA$[1..n]$, is defined as SA$[i] = n+1-|\,\text{path}(\ell_i)|$, the starting position in T of the ith lexicographically smallest suffix of T. The *suffix range* of P is the range SA$[sp, ep]$ pointing to the suffixes that start with P, T[SA$[i]$..SA$[i] + p - 1] = P$ for all $i \in [sp, ep]$. Also, ℓ_{sp} (resp., ℓ_{ep}) are the leftmost (resp., rightmost) leaf in the subtree of the locus of P.

Compressed Suffix Arrays. The compressed suffix array [98] of T, CSA, is a compressed representation of SA, and usually also of T. Its size in bits, |CSA|, is $\mathcal{O}(n \log \sigma)$ and usually much less. The CSA finds the interval $[sp, ep]$ of P in time $t_{\mathsf{s}}(p)$. It can output any value SA$[i]$, and even of its inverse permutation, SA$^{-1}[i]$, in time t_{SA}. For example, a CSA using $nH_h(\mathsf{T}) + o(n \log \sigma)$ bits [14] gives $t_{\mathsf{s}}(p) = \mathcal{O}(p)$ and $t_{\mathsf{SA}} = \mathcal{O}(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$, where H_h is the hth order empirical entropy [89].

Compressed Suffix Trees. The compressed suffix tree of T, CST, is a compressed representation of GST, where node identifiers are their corresponding suffix array ranges. The CST can be implemented using $o(n)$ bits on top of a CSA [100] and compute (among others) the lowest common ancestor (LCA) of two leaves ℓ_i and ℓ_j, in time $\mathcal{O}(t_{\mathsf{SA}} \log^{\epsilon} n)$, and the Weiner link Wlink(a, v), which leads to the node with path label $a \circ \text{path}(v)$, in time $\mathcal{O}(t_{\mathsf{SA}})$[1].

Rank/Select The Rank/Select problem is the following. Given a set $S \subseteq \{0, 1, 2, \ldots, n-1\}$ of size $|S| = m$ support the following operations:

1. Rank(x): Returns the number of elements in S that are less than x and -1 if $x \notin S$.

2. Select(i): Return the ith smallest element in S.

The set S can be represented as a bit array of length n where the ith bit is 1 if $i \in S$. That is, there are m ones in the bit string that represents S. In this setting the Rank operation corresponds to counting the number of 1s that precede a particular 1 in the bit string. Similarly the Select operation returns the position of the ith 1.

[1] Using $\mathcal{O}(n/\log^{\epsilon} n)$ bits and no special implementation for operations SA^{-1}[SA$[i] \pm 1$].

Rank/Select structures have many applications. One of the first applications was to use it for storing a static tree of arbitrary degree as efficiently as possible while also being able to traverse it [77].

The particular Rank/Select structure that we use is summarized in the following lemma [112, Remark 4.2].

Lemma 8. *There is a Rank/Select structure using $n + \mathcal{O}(n \log\log n / \log n)$ bits of space that allows for Rank and Select queries in $\mathcal{O}(1)$ time on S and the complement of S.*

Documents Indexing Let $\mathsf{T}[1..n] = d_1 \circ d_2 \circ \cdots d_D$ be the text (from an alphabet $\Sigma = [\sigma] \cup \{\$\}$) obtained by concatenating all the documents in $\mathcal{D}$. Each document is terminated with a special symbol \$, which does not appear anywhere else. A suffix $\mathsf{T}[i..n]$ of T belongs to d_j iff i is in the region corresponding to d_j in T. Thus, it holds $j = 1 + rank_B(i-1)$, where $B[1..n]$ is a bitmap defined as $B[l] = 1$ iff $\mathsf{T}[l] = \$$ and $rank_B(i-1)$ is the number of 1s in $B[1..i-1]$. This operation is computed in $\mathcal{O}(1)$ time on a representation of B that uses $D\log(n/D) + \mathcal{O}(D) + o(n)$ bits [112]. For simplicity, we assume $D = o(n)$, and thus B uses $o(n)$ bits.

Two Dimensional Range Maximum Queries

The Range Maximum Query problem in one dimension is the following. Given a static array $A[1..n]$ of n integers support the following operation: Given a range $[a, b]$ return the index of the maximum value in $A[a..b]$. There are several variations on this problem, sometimes it is not the index that we are interested in but rather the value. Note that this will actually influence the space requirements and running time of the queries!

The two dimensional Range Maximum Query problem is the same as before but extended to matrices instead of arrays. In this two dimensional case we are given an $n \times m$ matrix $A[1..n, 1..m]$ and the queries are two dimensional as well. That is, the answer to a query $[a..b] \times [c..d]$ is the index of the maximum value in the submatrix $A[a..b, c..d]$.

The bounds that we use for the Two Dimensional Range Maximum Query problem are given in the following lemma [24, Theorem 3].

Lemma 9. *The 2-Dimensional Range Maximum Query problem for an m by n matrix of size $N = n \cdot m$ is solved in $\mathcal{O}(N)$ preprocessing time and $\mathcal{O}(1)$ query time using $\mathcal{O}(N)$ bits additional space*

This result is in the *indexing model* where access to the original input matrix is allowed.

Range Emptiness Queries

It is often the case that we need to perform some kind of predecessor/successor query on an element. Sometimes, however, we only need to know *if* there is an element in the immediate viscinity and in this case we use the range emptiness structure. The One Dimensional Range Emptiness problem is the following. Store a set S of n integers while supporting the query: Given a, b determine if there is an $x \in S$ such that $a \leq x \leq b$. In other words, determine if S has any values in the interval $[a, b]$, i.e. is $S \cap [a, b]$ the empty set?

The One Dimensional Range Emptiness structure that we use is summarized in the following lemma [9, Theorem 4].

Lemma 10. *Given a set of n integers S, there is a data structure using $\mathcal{O}(n)$ words of space that supports Range-Emptiness queries on S in $\mathcal{O}(1)$ time.*

Orthogonal Range Successor/Predecessor Queries

The Range Successor/Predecessor problem is also known as range next-value problem. In this settings we are given a set S of two dimensional points to preprocess and to answer queries where we are given a range $Q = [a, \infty] \times [c, d]$ and must report the point in $S \cap Q$ with smallest first coordinate. This problem is studied in [104] and they achieve the following

Lemma 11. *The Range Successor (Predecessor) problem can be solved with $\mathcal{O}(n)$ space and $O(\log^{\varepsilon} n)$ query time where n is the number of points stored.*

Combining with a standard range tree partitioning [49], the following result easily follows.

Lemma 12. *Given n' points in $[n] \times [n] \times [n]$, a structure using $\mathcal{O}(n' \log^2 n)$ bits can support the following query in $\mathcal{O}(\log^{1+\epsilon} n)$ time, for any constant $\epsilon > 0$: find the point in a region $[x, x'] \times [y, y'] \times [z, z']$ with the lowest/highest x-coordinate.*

The Pointer Machine Model [115] This models data structures that solely use pointers to access memory locations[1] (e.g., any tree-based data structure) We focus on a variant that is the popular choice when proving lower bounds [41]. Consider an abstract "reporting" problem where the input includes a universe set $\mathcal{U}$ where each query q reports a subset, $q_{\mathcal{U}}$, of $\mathcal{U}$. The data structure is modelled as a directed graph G with outdegree two (and a root note $r(G)$) where each vertex represents one memory cell and each memory cell can store one element of $\mathcal{U}$; edges between vertices represent

[1] All known solutions for the 2P query problem use tree structures, such as suffix trees or wavelet trees. While sometimes trees are can be encoded using bit-vectors with rank/select structures on top, these tricks can only save polylogarithmic factors in space and query times.

pointers between the memory cells. All other information can be stored and accessed for free by the data structure. The only requirement is that given the query q, the data structure must start at $r(G)$ and explore a connected subgraph of G and find its way to vertices of G that store the elements of $q_{\mathcal{U}}$. The size of the subgraph explored is a lower bound on the query time.

An important remark The pointer-machine can be used to prove lower bounds for data structures with query time $Q(n) + \mathcal{O}(t)$ where t is the output size and $Q(n)$ is "search overhead". Since we can simulate any RAM algorithm on a pointer-machine with $\log n$ factor slow down, we cannot hope to get high lower bounds if we assume the query time is $Q(n) + \mathcal{O}(t \log n)$, since that would automatically imply RAM lower bounds for data structures with $Q(n)/\log n + \mathcal{O}(t)$ query time, something that is hopelessly impossible with current techniques. However, when restricted to query times of $Q(n) + \mathcal{O}(t)$, the pointer-machine model is an attractive choice and it has an impressive track record of proving lower bounds that match the best known data structures up to very small factors, even when compared to RAM data structures; we mention two prominent examples here. For the fundamental *simplex range reporting* problem, all known solutions are pointer-machine data structures and the most efficient solutions that use $S(n)$ space, and have the query time $Q(n)+\mathcal{O}(t)$ we have $S(n) = \mathcal{O}((n/Q(n))^{d+o(1)})$ [32, 44, 91] on the other hand, known lower bounds almost match these bounds [2, 43]. For the other fundamental *orthogonal range reporting*, the best RAM algorithms save a $o(\log n)$ factor in space (by "word-packing" techniques) and another $o(\log n)$ factor in the query time (by using van Emde Boas type approaches in low dimensions). In the pointer-machine model, Chazelle [41] provided tight space lower bounds and later very high query lower bounds were discovered [4, 5] (the last paper contains a tight query lower bound for the *rectangle stabbing problem*).

The Semi-Group Model In this model, each element of the input set $\mathcal{U} = \{x_1, \ldots, x_n\}$ is assigned a weight, by a weight function $\mathbf{w} : \mathcal{U} \to G$ where G is a semi-group. Given a query q, let $q_{\mathcal{U}}$ be the subset of $\mathcal{U}$ that matches q. The output of the query is $\sum_{x \in q_{\mathcal{U}}} \mathbf{w}(x)$. By restricting G to be a semi-group, the model can capture very diverse set of queries. For example, finding the document with maximum weight that matches a query can be modelled using the semi-group $\mathbb{R}$ with "max" operation. The lower bound variant of the semi-group model was first introduced by Chazelle [42] in 1990. In this variant, it is assumed that the data structure stores "precomputed" sums $s_1, \ldots, s_m$ where each $s_i = \sum_{j \in I_i} \alpha_{i,j} \mathbf{w}(x_j)$, $\alpha_{i,j} \in \mathbb{N}$, and I_i is some subset of $[n]$ and thus m is a lower bound on space cost of the data structure. While the integers α_{ij} can depend on the weight function $\mathbf{w}$, the set I_i must be independent of $\mathbf{w}$ (they both can depend on $\mathcal{U}$). Finally, the semi-group is required to be *faithful*

Observe that there is no inverse operation for max.

which essentially means that if $\sum_{i \in I} \alpha_i g_i = \sum_{i \in I'} \alpha'_i g_i$ for every assignment of semi-group values to variables g_i, then we must have $I = I'$ and $\alpha_i = \alpha'_i$. To answer a query, the data structure can pick t precomputed values to create the sum $\sum_{x \in q_{\mathcal{U}}} \mathbf{w}(x)$ and thus t is a lower bound on the query time. For a detailed description and justification of the model we refer the reader to Chazelle's paper [42]. Similar to the pointer machine model, this semi-group model is often used to prove lower bounds that closely match the bounds of the existing data structures (see e.g., [11, 40, 42]).

4.5 Term Proximity

In this section we present our results for ranked document retrieval. The primary result is that we can compute the *top-k* documents when the scoring function is the proximity measure. We quickly restate the problem definition and previous result, then move on to an overview of the data structure. Afterwards follows more detailed descriptions of the various parts of the data structure. Then we give an analysis and finally we extend the result for the case where the scoring function is a weighted sum of term frequency, term proximity, and static weights (PageRank).

Let $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ be a collection of D string documents of n characters in total, that are drawn from an alphabet set $\Sigma = [\sigma]$. The *top-k document retrieval problem* is to preprocess $\mathcal{D}$ into a data structure that, given a query $(P[1..p], k)$, can return the k documents of $\mathcal{D}$ most relevant to the pattern P. The relevance is captured using a the predefined ranking function, which depends on the set of occurrences of P in d_i. As mentioned we study term proximity (i.e., the distance between the closest pair of occurrences of P in d_i). Linear space and optimal query time solutions already exist for the standard top-k document retrieval problem (PageRank and term frequency). Compressed and compact space solutions are also known, but only for a few ranking functions such as term frequency and importance. However, space efficient data structures for term proximity based retrieval have been evasive. In this section we present the first sub-linear space data structure for term-frequency, which uses only $o(n)$ bits on top of any compressed suffix array of $\mathcal{D}$ and solves queries in $\mathcal{O}((p+k)\operatorname{polylog} n)$ time. We also show that scores that consist of a weighted combination of term proximity, term frequency, and document importance, can be handled using twice the space required to represent the text collection.

4.5.1 An Overview of our Data Structure

The top-k term proximity is related to a problem called *range restricted searching*, where one must report all the occurrences of P that are within a text range $\mathsf{T}[i..j]$. It is known that succinct data structures for that problem are unlikely to exist in general, whereas indexes of size $|\mathsf{CSA}| + \mathcal{O}(n/\log^{\epsilon} n)$ bits do exist for patterns longer than $\Delta = \log^{2+\epsilon} n$ [72]. Therefore, our basic strategy will be to have a separate data structure to solve queries of length $p = \pi$, for each $\pi \in \{1, \ldots, \Delta\}$. Patterns with length $p > \Delta$ can be handled with a single succinct data structure. More precisely, we design two different data structures that operate on top of a CSA:

- An $\mathcal{O}(n \log\log n/(\pi \log^{\gamma} n))$-bits structure for handling queries of fixed length $p = \pi$, in time $\mathcal{O}(t_{\mathsf{s}}(p) + k(t_{\mathsf{SA}} + \log\log n + \log k)\,\pi \log^{\gamma} n)$. This is described in Section 4.5.2 and the result is summarized in Lemma 14.

- An $\mathcal{O}(n/\log^{\epsilon} n + (n/\Delta)\log^2 n)$-bits structure for handling queries with $p > \Delta$ in time $\mathcal{O}(t_{\mathsf{s}}(p) + \Delta(\Delta + t_{\mathsf{SA}}) + k \log k \log^{2\epsilon} n(t_{\mathsf{SA}} + \Delta \log^{1+\epsilon} n))$. This is described in Section 4.5.3 and the result is summarized in Lemma 16.

By building the first structure for every $\pi \in \{1, \ldots, \Delta\}$, any query can be handled using the appropriate structure. The Δ structures for fixed pattern length add up to $\mathcal{O}(n(\log\log n)^2/\log^{\gamma} n) = o(n/\log^{\gamma/2} n)$ bits, whereas that for long patterns uses $\mathcal{O}(n/\log^{\epsilon} n)$ bits. By choosing $\varepsilon = 4\epsilon = 2\gamma$, the space is $\mathcal{O}(n/\log^{\varepsilon/4} n)$ bits. As for the time, the structures for fixed $p = \pi$ are most costly for $\pi = \Delta$, where the time is

$$\mathcal{O}(t_{\mathsf{s}}(p) + k(t_{\mathsf{SA}} + \log\log n + \log k)\,\Delta \log^{\gamma} n).$$

Adding up the time of the second structure, we get

$$\mathcal{O}(t_{\mathsf{s}}(p) + \Delta(\Delta + k(t_{\mathsf{SA}} + \log k \log^{1+\epsilon} n)\log^{2\epsilon} n)),$$

which is upper bounded by

$$\mathcal{O}(t_{\mathsf{s}}(p) + (\log^2 n + k(t_{\mathsf{SA}} + \log k \log n))\log^{2+\varepsilon} n).$$

This yields Theorem 7.

Now we introduce some formalization to convey the key intuition. The term proximity $\mathsf{tp}(P, i)$ can be determined by just two occurrences of P in d_i, which are the closest up to ties. We call them *critical occurrences*, and a pair of two closest occurrences is a *critical pair*. Note that one document can have multiple critical pairs.

Definition 3. *An integer $i \in [1, n]$ is an* occurrence *(or* match*) of P in d_j if the suffix $\mathsf{T}[i..n]$ belongs to d_j and $\mathsf{T}[i..i + p - 1] = P[1..p]$. The set of all occurrences of P in T is denoted $\mathsf{Occ}(P)$.*

Definition 4. *An occurrence m_i of P in d_i is a* critical occurrence *if there exists another occurrence m_i' of P in d_i such that $|m_i' - m_i| = \mathsf{tp}(P, i)$. The pair (m_i, m_i') is called a* critical pair *of d_i with respect to P.*

A key concept in our solution is that of *candidate sets* of occurrences, which contain sufficient information to solve the top-k query (note that, due to ties, a top-k query may have multiple valid answers).

Definition 5. *Let $\mathsf{Topk}(P, k)$ be a valid answer for the top-k query (P, k). A set $\mathsf{Cand}(P, k) \subseteq \mathsf{Occ}(P)$ is a* candidate set *of $\mathsf{Topk}(P, k)$ if, for each document identifier $i \in \mathsf{Topk}(P, k)$, there exists a critical pair (m_i, m_i') of d_i with respect to P such that $m_i, m_i' \in \mathsf{Cand}(P, k)$.*

Lemma 13. *Given a CSA on $\mathcal{D}$, a valid answer to query (P, k) can be computed from $\mathsf{Cand}(P, k)$ in $\mathcal{O}(z \log z)$ time, where $z = |\mathsf{Cand}(P, k)|$.*

Proof. Sort the set $\mathsf{Cand}(P,k)$ and traverse it sequentially. From the occurrences within each document d_i, retain the closest consecutive pair (m_i, m_i'), and finally report k documents with minimum values $|m_i - m_i'|$. This takes $\mathcal{O}(z \log z)$ time.

We show that this returns a valid answer set. Since $\mathsf{Cand}(P,k)$ is a candidate set, it contains a critical pair (m_i, m_i') for each $i \in \mathsf{Topk}(P,k)$, so this critical pair (or another with the same $|m_i - m_i'|$ value) is chosen for each $i \in \mathsf{Topk}(P,k)$. If the algorithm returns an answer other than $\mathsf{Topk}(P,k)$, it is because some document $d \in \mathsf{Topk}(P,k)$ is replaced by another $i' \notin \mathsf{Topk}(P,k)$ with the same score $\mathsf{tp}(P,i') = |m_{i'} - m_{i'}'| = |m_i - m_i'| = \mathsf{tp}(i)$. □

Our data structures aim to return a small candidate set (as close to size k as possible), from which a valid answer is efficiently computed using Lemma 13.

4.5.2 Data Structure for Queries with Fixed $p = \pi \leq \Delta$

We build an $o(n/\pi)$-bits structure for handling queries with pattern length $p = \pi$.

Lemma 14. *For any $1 \leq \pi \leq \Delta = \mathcal{O}(\text{polylog}\, n)$ and any constant $\gamma > 0$, there is an $\mathcal{O}(n \log\log n/(\pi \log^\gamma n))$-bits data structure solving queries $(P[1..p], k)$ with $p = \pi$ in $\mathcal{O}(t_{\mathsf{s}}(p) + k(t_{\mathsf{SA}} + \log\log n + \log k)\pi \log^\gamma n)$ time.*

The idea is to build an array $\mathsf{top}[1,n]$ such that a candidate set of size $\mathcal{O}(k)$, for any query (P,k) with $p = \pi$, is given by $\{\mathsf{SA}[i], i \in [sp, ep] \wedge \mathsf{top}[i] \leq k\}$, $[sp, ep]$ being the suffix range of P. The key property to achieve this is that the ranges $[sp, ep]$ are disjoint for all the patterns of a fixed length π. We build top as follows.

1. Initialize $\mathsf{top}[1..n] = n + 1$.
2. For each pattern Q of length π,
   a) Find the suffix range $[\alpha, \beta]$ of Q.
   b) Find the list $d_{r_1}, d_{r_2}, d_{r_3}, \ldots$ of documents in the ascending order of $\mathsf{tp}(Q, \cdot)$ values (ties broken arbitrarily).
   c) For each document d_{r_κ} containing Q at least twice, choose a *unique* critical pair with respect to Q, that is, choose two elements $j, j' \in [\alpha, \beta]$, such that $(m_{r_\kappa}, m_{r_\kappa}') = (\mathsf{SA}[j], \mathsf{SA}[j'])$ is a critical pair of d_{r_κ} with respect to Q. Then assign $\mathsf{top}[j] = \mathsf{top}[j'] = \kappa$.

The following observation is immediate.

Lemma 15. *For a query $(P[1..p], k)$ with $p = \pi$ and suffix array range $[sp, ep]$ for P, the set $\{\mathsf{SA}[j], j \in [sp, ep] \wedge \mathsf{top}[j] \leq k\}$ is a candidate set of size at most $2k$.*

Proof. A valid answer for (P,k) are the document identifiers $r_1,\ldots,r_k$ considered at construction time for $Q=P$. For each such document d_{r_κ}, $1\le\kappa\le k$, we have found a critical pair $(m_{r_\kappa},m'_{r_\kappa})=(\mathsf{SA}[j],\mathsf{SA}[j'])$, for $j,j'\in[sp,ep]$, and set $\mathsf{top}[j]=\mathsf{top}[j']=\kappa\le k$. All the other values of $\mathsf{top}[sp,ep]$ are larger than k. The size of the candidate set is thus $2k$ (or less, if there are less than k documents where P occurs twice). □

However, we cannot afford to maintain top explicitly within the desired space bounds. Therefore, we replace top by a *sampled* array top'. The sampled array is built by cutting top into blocks of size $\pi'=\pi\log^\gamma n$ and storing the logarithm of the minimum value for each block. This will increase the size of the candidate sets by a factor of $\mathcal{O}(\pi')$. More precisely, $\mathsf{top}'[1,n/\pi']$ is defined as

$$\mathsf{top}'[j] \quad = \quad \lceil \log \min F[(j-1)\pi'+1..j\pi'] \rceil$$

Since $\mathsf{top}'[j]\in[0..\log n]$, the array can be represented in $n\log\log n/(\pi\log^\gamma n)$ bits. We represent top' with a multiary wavelet tree [55], which maintains the space in $\mathcal{O}(n\log\log n/(\pi\log^\gamma n))$ bits, and since the alphabet size is logarithmic, supports in constant time operations *rank* and *select* on top'. Operation $rank(j,\kappa)$ counts the number of occurrences of κ in $\mathsf{top}'[1..j]$, whereas $select(j,\kappa)$ gives the position of the jth occurrence of κ in top'.

Query Algorithm. To answer a query $(P[1..p],k)$ with $p=\pi$ using a CSA and top', we compute the suffix range $[sp,ep]$ of P in time $t_{\mathsf{s}}(p)$, and then do as follows.

1. Among all the blocks of top overlapping the range $[sp,ep]$, identify those containing an element $\le 2^{\lceil\log k\rceil}$, that is, compute the set
$$S_{blocks}=\{j,\lceil sp/\pi'\rceil\le j\le\lceil ep/\pi'\rceil\wedge\mathsf{top}'[j]\le\lceil\log k\rceil\}.$$
2. Generate $\mathsf{Cand}(P,k)=\{\mathsf{SA}[j'],j\in S_{blocks}\wedge j'\in[(j-1)\pi'+1,j\pi']\}$.
3. Find the query output from the candidate set $\mathsf{Cand}(P,k)$, using Lemma 13.

For step 1, the wavelet tree representation of top' generates S_{blocks} in time $\mathcal{O}(1+|S_{blocks}|)$: All the 2^t positions $j\in[sp,ep]$ with $\mathsf{top}'[j]=t$ are $j=select(rank(sp-1,t)+i,t)$ for $i\in[1,2^t]$. We notice if there are no sufficient documents if we obtain a $j>ep$, in which case we stop.

The set $\mathsf{Cand}(P,k)$ is a candidate set of (P,k), since any $j\in[sp,ep]$ with $\mathsf{top}[j]\le k$ belongs to some block of S_{blocks}. Also the number of $j\in[sp,ep]$ with $\mathsf{top}[j]\le 2^{\lceil\log k\rceil}$ is at most $2\cdot 2^{\lceil\log k\rceil}\le 4k$, therefore $|S_{blocks}|\le 4k$.

Now, $\mathsf{Cand}(P,k)$ is of size $|S_{blocks}|\pi'=\mathcal{O}(k\pi')$, and it is generated in step 2 in time $\mathcal{O}(k\,t_{\mathsf{SA}}\,\pi')$. Finally, the time for generating the final output using Lemma 13 is $\mathcal{O}(k\pi'\log(k\pi')))=\mathcal{O}(k\pi\log^\gamma n(\log k+\log\log n+\log\pi))$. By considering that $\pi\le\Delta=\mathcal{O}(\mathrm{polylog}\,n)$, we obtain Lemma 14.

Except for $t=0$, which has 2 positions.

4.5.3 Data Structure for Queries with $p > \Delta$

We prove the following result in this section.

Lemma 16. *For any $\Delta = \mathcal{O}(\text{polylog}\, n)$ and any constant $\epsilon > 0$, there is an $\mathcal{O}(n/\log^{\epsilon} n + (n/\Delta)\log^2 n)$-bits structure solving queries $(P[1..p], k)$, with $p > \Delta$, in $\mathcal{O}(t_{\mathsf{s}}(p) + \Delta(\Delta + t_{\mathsf{SA}}) + k \log k \log^{2\epsilon} n(t_{\mathsf{SA}} + \Delta \log^{1+\epsilon} n))$ time.*

We start with a concept similar to that of a candidate set, but weaker in the sense that it is required to contain only one element of each critical pair.

Definition 6. *Let $\mathsf{Topk}(P, k)$ be a valid answer for the top-k query (P, k). A set $\mathsf{Semi}(P, k) \subseteq [n]$ is a* semi-candidate set *of $\mathsf{Topk}(P, k)$ if it contains at least one critical occurrence m_i of P in d_i for each document identifier $i \in \mathsf{Topk}(P, k)$.*

Our structure in this section generates a semi-candidate set $\mathsf{Semi}(P, k)$. Then, a candidate set $\mathsf{Cand}(P, k)$ is generated as the union of $\mathsf{Semi}(P, k)$ and the set of occurrences of P that are immediately before and immediately after every position in $\mathsf{Semi}(P, k)$. This is obviously a valid candidate set. Finally, we apply Lemma 13 on $\mathsf{Cand}(P, k)$ to compute the final output.

4.5.4 Generating a Semi-candidate Set

This section proves the following result.

Lemma 17. *For any constant $\delta > 0$, a structure of $\mathcal{O}(n(\log\log n)^2/\log^{\delta} n)$ bits plus a CSA can generate a semi-candidate set of size $\mathcal{O}(k \log k \log^{\delta} n)$ in time $\mathcal{O}(t_{\mathsf{SA}}\, k \log k \log^{\delta} n)$.*

Let node x be an ancestor of node y in GST. Let $\mathit{Leaf}(x)$ (resp., $\mathit{Leaf}(y)$) be the set of leaves in the subtree of node x (resp., y), and let $\mathit{Leaf}(x \backslash y) = \mathit{Leaf}(x) \setminus \mathit{Leaf}(y)$. Then the following lemma holds.

Lemma 18. *The set $\mathsf{Semi}(\text{path}(y), k) \cup \{\mathsf{SA}[j], \ell_j \in L(x \backslash y)\}$ is a semi-candidate set of $\mathsf{Topk}(\text{path}(x), k)$.*

Proof. Let $i \in \mathsf{Topk}(\text{path}(x), k)$, then our semi-candidate set should contain m_i or m'_i for some critical pair (m_i, m'_i). If there is some such critical pair where m_i or m'_i are occurrences of path(x) but not of path(y), then ℓ_j or $\ell_{j'}$ are in $L(x \backslash y)$, for $\mathsf{SA}[j] = m_i$ and $\mathsf{SA}[j'] = m'_i$, and thus our set contains it. If, on the other hand, both m_i and m'_i are occurrences of path(y) for all critical pairs (m_i, m'_i), then $\mathsf{tp}(\text{path}(y), d) = \mathsf{tp}(\text{path}(x), d)$, and the critical pairs of path(x) are the critical pairs of path(y). Thus $\mathsf{Semi}(y, k)$ contains m_i or m'_i for some such critical pair. □

Our approach is to precompute and store $\mathsf{Semi}(\mathrm{path}(y), k)$ for carefully selected nodes $y \in \mathsf{GST}$ and k values, so that any arbitrary $\mathsf{Semi}(\mathrm{path}(x), k)$ set can be computed efficiently. The succinct framework of Hon et al. [70] is adequate for this.

Node Marking Scheme. The idea [70] is to mark a set Mark_g of nodes in GST based on a *grouping factor* g: Every gth leaf is marked, and the LCA of any two consecutive marked leaves is also marked. Then the following properties hold.

1. $|\mathsf{Mark}_g| \leq 2n/g$.
2. If there exists no marked node in the subtree of x, then $|Leaf(x)| < 2g$.
3. If it exists, then the highest marked descendant node y of any unmarked node x is unique, and $|Leaf(x \backslash y)| < 2g$.

We use this idea, and a later refinement [74]. Let us first consider a variant of Lemma 17 where $k = \kappa$ is fixed at construction time. We use a CSA and an $o(n)$-bit CST on it, see Section 4.4. We choose $g = \kappa \log \kappa \log^{1+\delta} n$ and, for each node $y \in \mathsf{Mark}_g$, we explicitly store a candidate set $\mathsf{Semi}(\mathrm{path}(y), \kappa)$ of size κ. The space required is $\mathcal{O}(|\mathsf{Mark}_g| \kappa \log n) = \mathcal{O}(n/(\log \kappa \log^{\delta} n))$ bits.

To solve a query (P, κ), we find the suffix range $[sp, ep]$, then the locus node of P is $x = \mathsf{LCA}(\ell_{sp}, \ell_{ep})$ (but we do not need to compute x). The node we compute is $y = \mathsf{LCA}(\ell_{g\lceil sp/g \rceil}, \ell_{g\lfloor ep/g \rfloor})$, the highest marked node in the subtree of x, as it has associated the set $\mathsf{Semi}(\mathrm{path}(y), \kappa)$. This takes time $\mathcal{O}(t_{\mathsf{SA}} \log^{\epsilon} n)$ for any constant $\epsilon > 0$ (see Section 4.4). Then, by the given properties of the marking scheme, combined with Lemma 18, a semi-candidate set of size $\mathcal{O}(g + \kappa) = \mathcal{O}(\kappa \log \kappa \log^{1+\delta} n)$ can be generated in $\mathcal{O}(t_{\mathsf{SA}} \kappa \log \kappa \log^{1+\delta} n)$ time.

To reduce this time, we employ dual marking scheme [74]. We identify a larger set $\mathsf{Mark}_{g'}$ of nodes, for $g' = \frac{g}{\log n} = \kappa \log \kappa \log^{\delta} n$. To avoid confusion, we call these *prime* nodes, not marked nodes. For each prime node $y' \in \mathsf{Mark}_{g'}$, we precompute a candidate set $\mathsf{Semi}(\mathrm{path}(y'), \kappa)$ of size κ. Let y be the (unique) highest marked node in the subtree of y'. Then we store κ bits in y' to indicate which of the κ nodes stored in $\mathsf{Semi}(\mathrm{path}(y), \kappa)$ also belong to $\mathsf{Semi}(\mathrm{path}(y'), \kappa)$. By the same proof of Lemma 18, elements in $\mathsf{Semi}(\mathrm{path}(y'), \kappa) \backslash \mathsf{Semi}(\mathrm{path}(y), \kappa)$ must have a critical occurrence in $Leaf(y' \backslash y)$. Then, instead of explicitly storing the critical positions $m_i \in \mathsf{Semi}(\mathrm{path}(y'), \kappa) \backslash \mathsf{Semi}(\mathrm{path}(y), \kappa)$, we store their left-to-right position in $Leaf(y' \backslash y)$. Storing κ such positions in leaf order requires $\mathcal{O}(\kappa \log(g/\kappa)) = \mathcal{O}(\kappa \log \log n)$ bits, using for example gamma codes. The total space is $\mathcal{O}(|\mathsf{Mark}_{g'}| \kappa \log \log n) = \mathcal{O}(n \log \log n / (\log \kappa \log^{\delta} n))$ bits.

Now we can compute CST prime node $y' = \mathsf{LCA}(\ell_{g'\lceil sp/g' \rceil}, \ell_{g'\lfloor ep/g' \rfloor})$ and marked node y, compute $\mathsf{Semi}(\mathrm{path}(y'), \kappa)$ with the help of the precomputed set $\mathsf{Semi}(\mathrm{path}(y), \kappa)$ and the differential information stored at node y', and

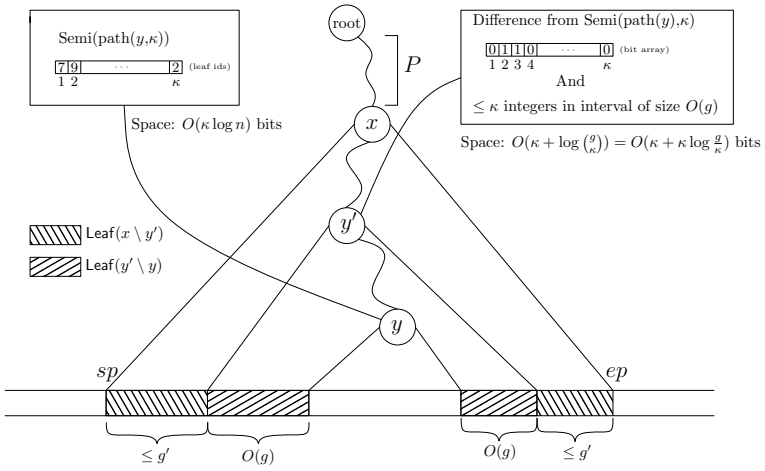


Figure 4.1: Scheme using marked and prime nodes. Set $\mathsf{Semi}(\text{path}(x), \kappa)$ is built from $\mathsf{Semi}(\text{path}(y'), \kappa)$ and $\mathit{Leaf}(x \backslash y')$. Set $\mathsf{Semi}(\text{path}(y'), \kappa)$ is built from selected entries of $\mathsf{Semi}(\text{path}(y), \kappa)$ and selected elements of $\mathit{Leaf}(y' \backslash y)$.

apply the same technique above to obtain a semi-candidate set from $\mathsf{Mark}_{g'}$, yet of smaller size $\mathcal{O}(g' + \kappa) = \mathcal{O}(\kappa \log \kappa \log^{\delta} n)$, in $\mathcal{O}(t_{\mathsf{SA}} \kappa \log \kappa \log^{\delta} n)$ time. Figure 4.1 illustrates the scheme.

We are now ready to complete the proof of Lemma 17. We maintain structures as described for all the values of κ that are powers of 2, in total

$$O\left(\left(n \log\log n / \log^{\delta} n\right) \cdot \sum_{i=1}^{\log D} 1/i\right) = \mathcal{O}(n(\log\log n)^2 / \log^{\delta} n)$$

bits of space. To solve a query (P, k), we compute $\kappa = 2^{\lceil \log k \rceil} < 2k$ and return the semi-candidate set of (P, κ) using the corresponding structure.

4.5.5 Generating the Candidate Set

The problem of obtaining $\mathsf{Cand}(P, k)$ from $\mathsf{Semi}(P, k)$ boils down to the task: given $P[1..p]$ and an occurrence q, find the occurrence of P closest to q. In other words, finding the first and the last occurrence of P in $\mathsf{T}[q + 1..n]$ and $\mathsf{T}[1..q + p - 1]$, respectively. We employ suffix sampling to obtain the desired space-efficient structure. The idea is to exploit the fact that, if $p > \Delta$, then for every occurrence q of P, there must be an integer $j = \Delta\lceil q/\Delta \rceil$ (a multiple of Δ) and $t \leq \Delta$, such that $P[1..t]$ is a suffix of $\mathsf{T}[1..j]$ and $P[t + 1..p]$ is a prefix of $\mathsf{T}[j + 1..n]$. We call q an *offset-t occurrence* of P. Then, $\mathsf{Cand}(P, k)$ can be computed as follows:

1. Find $\mathsf{Semi}(P, k)$ using Lemma 17.

2. For each $q \in \mathsf{Semi}(P, k)$ and $t \in [1, \Delta]$, find the offset-t occurrences of P that are immediately before and immediately after q.

3. The occurrences found in the previous step, along with the elements in $\mathsf{Semi}(P, k)$, constitute $\mathsf{Cand}(P, k)$.

In order to perform step 2 efficiently, we maintain the following structures.

- **Sparse Suffix Tree (SST)**: A suffix $\mathsf{T}[\Delta i + 1..n]$ is a *sparse suffix*, and the trie of all sparse suffixes is a *sparse suffix tree*. The *sparse suffix range* of a pattern Q is the range of the sparse suffixes in SST that are prefixed by Q. Given the suffix range $[sp, ep]$ of a pattern, its sparse suffix range $[ssp, sep]$ can be computed in constant time by maintaining a bitmap $B[1..n]$, where $B[j] = 1$ iff $\mathsf{T}[\mathsf{SA}[j]..n]$ is a sparse suffix. Then $ssp = 1 + rank_B(sp - 1)$ and $sep = rank_B(sp)$. Since B has n/Δ 1s, it can be represented in $\mathcal{O}((n/\Delta) \log \Delta)$ bits while supporting $rank_B$ operation in constant time for any $\Delta = \mathcal{O}(\text{polylog}\, n)$ [106].

- **Sparse Prefix Tree (SPT)**: A prefix $\mathsf{T}[1..\Delta i]$ is a *sparse prefix*, and the trie of the *reverses* of all sparse prefixes is a *sparse prefix tree*. The *sparse prefix range* of a pattern Q is the range of the sparse prefixes in SPT with Q as a suffix. The SPT can be represented as a blind trie [53] using $\mathcal{O}((n/\Delta) \log n)$ bits. Then the search for the sparse prefix range of Q can be done in $\mathcal{O}(|Q|)$ time, by descending using the reverse of Q. Note that the blind trie may return a fake node when Q does not exist in the SPT.

- **Orthogonal Range Successor/Predecessor Search Structure** over a set of $\lceil n/\Delta \rceil$ points of the form (x, y, z), where the yth leaf in SST corresponds to $\mathsf{T}[x..n]$ and the zth leaf in SPT corresponds to $\mathsf{T}[1..(x-1)]$. The space needed is $\mathcal{O}((n/\Delta) \log^2 n)$ bits (recall Lemma 12).

The total space of the structures is $\mathcal{O}((n/\Delta) \log^2 n)$ bits. They allow computing the first offset-t occurrence of P in $\mathsf{T}[q + 1..n]$ as follows: find $[ssp_t, sep_t]$ and $[ssp'_t, sep'_t]$, the sparse suffix range of $P[t+1..p]$ and the sparse prefix range of $P[1..t]$, respectively. Then, using an orthogonal range successor query, find the point $(e, \cdot, \cdot)$ with the lowest x-coordinate value in $[q + t + 1, n] \times [ssp_t, sep_t] \times [ssp'_t, sep'_t]$. Then, $e - t$ is the answer. Similarly, the last offset-t occurrence of P in $\mathsf{T}[1..q - 1]$ is $f - t$, where $(f, \cdot, \cdot)$ is the point in $[1, q + t - 1] \times [ssp_t, sep_t] \times [ssp'_t, sep'_t]$ with the highest x-coordinate value.

First, we compute all the ranges $[ssp_t, sep_t]$ using the SST. This requires knowing the interval $\mathsf{SA}[sp_t, ep_t]$ of $P[t + 1..p]$ for all $1 \le t \le \Delta$. We compute these by using the CSA to search for $P[\Delta + 1..p]$ (in time at most $t_{\mathsf{s}}(p)$), which gives $[sp_\Delta, ep_\Delta]$, and then computing $[sp_{t-1}, ep_{t-1}] = \mathsf{Wlink}(P[t], [sp_t, ep_t])$ for

Using perfect hashing to move in constant time towards the children.

$t = \Delta - 1, \ldots, 1$. Using an $o(n)$-bits CST (see Section 4.4), this takes $\mathcal{O}(\Delta\, t_{\mathsf{SA}})$ time. Then the SST finds all the $[ssp_t, sep_t]$ values in time $\mathcal{O}(\Delta)$. Thus the time spent on the SST searches is $\mathcal{O}(t_{\mathsf{s}}(p) + \Delta\, t_{\mathsf{SA}})$.

Second, we search the SPT for reverse pattern prefixes of lengths 1 to Δ. They can all be searched for in time $\mathcal{O}(\Delta^2)$. The returned interval $[ssp'_t, sep'_t]$ is either the correct interval of $P[1..t]$, or $P[1..t]$ does not terminate any sparse prefix. A simple way to determine which is the case is to perform the orthogonal range search as explained, asking for the successor e_0 of position 1, and check whether the resulting position, $e_0 - t$, is an occurrence of P. That is, check whether $\mathsf{SA}^{-1}[e_0 - t] \in [sp, ep]$. This takes $\mathcal{O}(t_{\mathsf{SA}} + \log^{1+\epsilon} n)$ time per verification. Considering the searches plus verifications, the time spent on the SPT searches is $\mathcal{O}(\Delta(\Delta + t_{\mathsf{SA}} + \log^{1+\epsilon} n))$.

Finally, after determining all the intervals $[ssp_t, sep_t]$ and $[ssp'_t, sep'_t]$, we perform $\mathcal{O}(|\mathsf{Semi}(P,k)|\Delta)$ orthogonal range searches for positions q, in time $\mathcal{O}(|\mathsf{Semi}(P,k)|\Delta \log^{1+\epsilon} n)$, and keep the closest one for each q.

Lemma 19. *Given a semi-candidate set $\mathsf{Semi}(P,k)$, where $p > \Delta$, a candidate set $\mathsf{Cand}(P,k)$ of size $\mathcal{O}(|\mathsf{Semi}(P,k)|)$ can be computed in time $\mathcal{O}(t_{\mathsf{s}}(p) + \Delta(\Delta + t_{\mathsf{SA}} + |\mathsf{Semi}(P,k)| \log^{1+\epsilon} n))$ using a data structure of $\mathcal{O}((n/\Delta)\log^2 n)$ bits.*

Thus, by combining Lemma 17 using $\delta = 2\epsilon$ (so its space is $o(n/\log^{\epsilon} n)$ bits) and Lemma 19, we obtain Lemma 16.

4.5.6 Extension

Up to now we have considered only term proximity. In a more general scenario one would like to use a scoring function that is a linear combination of term proximity, term frequency, and a document score like PageRank (document score counts for d_i only if P appears in d_i at least once). In this section we provide the first result on supporting such a combined scoring function in compact space.

Theorem 8. *Using a $2n\log\sigma + o(n\log\sigma)$ bits data structure, one can answer top-k document retrieval queries, where $\mathsf{score}(\cdot,\cdot)$ is a weighted sum of a document score, term-frequency and term-proximity with predefined non-negative weights, in time $\mathcal{O}(p + k\log k \log^{4+\epsilon} n)$*

Proof. The theorem can be obtained by combing our previous results as follows:

1. Lemma 17 with $\delta > 0$ gives the following: using a $|\mathsf{CSA}| + o(n)$ bits structure, we can generate a sem-candidate set $\mathsf{Semi}(P,k)$ of size $\mathcal{O}(k\log k \log^{\delta} n)$ in time $\mathcal{O}(t_{\mathsf{SA}} k \log k \log^{\delta} n)$. Although the ranking function assumed in Lemma 17 is term-proximity, it is easy to see that the result holds true for our new ranking function $\mathsf{score}(\cdot,\cdot)$ as well: we precompute $\mathsf{Semi}(\mathrm{path}(y), \kappa)$ for $\mathsf{score}(\cdot,\cdot)$ rather than for $\mathsf{tp}(\cdot,\cdot)$. Any document

that is not top-k on node y and does not appear further in $Leaf(x \backslash y)$, cannot be top-k on node x, because its score cannot increase.

2. We wish to compute $\mathsf{tf}(P, i) = ep_i - sp_i + 1$ for each entry of d_i in $\mathsf{Semi}(P, k)$, where $[sp_i, ep_i]$ is the range in the suffix array SA_i of d_i for the pattern P. However, we do not wish to spend time $t_{\mathsf{s}}(p)$, since that could potentially be expensive. Note we have already computed sp and ep. By using these and the compressed suffix array CSA_i, which we will store for each document, we can compute sp_i and ep_i more efficiently as follows. The position in T where d_i begins is $select_B(i-1)$, and $|d_i| = select_B(i) - select_B(i-1)$. Note that we are already given one position m_i in the region of d_i in T by the entry in $\mathsf{Semi}(P, k)$, and we compute the corresponding entry solely in d_i as $m'_i = m_i - select_B(i-1)$. We compute the corresponding point to m'_i in $[sp_i, ep_i]$ as $q = \mathsf{SA}_i^{-1}[m'_i]$. We can now define

$$ep_i = \max\{j \mid j \geq q \wedge j \leq |d_i| \wedge \mathsf{SA}^{-1}[select_B(i-1) + \mathsf{SA}_i[j]] \leq ep\}$$

Which is computed by an exponential search starting from q. A similar equation holds for sp_d [70]. Computing q costs $\mathcal{O}(t_{\mathsf{SA}})$ and the two exponential searches require $\mathcal{O}(t_{\mathsf{SA}} \log n)$ time each.

3. We need to be able to compute the term proximity distance for each d_i in $\mathsf{Semi}(P, k)$. This can be computed in time $\mathcal{O}((t_{\mathsf{SA}} + \log^2 n) \log^{2+\varepsilon} n)$ once we know $[sp_i, ep_i]$ of P in CSA_i by applying Theorem 7: For each document d_i we store the $o(|d_i|)$ extra bits required by the theorem so we can answer queries for $k = 1$. That query will then return the single tp value for the query pattern.

4. The document rank for each document is easily obtained, as it does not depend on the pattern. Finally k documents with the highest $\mathsf{score}(P, i)$ are reported.

By maintaining structures of overall space $|\mathsf{CSA}| + \sum_d |\mathsf{CSA}_d| + o(n)$ bits, any (P, k) query can be answered in $\mathcal{O}(t_{\mathsf{s}}(p) + k \log k \log^{\delta} n (t_{\mathsf{SA}} + \log^2 n) \log^{2+\varepsilon} n)$ time. Using the version of the compressed suffix array by Belazzougui and Navarro [14], where $|\mathsf{CSA}| = n \log \sigma + o(n \log \sigma), t_{\mathsf{s}}(p) = \mathcal{O}(p)$ and $t_{\mathsf{SA}} = \mathcal{O}(\log n \log \log n)$, the space becomes $2n \log \sigma + o(n \log \sigma)$ bits and the query time becomes $\mathcal{O}(p + k \log k \log^{4+\epsilon} n)$. The proof is completed by choosing $0 < \delta, \varepsilon < \epsilon/2$. □

4.5.7 Conclusions

We have presented the first compressed data structures for answering top-k term-proximity queries, achieving the asymptotically optimal $|\mathsf{CSA}| + o(n)$ bits, and query times in $\mathcal{O}((p + k) \operatorname{polylog} n)$. This closes the gap that separated

this relevance model from term frequency and document ranking, for which optimal-space solutions (as well as other intermediate-space tradeoffs) had existed for several years. The plausible hypothesis that term proximity was inherently harder than the other relevance measures, due to its close relation with range restricted searching [72], has then been settled on the negative.

For the case where the ranking function is a weighted average of document rank, term-frequency, and term-proximity, we have introduced a compact-space solution that requires twice the minimum space required to represent the text collection. An interesting challenge is to find an efficient space-optimal data structure that solves this more general problem.

4.6 The Common Colors Problem

In this section we deal with the common colors problem. As mentioned in the introduction giving a data structure for the common colors problem gives data structures for the 2P and 2DSS problems. We also note that the solution for the common colors we give here, can be adapted to also work for the FP problem. The result of this section is captured in the following theorem.

Theorem 9. *An array E of n colors can be indexed in $\mathcal{O}(n)$-word space so that the following query can be answered in $\mathcal{O}(\sqrt{nk}\log^{1/2+\varepsilon} n)$ time: report the unique colors appearing in both $E[a...b]$ and $E[c...d]$, where a, b, c and d are input values, k is the output size and ε is any positive constant.*

4.6.1 The Common Colors Data Structure

First we give an overview, and then present the details of the proposed data structure. Let $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, ..., \sigma_{|\Sigma|}\}$ be the set of colors appearing in E. Without loss of generality we assume $|\Sigma| \leq n$. The main structure is a binary tree Δ (not necessarily balanced) of $|\Sigma|$ nodes, where each color is associated with a unique node in Δ. Specifically, the color associated with a node u is given by $\sigma_{ino(u)}$, where $ino(u)$ is the in-order rank of u. Also we use $\Sigma(u)$ to represent the set of colors associated with the nodes in the subtree of u. Let $[q, r]$ and $[s, t]$ be two given ranges in E, then $Out_{q,r,s,t}$ denotes the set of colors present in both $[q, r]$ and $[s, t]$. We maintain auxiliary data structures for answering the following subqueries efficiently.

Subquery 1. *Given $i \in [1, |\Sigma|]$ and ranges $[q, r]$ and $[s, t]$ is $\sigma_i \in Out_{q,r,s,t}$?*

Subquery 2. *Given $u \in \Delta$ and ranges $[q, r]$ and $[s, t]$ is $\Sigma(u) \cap Out_{q,r,s,t}$ empty?*

Query Algorithm

To answer the query (i.e., find $Out_{a,b,c,d}$, where $[a, b]$ and $[c, d]$ are the input ranges), we perform a *preorder* traversal of Δ. Upon reaching a node u, we issue a Subquery 2: Is $\Sigma(u) \cap Out_{a,b,d,d}$ empty?

- If the answer is *yes* (i.e., empty), we can infer that none of the color associated with any node in the subtree of u is an output. Therefore, we skip the subtree of u and move to the next node in the preorder traversal.
- On the other hand if Subquery 2 at u returns *no*, there exists at least one node v in the subtree of u, where $\sigma_{ino(v)}$ is an output. Notice that v can be the node u itself. Therefore, we first check if $\sigma_{ino(u)} \in Out_{a,b,c,d}$ using Subquery 1. If the query returns *yes*, we report $\sigma_{ino(u)}$ as an output and continue the preorder traversal.

By the end of this procedure, all colors in $Out_{a,b,c,d}$ will have been reported.

Details of the Data Structure

We now present the details. For any node u in Δ, we use n_u to represent the number of elements in E with colors in $\Sigma(u)$. i.e., $n_u = |\{i|E[i] \in \Sigma(u)\}|$. Then, we construct Δ as follows, maintaining the invariant:

$$n_u \leq n\left(\frac{1}{2}\right)^{d(u)}$$

Here $d(u) \leq \log n$ is the number of ancestors of u (i.e. the depth of u). We remark that this property is essential to achieve the result in Lemma 21 stated below. The following recursive algorithm can be used for constructing Δ. Let f_i be the number of occurrences of σ_i in E. Initialize u as the root node and $\Sigma(u) = \Sigma$. Then, find the color $\sigma_z \in \Sigma(u)$, where

$$\sum_{i<z,\sigma_i\in\Sigma(u)} f_i \leq \frac{1}{2}\sum_{\sigma_i\in\Sigma(u)} f_i \text{ and } \sum_{i>z,\sigma_i\in\Sigma(u)} f_i \leq \frac{1}{2}\sum_{\sigma_i\in\Sigma(u)} f_i$$

Partition $\Sigma(u)$ into three disjoint subsets $\Sigma(u_L), \Sigma(u_R)$ and $\{\sigma_z\}$, where

$$\Sigma(u_L) = \{\sigma_i|i < z, \sigma_i \in \Sigma(u)\}$$

$$\Sigma(u_R) = \{\sigma_i|i > z, \sigma_i \in \Sigma(u)\}$$

If $\Sigma(u_L)$ is not empty, then we add a left child u_L for u and recurse further from u_L. Similarly, if $\Sigma(u_R)$ is not empty, we add a right child u_R for u and recurse on u_R. This completes the construction of Δ. Since Δ is a tree of $\mathcal{O}(|\Sigma|) = \mathcal{O}(n)$ nodes, it occupies $\mathcal{O}(n)$ words. The following lemmas summarize the results on the structures for handling Subquery 1 and Subquery 2.

Lemma 20. *Subquery 1 can be answered in $\mathcal{O}(1)$ time using an $\mathcal{O}(n)$-word structure.*

Proof. For each color a one dimensional range-emptiness structure is stored (Lemma 10). The points stored in the structure for color σ are all i where $E[i] = \sigma$. The answer to the query is *no* if and only if at least one of the two intervals $[q, r]$ and $[s, t]$ is empty for that particular color. □

Lemma 21. *There exists an $\mathcal{O}(n)$-word structure for handling Subquery 2 in the following manner:*

- *If $\Sigma(u) \cap Out_{q,r,s,t} = \emptyset$, then return* yes *in time $O\left(\sqrt{n_u}\log^{\varepsilon} n\right)$*
- *Otherwise, one of the following will happen*
  - *Return* no *in $\mathcal{O}(1)$ time*

– *Return the set* $\Sigma(u) \cap Out_{q,r,s,t}$ *in* $O\left(\sqrt{n_u}\log^{\varepsilon} n\right)$ *time*

Proof. See Section 4.6.2. □

By putting all the pieces together, the total space becomes $\mathcal{O}(n)$ words.

Analysis of Query Algorithm

The structures described in Lemma 20 and Lemma 21 can be used as black boxes to support our query algorithm. However, we slightly optimize the algorithm as follows: when we issue Subquery 2 at a node u, and the structure in Lemma 21 returns the set $\Sigma(u) \cap Out_{a,b,c,d}$ (this includes the case where Subquery 2 returns *yes*), we do not recurse further in the subtree of u. Next we bound the total time for all Subqueries.

Let $k = |Out_{a,b,c,d}|$ be the output size and Δ' be the subtree of Δ consisting only of those nodes which we visited processing the query. Then we can bound the size of Δ':

Lemma 22. *The number of nodes in* Δ' *is* $\mathcal{O}(k\log(n/k))$

Proof. The parent of any node in Δ' must be a node on the path from the root to some node u, where $\sigma_{ino(u)} \in Out_{a,b,c,d}$. Since the height of Δ' is at most $\log n$, the number of nodes with depth at least $\log k$ on any path is at most $\log n - \log k = \log(n/k)$. Therefore, number of nodes in Δ' with depth at least $\log k$ is $\mathcal{O}(k\log(n/k))$. Also the total number of nodes in Δ with depth at most $\log k$ is $\mathcal{O}(k)$. □

We spend $\mathcal{O}(1)$ time for Subquery 1 in every node in Δ', which adds up to $\mathcal{O}(|\Delta'|)$. If a node u is an internal node in Δ', then Subquery 2 in u must have returned *no* in $O(1)$ time (otherwise, the algorithm does not explore its subtree). On the other hand, if a node u is a leaf node in Δ', we spend $\sqrt{n_u}\log^{\varepsilon} n$ time. Analyzing the time for Subquery 2 we see

$$\begin{aligned}\sum_{u\in\text{leaves}(\Delta')}\sqrt{n_u}\log^{\varepsilon} n &= O\left(k\log(n/k) + \log^{\varepsilon} n \sum_{u\in\text{leaves}(\Delta')}\sqrt{\tfrac{1}{2}^{d(u)}n}\right)\\ &= O\left(k\log(n/k) + \sqrt{n}\log^{\varepsilon} n \sum_{u\in\text{leaves}(\Delta')}2^{-d(u)/2}\right)\end{aligned}$$

By Cauchy-Schwarz' inequality we have $\sum_{i=1}^{n} x_i y_i \le \sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}$, which gives us

$$\sqrt{n}\log^{\varepsilon} n \sum_{u\in\text{leaves}(\Delta')}2^{-d(u)/2} = O\left(\sqrt{n}\log^{\varepsilon} n\sqrt{\sum_{u\in\text{leaves}(\Delta')}1^2 \sum_{u\in\text{leaves}(\Delta')}2^{-d(u)}}\right)$$

We know from Kraft's inequality in any binary tree, $\sum_{\ell \in leaves} 2^{-d(\ell)} \leq 1$, which gives us

$$O\left(\sqrt{n}\log^{\varepsilon} n \sqrt{\sum_{u \in \text{leaves}(\Delta')} 1^2 \sum_{u \in \text{leaves}(\Delta')} 2^{-d(u)}}\right) = O\left(\sqrt{n}\log^{\varepsilon} n \sqrt{k \log(n/k) \times 1}\right)$$

Thus we finally bound the running time:

$$\sum_{u \in \text{leaves}(\Delta')} \sqrt{n_u} \log^{\varepsilon} n + \mathcal{O}(\Delta') = O\left(\sqrt{nk} \log^{1/2+\varepsilon} n\right)$$

This completes the proof of Theorem 9.

Corollary 2. *The 2P, FP, and 2DSS problems can be solved using $\mathcal{O}(n)$ words of space and having $O\left(\sqrt{nk} \log^{1/2+\varepsilon} n\right)$ query time, where n is the total number of characters in the input documents and k is the output size of the query.*

Proof. The reduction from 2DSS to the common colors query problem comes from [54], and by the discussion in the introduction 2P and 2DSS are equivalent up to constant factors. □

4.6.2 Proof of Lemma 21

The following are the building blocks of our $\mathcal{O}(n)$-word structure.

1. For every node u in Δ, we define (but not store) E_u as an array of length n_u, where $E_u[i]$ represents the ith leftmost color in E among all colors in $\Sigma(u)$. Thus for any given range $[x, y]$, the list of colors in $E[x...y]$, which are from $\Sigma(u)$ appears in a contiguous region $[x_u, y_u]$ in E_u, where

   - $x_u = 1+$ the number of elements in $E[1...x-1]$, which are from $\Sigma(u)$.
   - $y_u =$ the number of elements in $E[1...y]$, which are from $\Sigma(u)$.

   Notice that x_u and y_u can be computed as the tree is traversed by storing 3 bit arrays each of length n_u and with a rank/select structure in every node as follows (Lemma 8). Let u_L and u_R be the left, respectively right, child of u. Then the first bit array, B_L has a 1 in position i if and only if $E_u[i] \in \Sigma(u_L)$, i.e. the colors that are assigned to the left subtree. Similarly store a bit array, B_R, for the colors that are associated with the right subtree and finally a bit array, B, for the color that is stored at u. This uses $\mathcal{O}(n_u)$ bits per node. Since every level then uses $\mathcal{O}(n)$ bits and there are $\mathcal{O}(\log n)$ levels this gives $\mathcal{O}(n)$ words in total. To translate the ranges from a node u to its left (similarly for right) child u_L (u_R) we perform the following computation

- $x_{u_L} = rank(B_L, x_u - 1)$ (Right child: $x_{u_R} = rank(B_R, x_u - 1)$)
- $y_{u_L} = rank(B_L, y_u)$ (Right child: $y_{u_R} = rank(B_R, y_u)$)

2. An $\sqrt{n_u} \times \sqrt{n_u}$ boolean matrix M_u, for every node u in Δ. For this, we first partition E_u into blocks of size $\sqrt{n_u}$, where the ith block is given by $E_u[1 + (i-1)\sqrt{n_u}, i\sqrt{n_u}]$. Notice that the number of blocks is at most $\sqrt{n_u}$. Then, $M_u[i][j] = 1$, iff there is at least one color, which appears in both the ith block and the jth block of E_u. We also maintain a two-dimensional range maximum query structure (RMQ) with constant query time (Lemma 9) over each M_u. The total space required is $\mathcal{O}(\sum_{u\in\Delta} n_u) = O(n \sum_{u\in\Delta} 2^{-d(u)}) = \mathcal{O}(n \log n)$ bits.

3. Finding the leftmost/rightmost element in $E_u[x...y]$, which is from $\Sigma(u)$, for any given x, y, and u can be reduced to an orthogonal successor/predecessor query: let $S = \{(i, \sigma) \mid E[i] = \sigma\}$. Since $\Sigma(u)$ is a contiguous range from some a to some b (a and b are color numbers) the query simply becomes $[x, \infty] \times [a, b]$. We therefore maintain an $\mathcal{O}(n)$-word structure on S for supporting this query in $\mathcal{O}(\log^{\varepsilon} n)$ time (Lemma 11).

We use the following steps to answer if $\Sigma(u) \cap Out_{q,r,s,t}$ is empty.

1. Find $[q_u, r_u]$ and $[s_u, t_u]$ in $\mathcal{O}(1)$ time.

2. Find $[q'_u, r'_u]$ and $[s'_u, t'_u]$, the ranges that correspond to the longest spans of blocks within $E_u[q_u, r_u]$ and $E_u[s_u, t_u]$ respectively. Notice that $q'_u - q_u, r_u - r'_u, s'_u - s_u, t_u - t'_u \in [0, \sqrt{n_u})$. Check if there is at least one common in both $E_u[q'_u, r'_u]$ and $E_u[s'_u, t'_u]$ with the following steps.

   - Perform an RMQ on M_u with R as the input region, where $R = [1 + (q' - 1)/\sqrt{n_u}, r'/\sqrt{n_u}] \times [1 + (s' - 1)/\sqrt{n_u}, t'/\sqrt{n_u}]$.
   - If the maximum value within R is 1, then we infer that there is one color common in $E_u[q'_u, r'_u]$ and $E_u[s'_u, t'_u]$. Also we can return *no* as the answer to Subquery 2.

   The time spent so far is $\mathcal{O}(1)$.

3. If the maximum value within R in the previous step is 0, we need to do some extra work. Notice that any color, which is an output must have an an occurrence in at least one of the following spans $E_u[q_u, q'_u - 1], E_u[r'_u + 1, r_u], E_u[s_u, s'_u - 1], E_u[t'_u + 1, t_u]$ of length at most $4\sqrt{n_u}$. Therefore, these colors can be retrieved using $\mathcal{O}(\sqrt{n_u})$ successive orthogonal predecessor/successor queries and with Subquery 1, we can verify if a candidate belongs to the output. The total time required is $\mathcal{O}(\sqrt{n_u} \log^{\varepsilon} n)$.

This completes the proof of Lemma 21.

4.6.3 Construction time

The data structure consists of several smaller building blocks and in this section we give the construction time for the entire data structure. A range emptiness structure is used for each color to answer Subquery 1. Constructing a range-emptiness structure on m points takes $O(mw^{\varepsilon})$ (w is the word size) time in expecation [92]. Since that structure is built for each color the total time becomes $O(n \log^{\varepsilon} n)$.

Next the tree Δ is built and for each node $u \in \Delta$ three rank/select structures, the matrix M_u as well as the Range Maximum Query structure based on M_u are constructed. Building a rank/select strucutre on a bit-array with n bits takes $\mathcal{O}(n)$ time [112]. In each node u of Δ three such structures with $\mathcal{O}(n_u)$ bits each are built. For any level of Δ this adds up to at most n bits, giving us $\mathcal{O}(n)$ time per layer of Δ. Since there are $\mathcal{O}(\log n)$ levels of Δ we get $\mathcal{O}(n \log n)$ time in total. Note that the 1 bits in the arrays can be computed by a single scan through the E_u array (which is thrown away after construction). A Range Maximum Query structure on an $\sqrt{n_u}$ by $\sqrt{n_u}$ matrix can be built in $\mathcal{O}(n_u)$ time [24]. However, the most time consuming part of the data structure are all the matrices M_u. In the matrix M_u the i,jth entry is computed by iterating over the interval of length $\sqrt{n_u}$ the i corresponds to and performing range emptiness queries in the interval j corresponds to. The i,jth bit is 1 if and only if at least one of the queries returns 'non-empty'. Since each interval is of length $\sqrt{n_u}$ and a query takes $\mathcal{O}(1)$ time it takes $O(\sqrt{n_u})$ time to compute each bit in M_u. In total it takes $n_u\sqrt{n_u}$ time to built M_u. We get the total time for computing all the M_u matrices:

$$\sum_{u \in \Delta} n_u \sqrt{n_u} \leq \sum_{i=0}^{\log n} 2^i \left(\tfrac{1}{2}\right)^i n \sqrt{\left(\tfrac{1}{2}\right)^i n} = n\sqrt{n} \sum_{i=0}^{\log n} \left(\tfrac{1}{\sqrt{2}}\right)^i = \mathcal{O}(n\sqrt{n})$$

And this is also the total construction time.

4.7 Hardness Results

The hardness results are reductions from *boolean matrix multiplication*. Through this section we use similar techniques to [34, 36, 37]. In the boolean matrix multiplication problem we are given two $n \times n$ matrices A and B with $\{0, 1\}$ entries. The task is to compute the boolean product of A and B, that is replace multiplication by logical and, and replace addition by logical or. Letting $a_{i,j}, b_{i,j}, c_{i.j}$ denote entry i, j of respectively A, B and C the task is to compute for all i, j

$$c_{i,j} = \bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j}). \tag{4.1}$$

The asymptotically fastest algorithm known for matrix multiplication currently uses $\mathcal{O}(n^{2.3728639})$ time [63]. This bound is achieved using algebraic techniques (like in Strassen's matrix multiplication algorithm) and the fastest combinatorial algorithm is still cubic divided by some poly-logarithmic factor [13]. There is not an exact definition on what characterizes and algebraic algorithm compared to a combinatorial algorithm. The easiest way to think of these terms in the context of matrix multiplication is that an algebraic algorithm fails when the operations are $(\min, +)$ rather than $(+, \times)$, since the min operation does not have an inverse.

In this section, we prove that the problem of multiplying two $\sqrt{n} \times \sqrt{n}$ boolean matrices A and B can be reduced to the problem of indexing $\mathcal{D}$ (in 2P or FP) and answering n counting queries. This is evidence that unless better matrix multiplication algorithms are discovered we should not expect to be able to preprocess the data and answer the queries much faster than $\Omega((\sqrt{n})^{\omega})) = \Omega(n^{\omega/2})$ (ignoring poly-logarithmic factors) where ω is the matrix multiplication exponent. In other words one should not expect to be able to have small preprocessing and query time simultaneously. Currently we cannot achieve better than $\Omega(n^{1.18635})$ preprocessing time and $\Omega(n^{0.18635})$ query time simultaneously.

We start the next section with a brief discussion on how to view boolean matrix multiplication as solving many set intersection problems. Then we give the reductions from the matrix multiplication problem to 2P and describe how to adapt it for FP.

4.7.1 Boolean Matrix Multiplication

A different way to phrase the boolean matrix multiplication problem is that entry $c_{i,j} = 1$ if and only if $\exists k : a_{i,k} = b_{k,j} = 1$. For any two matrices A and B let $A_i = \{k \mid a_{i,k} = 1\}$ and similarly let $B_j = \{k \mid b_{k,j} = 1\}$. It follows that $c_{i,j} = 1$ if and only if $A_i \cap B_j \neq \emptyset$. In this manner we view each row of matrix A as a set containing the elements corresponding to the indices where there is a 1, and similarly for columns in B. For completeness we also use $\overline{A_i} = \{k \mid a_{i,k} = 0\}$ and $\overline{B_i} = \{k \mid b_{k,j} = 0\}$.

A naive approach to solving 2P would be to index the documents such that we can find all documents containing a query pattern fast. This way a query would be to find all the documents that P_1 occurs in and the documents that P_2 occurs in separately and then return the intersection of the two result sets. This is obviously not a good solution in the worst case, but it illustrates that the underlying challenge is to solve set intersection.

We observed that boolean matrix multiplication essentially solves set intersection between rows of A and columns of B, so the idea for the reductions is to use the fact that queries for 2P and FP essentially also solve set intersection. We now give the reductions.

4.7.2 The Reductions

We first relax the data structure problems. Instead of returning a list of documents satisfying the criteria we just want to know whether the list is empty or not, i.e. return 0 if empty and 1 if nonempty.

Let A and B be two $\sqrt{n} \times \sqrt{n}$ boolean matrices and suppose we have an algorithm for building the data structure for the relaxed version of 2P. We now wish to create a set of strings $\mathcal{D}$ based on A and B, build the data structure on $\mathcal{D}$ and peform n queries, one for each entry in the product of A and B. In the following we need to represent a subset of $\{0, 1, \ldots, 2\sqrt{n}\}$ as a string, which we do in the following manner.

Definition 7. *Let* $X = \{x_1, x_2, \ldots, x_\ell\} \subseteq \{0, 1, \ldots, 2\sqrt{n}\}$ *and* $\text{bin}(\cdot)$ *gives the binary representation of the number* $\cdot$ *using* $\left\lceil \frac{1}{2} \log n + 1 \right\rceil$ *bits, then we represent the set* X *as* $\text{str}(X) = \text{bin}(x_1)\#\text{bin}(x_2)\#\cdots\#\text{bin}(x_\ell)\#$.

For the matrix A we define $\sqrt{n}$ strings: $d_1^A, d_2^A, \ldots, d_{\sqrt{n}}^A$ and similarly for B we define $d_1^B, d_2^B, \ldots, d_{\sqrt{n}}^B$. Construct $d_j^B = \text{str}(\{k + \sqrt{n} \mid k \in B_j^T\})$ and $d_i^A = \text{str}(A_i^T)$. We construct the $\sqrt{n}$ strings in $\mathcal{D}$ as: $d_\ell = d_\ell^A d_\ell^B$ for $1 \leq \ell \leq \sqrt{n}$.

Lemma 23. *Each string in* $\mathcal{D}$ *is at most* $\mathcal{O}(\sqrt{n} \log n)$ *characters long.*

Proof. There are at most $\sqrt{n}$ elements in A_ℓ^T and at most $\sqrt{n}$ elements in B_ℓ^T. Each element in A_ℓ^T and B_ℓ^T contributes exactly one number to the string d_ℓ and one '#'. Each number uses exactly $\left\lceil \frac{1}{2} \log n + 1 \right\rceil$ characters. In total we get $\left(|A_\ell^T| + |B_\ell^T| \right) \left(\left\lceil \frac{1}{2} \log n + 1 \right\rceil + 1 \right) \leq 2\sqrt{n} \left\lceil \frac{1}{2} \log n + 1 \right\rceil + 2\sqrt{n} = \mathcal{O}(\sqrt{n} \log n)$ □

Corollary 3. *The total length of the strings in* $\mathcal{D}$ *is* $\sum_{i=1}^{\sqrt{n}} d_i = \mathcal{O}(n \log n)$.

Proof. Follows since there are at most $\sqrt{n}$ strings in $\mathcal{D}$ and by Lemma 23 each string is at most $\mathcal{O}(\sqrt{n} \log n)$ characters long. □

We have now created the set of strings, $\mathcal{D}$, that we wish to build the data structure on. We now specify the queries and prove that using these queries we can solve the matrix multiplication problem.

Lemma 24. *The entry $c_{i,j} = 1$ if and only if the query $P_1 = \text{bin}(i)$, $P_2 = \text{bin}(\sqrt{n} + j)$ returns 1.*

Proof. Suppose that $c_{i,j} = 1$. Then by a previous discussion there must exist a k such that $a_{i,k} = b_{k,j} = 1$. Since document d_k is constructed from the kth column of A, we get that $\text{bin}(i)$ occurs as a substring in D_k. Similarly, d_k is also constructed from the kth row of B, meaning $\text{bin}(\sqrt{n} + j)$ occurs as a substring in D_k. It follows that the string d_k satisfies the conditions and the query returns 1.

Suppose the query $(P_1,\ P_2)$ returns 1, then by definition there exists a $d_\ell \in \mathcal{D}$ such that $\text{bin}(i)$ occurs in d_ℓ and $\text{bin}(\sqrt{n} + j)$ occurs in d_ℓ. All numbers in d_ℓ but the first are surrounded by '#' and all numbers are of length $|P_1|$. Furthermore any number in d_ℓ less than $\sqrt{n}$ is only there because of a 1 in column ℓ of A. In particular $a_{i,\ell} = 1$, otherwise d_ℓ would not satisfy the conditions. Additionally by construction the binary representation of any number $2\sqrt{n} \geq m > \sqrt{n}$ appears in d_ℓ if and only if $b_{\ell,m} = 1$. In particular $\text{bin}(j + \sqrt{n})$ did occur in d_ℓ, therefore $b_{\ell,j} = 1$. We now have a witness (ℓ) where $a_{i,\ell} = b_{\ell,j} = 1$, and we conclude $c_{i,j} = 1$. □

We are now able to give the following theorems:

Theorem 10. *Let $P(n)$ be the preprocessing time for building the data structure for 2P on a set of strings of total length n and let $Q(n, m)$ be the query time for a pattern with length m. In time $\mathcal{O}(P(n \log n)+n\cdot Q(n \log n, \mathcal{O}(\log n))+ n \log n)$ we can compute the product of two $\sqrt{n} \times \sqrt{n}$ boolean matrices.*

Proof. Follows by the lemmas and the discussion above. □

Similarly for FP we obtain:

Theorem 11. *Let $P(n)$ be the preprocessing time for building the data structure for FP on a set of strings of total length n and let $Q(n)$ be the query time. In time $\mathcal{O}(P(n \log n) + n \cdot Q(n \log n, \mathcal{O}(\log n)) + n \log n)$ we can compute the product of two boolean matrices.*

Proof. In the reduction above substitute B_j with $\overline{B_j}$, P_1 and P_2 with P^+ and P^- respectively. □

As a side note, observe that if we replace the problems by their counting version (i.e. count the number of strings in $\mathcal{D}$ that satisfy the condition) then using the same reductions these problems solve matrix multiplication where the input is 0/1 matrices and the operations are addition and multiplication. Also note that if the preprocessing time is small then with current knowledge

there must either be an exponential time dependency on the pattern length or a polynomial dependency on the total length of the documents.

4.8 Wild Card Indexing Lower Bounds

In this section we consider the wild card indexing (WCI) problem and prove both space and query lower bounds in the pointer machine model of computation. Note that our query lower bound even applies to an alphabet size of two (i.e., binary strings).

4.8.1 The Query Lower Bound

Our lower bound addresses the following class of data structures. Consider an algorithm $\mathscr{A}$ such that, given *any* set of documents with total size n, and parameters and m and κ, it builds a pointer-machine structure $\mathscr{D}$, such that, $\mathscr{D}$ consumes $S(n, m, \kappa)$ space and it is able to answer *any* WCI query in $Q(n, \kappa) + \mathcal{O}(m + t)$ time, where t is the output size, m is the pattern length, and κ is the number of wild cards. Here, $Q(\cdot)$ and $S(\cdot)$ are universal functions that only depend on their parameters. Our main result here is the following.

Theorem 12. *Assume $\kappa \geq 3\sqrt{\log n}$. If $Q(n, \kappa) = \mathcal{O}(2^{\kappa/2})$, then $S(n, m, \kappa) = \Omega(n2^{\Theta(\kappa)}n^{\Theta(1/\log \kappa)})$.*

To prove the lower bound, we build a *particular* set of documents and patterns and prove that if the data structure $\mathscr{D}$ can answer the queries fast, then it must consume lots of space, for this particular input, meaning, we get lower bounds for the function $S(\cdot)$. We now present the details. We assume $Q(n, \kappa) \leq 2^{\kappa/2}$, as otherwise the theorem is trivial.

Documents and patterns. We build the set of documents in two stages. Consider the set of all bit strings of length m with exactly $\ell = \kappa/2$ "1"s. In the first stage, we sample each such string uniformly and independently with probability r^{-1} where $r = 2^{\kappa/3}$. Let $\mathcal{D}$ be the set of sampled strings. In the second stage, for every set of $\ell + \ell'$ indices, $1 \leq i_1 < i_2 < \cdots < i_{\ell+\ell'} \leq m$, where $\ell' = (\log_\ell r)/2 = \Theta(\kappa/\log \kappa)$, we perform the following operation, given another parameter β: if there are more than β strings in $\mathcal{D}$ that have "1"s only among positions $i_1, \cdots, i_{\ell+\ell'}$, then we remove all such strings from $\mathcal{D}$. Consequently, among the remaining strings in $\mathcal{D}$, "1"s in every subset of β strings will be spread over at least $\ell + \ell' + 1$ positions. The set of remaining strings $\mathcal{D}$ will form our input set of documents. Now we consider the set $\mathcal{P}$ of all the patterns of length m that have exactly κ wild cards and $m - \kappa$ "0"s. We remove from $\mathcal{P}$ any pattern that matches fewer than $\binom{\kappa}{\ell}/(2r)$ documents from $\mathcal{D}$. The remaining patterns in $\mathcal{P}$ will form our query set of patterns.

Lemma 25. *With positive probability, we get a set $\mathcal{D}$ of $\Theta(\binom{m}{\ell}/r)$ documents and a set $\mathcal{P}$ of $\Theta(\binom{m}{\kappa})$ patterns such that (i) each pattern matches $\Theta(\binom{\kappa}{\ell}/r)$ documents, and (ii) there are no $\beta = \Theta(\log_\kappa m)$ documents whose "1"s are contained in a set of $\ell + \ell'$ indices.*

Proof. Observe that the second stage of our construction guarantees property (ii). So it remains to prove the rest of the claims in the lemma.

Consider a sampled document (bit string) d. Conditioned on the probability that d has been sampled, we compute the probability that d gets removed in the second stage of our sampling (due to conflict with $\beta - 1$ other sampled documents).

Let $I \subset [m]$ be the set of indices that describe the position of "1"s in d. We know that $|I| = \ell$ by construction. Consider a subset $I' \subset [m] \setminus I$ with $|I'| = \ell'$. By our construction, we know that if there are $\beta - 1$ other sampled documents whose "1"s are at positions $I \cup I'$, then we will remove d (as well as all those $\beta - 1$ documents). We first compute the probability that this happens for a fixed set I' and then use the union bound. The total number of documents that have "1"s in positions $I \cup I'$ is

$$\binom{\ell + \ell'}{\ell'} \leq \left(\frac{\ell e}{\ell'}\right)^{\ell'} < \ell^{\ell'} \leq \sqrt{r} \tag{4.2}$$

and thus we expect to sample at most $1/\sqrt{r}$ of them. We bound the probability that instead β documents among them are sampled. We use the Chernoff bound by picking $\mu = 1/\sqrt{r}$, and $(1 + \delta)\mu = \beta$ and we obtain that the probability that β of these documents are sampled is bounded by

$$\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leq \left(\frac{\mathcal{O}(1)}{\sqrt{r}\beta}\right)^{\beta}.$$

We use the union bound now. The total number of possible ways for picking the set I' is $\binom{m-\ell}{\ell'}\binom{\ell}{\ell'} < m^{2\ell'}$ which means the probability that we remove document d at the second stage of our construction is less than

$$\left(\frac{\mathcal{O}(1)}{\sqrt{r}\beta}\right)^{\beta} m^{2\ell'} < \frac{1}{r^6} = 2^{-2\kappa}$$

if we pick $\beta = c \log_{\kappa} m$ for a large enough constant c. Thus, most documents are expected to survive the second stage of our construction. Now we turn our attention to the patterns.

For a fixed pattern $p \in \mathcal{P}$, there are $\binom{\kappa}{\ell} = \binom{\kappa}{\kappa/2} \geq 2^{\kappa}/\kappa$ documents that could match p and among them we expect to sample $\binom{\kappa}{\ell}/r$ documents. An easy application of the Chernoff bound can prove that with high probability, we will sample a constant fraction of this expected value, for every pattern, in the first stage of our construction. Since the probability of removing every document in the second stage is at most $2^{-2\kappa}$, the probability that a pattern is removed at the second stage is less than $2^{2\kappa/3}/(2\kappa)2^{-2\kappa} < 2^{-\kappa}$ and thus, we expect a constant fraction of them to survive the second stage. □

To prove the lower bound, we use the pointer machine framework of Afshani [2] which was originally designed for "geometric stabbing problems": given an input set of n geometric regions, the goal is store them in a data structure such that given a query point, one can output the subset of regions that contain the query point. The framework is summarized below.

Theorem 13. *[2] Assume one can construct a set of n geometric regions inside the d-dimensional unit cube such that (i) every point of the unit cube is contained in at least t regions, and (ii) the volume of the intersection of every β regions is at most v, for some parameters β, t, and v. Then, for any pointer-machine data structure that uses $S(n)$ space and can answer geometric stabbing queries on the above input in time $g(n)+\mathcal{O}(k)$, where k is the output size and $g(\cdot)$ is some increasing function, we must either have $g(n) > t$, or $S(n) = \Omega(tv^{-1}2^{-\mathcal{O}(\beta)})$.*

As remarked by Afshani [2], the framework does not need to be operated in the d-dimensional unit cube and in fact any measure could be substituted instead of the d-dimensional Lebesgue measure; we use a discrete measure here: each pattern is modelled as a "discrete" point with measure $\frac{1}{|\mathcal{P}|}$, meaning, the space of all patterns has measure one. Each document forms a range: a document d_i contains all the patterns (discrete points) that match d_i. Thus, the measure of every document $d_i \in \mathcal{D}$ is $t_i/|\mathcal{P}|$, where t_i is the number of patterns that match d_i. We consider the measure of the intersection of β documents (regions) $d_1, \ldots, d_\beta$. By Lemma 25, there are $\ell + \ell'$ indices where one of these documents has a "1"; any pattern that matches all of these documents must have a wild card in all of those positions. This means, there are at most $\binom{m-\ell-\ell'}{\kappa-\ell-\ell'}$ patterns that could match documents $d_1, \ldots, d_\beta$. This means, when we consider documents as ranges, the intersection of every β documents has measure at most $\binom{m-\ell-\ell'}{\kappa-\ell-\ell'}/|\mathcal{P}|$ which is an upper bound for parameter v in Theorem 13. For the two other parameters t and $g(n)$ in the theorem we have, $t = \Theta(\binom{k}{\ell}/r)$ (by Lemma 25) and $g(n) = Q(n,\kappa) + \mathcal{O}(m)$. Thus, we have satisfied all the requirements of the theorem. Now, we consider the consequences.

We will maintain the condition that $t \geq Cm$, for a large enough constant C. Since we have assumed that $f(n,\kappa) \leq 2^{\kappa/2} < t$, by picking C large enough we can guarantee that $t > g(n)$, which means Theorem 13 gives us a space lower bound of $\Omega(tv^{-1}2^{-\mathcal{O}(\beta)})$.

We now need to plug in the parameters. Recall that $\ell = \kappa/2$ and $|\mathcal{P}| = \Theta(\binom{m}{\kappa})$ by Lemma 25. Furthermore, we would like to create an input of size $\Theta(n)$ which means the number of sampled documents must be $\Theta(n/m)$.

A curious reader can verify that for this input instance Chazelle's framework does not give a lower bound.

In [2] this is stated as "exactly t ranges" but the proof works with only a lower bound on t.

Thus, we need to pick m such that $\binom{m}{\ell}/r = \Theta(n/m)$, or in other words, $m(m-1)\dots(m-\ell+1) = \Theta(n\ell!2^{\kappa/3}/m)$ (a looser bound is $m = \Theta(\kappa n^{1/\ell})$). By carefully simplifying the binomial coefficients that are involved in parameter v, we get that $S(n) = \Omega(n/m2^{\kappa/3}(m/\kappa)^{\Theta(\ell')})$. By substituting $m = \Theta(\kappa n^{2/\kappa})$ and $\ell' = \Theta(\kappa/\log\kappa)$ we get that $S(n) = \Omega(n2^{\Theta(\kappa)}n^{\Theta(1/\log\kappa)})$. It thus remains to satisfy the condition $t \geq Cm$, for a large enough constant C. Since $t > 2^{k/2}$, we need to satisfy $2^{k/2} \geq Cm$ or $k/2 \geq \mathcal{O}(1) + \log m = \mathcal{O}(\log k) + (2\log n)/k$. It can be seen that this is satisfied as long as $\kappa \geq 3\sqrt{\log n}$. This concludes the proof of Theorem 12.

4.8.2 The Space Lower Bound

Here we prove the following theorem.

Theorem 14. *Any pointer-machine data structure that answers WCI queries with κ wild cards in time $Q(n) + \mathcal{O}(m + t)$ over an input of size n must use $\Omega\left(\frac{n}{\kappa}\Theta\left(\frac{\log_{Q(n)} n}{\kappa}\right)^{\kappa-1}\right)$ space, as long as $\kappa < \log_{Q(n)} n$, where t is the output size, and m is the pattern length.*

To prove the lower bound we use the framework of Chazelle and Rosenberg [43]. However, we will need a slightly improved version of their framework that is presented in the following lemma. The bounds we need for our construction to work are tighter than what was needed in [43]. Because of this fact we have to refine the framework before it can be used.

Lemma 26. *Let $\mathcal{U}$ be a set of n input elements and $\mathcal{Q}$ a set of queries where each query outputs a subset of $\mathcal{U}$. Assume there exists a data structure that uses $S(n)$ space and answers each query in $Q(n) + ak$ time, where k is the output size. Assume*

1. *The output size of any query $q \in \mathcal{Q}$, denoted by $|\mathcal{U} \cap q|$, is at least t, for a parameter $t \geq Q(n)$ and*
2. *For integers ℓ and β, and indices, $i_1, \cdots, i_\ell$, $|\mathcal{U} \cap q_{i_1} \cap \cdots \cap q_{i_\ell}| < \beta$.*

Then,

$$S(n) = \Omega\left(\frac{|\mathcal{Q}|t}{\ell \cdot 2^{\mathcal{O}(a\beta)}}\right)$$

Proof. The proof is very similar to the one found in [43], but we count slightly differently to get a better dependency on β. Recall the data structure is a graph where each node stores 2 pointers and some input element. At the query time, the algorithm must explore a subset of the graph. The main idea is to show that the subsets explored by different queries cannot overlap too much, which would imply that there must be many vertices in the graph, i.e., a space lower bound.

By the assumptions above a large fraction of the visited nodes during the query time will be output nodes (i.e., the algorithm must output the value stored in that memory cell). We count the number of nodes in the graph by partitioning each query into sets with β output nodes. By assumption 2 each such set will at most be counted ℓ times. We need the following fact:

Fact 1 ([2], Lemma 2)**.** *Any binary tree of size ct with t marked nodes can be partitioned into subtrees such that there are $\Theta(\frac{ct}{\beta})$ subtrees each with $\Theta(\beta)$ marked nodes and size $\Theta(\frac{ct}{\beta})$, for any $\beta \geq 1$.*

In this way we decompose all queries into these sets and count them. There are $|\mathcal{Q}|$ different queries, each query gives us $\frac{ct}{\beta}$ sets. Now we have counted each set at most ℓ times, thus there are at least $\frac{ct|Q|}{\beta\ell}$ distinct sets with β output nodes. On the other hand we know that starting from one node and following at most $a\beta$ pointers we can reach at most $2^{\mathcal{O}(a\beta)}$ different sets (Catalan number). In each of those sets there are at most $\binom{a\beta}{\beta}$ possibilities for having a subset with β marked nodes. This gives us an upper bound of $S(n)2^{\mathcal{O}(a\beta)}\binom{a\beta}{\beta}$ for the number of possible sets with β marked nodes. In conclusion we get

$$S(n)2^{\mathcal{O}(a\beta)}\binom{a\beta}{\beta} \geq \frac{ct|\mathcal{Q}|}{\beta\ell} \Rightarrow S(n) = \Omega\left(\frac{t|\mathcal{Q}|}{\beta\ell 2^{\mathcal{O}(a\beta)}}\right) = \Omega\left(\frac{t|\mathcal{Q}|}{\ell 2^{\mathcal{O}(a\beta)}}\right)$$

□

Unlike the case for the query lower bound, we build the set of *queries* in two stages. In the first stage, we consider all documents of length m over the alphabet $[\sigma]$ (that is $[\sigma]^m$) and independently sample n/m of them (with replacement) to form the initial set $\mathcal{D}$ of input documents. And for queries, we consider the set $\mathcal{Q}$ of all strings of length m over the alphabet $[\sigma] \cup \{*\}$ containing exactly κ wild cards (recall that $*$ is the wild card character). In total we have $|\mathcal{Q}| = \binom{m}{\kappa}\frac{\sigma^m}{\sigma^\kappa}$ queries. In the second stage, for a parameter β, we consider all pairs of queries and remove both queries if the number of documents they both match is β or more. No document is removed in this stage. We now want to find a value of β such that we retain almost all of our queries after the second stage.

The probability that a fixed query matches a random document is $\frac{\sigma^\kappa}{\sigma^m}$. There are in total $|\mathcal{D}|$ documents, meaning we expect a query output $t = \frac{\sigma^\kappa}{\sigma^m}|\mathcal{D}|$ documents. By an easy application of Chernoff bound we can prove that with high probability, all queries output $\Theta(\frac{\sigma^\kappa}{\sigma^m}|\mathcal{D}|)$ documents.

We now bound the number of queries that survive the second stage. First observe that if two queries do not have any wild card in the same position, then there is at most $1 \leq \beta$ document that matches both. Secondly, observe that for a fixed query q there are $\binom{\kappa}{s}\sigma^s\binom{m-\kappa}{s}$ other queries sharing $\kappa - s$ wild cards. We say these other queries are at distance s from q. For a fixed query,

we prove that with constant probability it survives the second stage. This is accomplished by considering each $s = 1, 2, \ldots, \kappa$ individually and using a high concentration bound on each, and then using a union bound. Since there are κ different values for s we bound the probability for each individual value by $\Theta(1/\kappa)$.

Now consider a pair of queries at distance s. The expected number of documents in their intersection is $\frac{t}{\sigma^s}$. Letting X to be the random variable indicating the number of documents in their intersection we get

$$\Pr[X > (1+\delta)\mu] = \Pr[X > \beta] = \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu < \left(\frac{t}{\sigma^s}\right)^{\Theta(\beta)}.$$

Recall that there are κ values for s and there are $\binom{\kappa}{s}\sigma^s\binom{m-\kappa}{s}$ "neighbours" at distance s, we want the following condition to hold:

$$\left(\frac{t}{\sigma^s}\right)^{\Theta(\beta)} \cdot \binom{\kappa}{s}\sigma^s\binom{m-\kappa}{s}\cdot\kappa \leq \frac{1}{100} \Leftrightarrow \sigma^{\Theta(s\beta)} \geq 100t^{\Theta(\beta)}\binom{\kappa}{s}\sigma^s\binom{m-\kappa}{s}\cdot\kappa$$

We immediately observe that the query time, t, should always be greater than m (just for reading the query) and that there are never more than $\kappa \leq m$ wild cards in a query. Picking $\sigma = t^{1+\varepsilon}$ for some $\varepsilon > 0$ and letting β be sufficiently large, we can disregard the factors t^β and σ^s. If $\sigma^{\Theta(s\beta)} > 100\kappa(\frac{e\kappa}{s})^s(\frac{m}{s})^s$ it follows that the condition above is satisfied. Since $\kappa \leq m \leq t \leq \sigma^{\frac{1}{1+\varepsilon}}$ it is sufficient to set $\beta = \Theta(1)$. We still have to derive the value of m. Since $t = D\frac{\sigma^\kappa}{\sigma^m} = D\frac{t^{(1+\varepsilon)\kappa}}{t^{(1+\varepsilon)m}}$, $t \cdot t^{(1+\varepsilon)(m-\kappa)} = D$. Manipulating this equation we see that $m = \kappa + \frac{\log_t D - 1}{1+\varepsilon}$. We can now use Chazelle's framework (Lemma 26): by construction, the output size of any query is $t = \Omega(Q(n))$ and the any two queries have $\mathcal{O}(1)$ documents in common. By the framework, we get the space lower bound of

$$S(n) = \Omega\left(\binom{m}{\kappa}\frac{\sigma^m}{\sigma^\kappa}D\frac{\sigma^\kappa}{\sigma^m}\right) = \Omega\left(\frac{n}{m}\binom{m}{\kappa}\right).$$

For $\kappa < \log_{Q(n)} n$, we can upper bound $\binom{m}{\kappa}$ by $\Theta(\log_{Q(n)} n)^\kappa$. Thus, we obtain the following theorem.

Theorem 14. *Any pointer-machine data structure that answers WCI queries with κ wild cards in time $Q(n) + \mathcal{O}(m+t)$ over an input of size n must use $\Omega\left(\frac{n}{\kappa}\Theta\left(\frac{\log_{Q(n)} n}{\kappa}\right)^{\kappa-1}\right)$ space, as long as $\kappa < \log_{Q(n)} n$, where t is the output size, and m is the pattern length.*

Since $m = \kappa + \frac{\log_t D - 1}{1+\varepsilon}$ we see that for $\kappa \leq \sqrt{\log n}$ we achieve $S(n) \geq \Omega\left(\frac{n}{\log n}(\frac{\log_t n}{\kappa})^\kappa\right)$. For query times close to $t = 2^\kappa$ we thus see $S(n) \geq \Omega\left(\frac{n}{\log n}(\frac{\log n}{\kappa^2})^\kappa\right)$. It is worth noting that this is close to the upper bounds seen in ....

4.8.3 Consequences

Our results when juxtaposed with the known upper bounds on WCI give us the following interesting observations. First, let us consider the case when $\kappa = 3\sqrt{\log n}$. Bille et al. [19] provided a special data structure, for when $m < \sigma^\kappa$, that uses $\mathcal{O}(n\sigma^{\kappa^2}(\log\log n)^\kappa)$ space and can answer queries in optimal time of $\mathcal{O}(m + j + t)$ where j is the number of wild cards in the query. Our query lower bound in Theorem 12 shows this is almost tight, for this particular range of κ: even if we are willing to pay a larger query time of $\mathcal{O}(2^{\kappa/2} + m + t)$, we would need $\Omega(n \cdot n^{\Theta(1/\log\kappa)}) = \Omega(n \cdot 2^{\Theta(\kappa^2/\log\kappa)})$ space. In other words, one can at best hope to remove the $(\log\log n)^\kappa$ factors, or reducing the base of the exponent from σ to 2.

Now let us consider the data structure of Cole et al. [47] that uses $\mathcal{O}(n(\log^\kappa n)/k!)$ space and can answer queries in $\mathcal{O}(m + 2^\kappa \log\log n + t)$ time (we assume all queries have κ wild cards). Our space lower bound shows that if we insist on query time of $\mathcal{O}(\log^{\mathcal{O}(1)} n + m + t)$, then our lower bound is within a $2^{\mathcal{O}(k)}(\log\log n)^k$ factor of their space usage. If we allow a higher query time, then the gap widens and it is possible that either our lower or their data structure can be improved (or both). Nonetheless, for $\kappa = 3\sqrt{\log n}$ and $Q(n) = 2^\kappa$, we can apply our space lower bound with fewer wild cards, specifically, with $\kappa' = \kappa^{1-\varepsilon}$ wild cards, for a constant $\varepsilon > 0$. This would yield a space lower bound of $\Omega(n \log^{\varepsilon\kappa^{1-\varepsilon}} n)$, which rules out the possibility of significantly improving the data structure of Cole et al. [47].

4.9 Two Patterns Reporting Lower Bound

Here we also use the framework of Chazelle and Rosenberg, specifically we use the refined lemma as stated in the previous section. To apply the framework we need to create a set of queries and a set of inputs satisfying the stated properties. First we only focus on the 2P problem and later describe how to adapt to the FP and 2FP problems. The rest of this section is divided into four parts. The first part is preliminaries and technicalities we use in the remaining parts. Next we describe how to create the documents, then we define the queries and finally we refine the two sets and prove that they satisfy the conditions stated in Lemma 26.

Preliminaries. Consider the alphabet $\Sigma = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \cdots, \mathbf{2^\sigma - 1}\}$ with 2^σ characters (we adopt the convention to represent the characters in bold). In our proof, documents and patterns are bitstrings. For convenience we use $\mathbf{i}$ to interchangeably refer to the character $\mathbf{i}$ and its binary encoding. The input is a set $\mathcal{D} = \{d_1, d_2, \ldots, d_D\}$ of D documents where each document is a string over Σ and the total length of the documents is n. The set $\mathcal{D}$ is to be preprocessed such that given two patterns P_1 and P_2, that are also strings over Σ, all documents where both P_1 and P_2 occur can be reported. Our main theorem is the following.

Theorem 15. *Any data structure on the Pointer Machine for the Two Pattern Query Problem with query time $Q(n)$ and space usage $S(n)$ must obey*

$$S(n)Q(n) = \Omega\left(n^{2-o(1)}\right)$$

Also, if query time is $\mathcal{O}((nk)^{1/2-\alpha} + k)$ for a constant $0 < \alpha < 1/2$, then

$$S(n) = \Omega\left(n^{\frac{1+6\alpha}{1+2\alpha}-o(1)}\right)$$

The Documents. Let σ be some parameter to be chosen later. In our construction, the first character, $\mathbf{0}$, works as a delimiter and for convenience (and to avoid confusion) the symbol # to denotes it. The number of documents created is D which is also a parameter to be chosen later. The set of documents is created randomly as follows. Each document will have $3(|\Sigma| - 1)$ characters, in $|\Sigma| - 1$ consecutive parts of three characters each. The i-th part, $1 \leq i \leq 2^\sigma - 1$, is $\#\mathbf{i}b_1b_2\cdots b_\sigma$ where each $b_j, 1 \leq j \leq \sigma$ is uniformly and independently set to "0" or "1". In other words, the i-th part is the encoding of the delimiter (which is basically σ "0"s) followed by the encoding of $\mathbf{i}$, followed by σ random "0"s and "1"s.

The Queries. The patterns in our queries always starts with #, followed by another character $\mathbf{i}$, $1 \leq \mathbf{i} < 2^\sigma$ (called the *initial* character), followed

by some trailing bits. Observe that any pattern $P_1 = \#\mathbf{i}$, for $1 \leq \mathbf{i} < 2^\sigma$, matches all the documents. Observe also, if two patterns $P_1 = \#\mathbf{i}b_1 \cdots b_p$ and $P_2 = \#\mathbf{i}b'_1 \cdots b'_p$ where $b_j, b'_j \in \{0,1\}, 1 \leq j \leq p$, match the same document, then we must have $b_j = b'_j$ for every $1 \leq j \leq p$. Based on this observation, our set of queries, $\mathcal{Q}$, is all pairs of patterns (P_1, P_2) with different initial characters and p trailing bits each, for some parameter p to be set later. The previous observation is stated below.

Observation 1. *Consider two patterns P and P' that have the same initial characters. If a document matches both P and P', then $P = P'$ (i.e., they also have the same trailing bits).*

A consequence of our construction is that, each pattern has one position where it can possibly match a document and matching only depends on the random bits after the initial character.

Analysis. We start by counting the number of queries. For each pattern in a query we have one initial character and p trailing bits but the two initial characters should be different. So the number of queries is

$$|\mathcal{Q}| = \binom{2^\sigma - 1}{2} 2^{2p} = \Theta(2^{2\sigma+2p})$$

We need to obey the properties stated in Lemma 26 in the worst case, i.e. the intersection of l queries should be $\leq \beta$. The analysis proceeds by using high concentration bounds and picking parameters carefully such that the requirements of Lemma 26 are satisfied with high probability. First we study the probability of fixed patterns or fixed queries matching random documents.

Lemma 27. *The probability that a fixed pattern matches a random document is 2^{-p} and the probability that a fixed query matches a random document is 2^{-2p}.*

Proof. The first part follows by the construction. Two patterns with distinct initial characters matching a document are independent events, thus the second part also follows. □

Corollary 4. *The expected number of documents matching a fixed query is $\Theta\left(D2^{-2p}\right)$*

According the framework, every query should have a large output (first requirement in Lemma 26). By a straightforward application of Chernoff bounds it follows that all queries output at least a constant fraction of the expected value. We also need to satisfy the second requirement in Lemma 26. Consider the intersection of ℓ^2 queries, $q_1, \cdots, q_{\ell^2}$, for some parameter ℓ. Assume the intersection of $q_1, \cdots, q_{\ell^2}$ is not empty, since the second requirement is already

satisfied otherwise. There must be at least ℓ distinct patterns among these queries thus it follows by Observation 1 that there are at least ℓ distinct initial characters among the ℓ^2 queries.

Observation 2. *The number of sets that contain ℓ patterns, $P_1, \cdots, P_\ell$, with distinct initial characters is at most $(2^{\sigma+p})^\ell$.*

Lemma 28. *The probability that a random document satisfies a set of ℓ patterns, $P_1, \cdots, P_\ell$ is at most $\frac{1}{2^{p\ell}}$, and the expected number of such documents is at most $\frac{D}{2^{p\ell}}$.*

Proof. This is just an extension of the case where $\ell = 2$. If the patterns do not have distinct initial characters then the probability is 0 by Observation 1. Otherwise, the events for each constraint matching a document are mutually independent. □

We will choose parameters such that $\frac{D}{2^{p\ell}} = \mathcal{O}(1)$. We need to consider the intersection of ℓ^2 queries and bound the size of their intersection.

Lemma 29. *The probability that at least β documents satisfy ℓ given patterns is $O\left(\left(\frac{eD}{\beta 2^{p\ell}}\right)^\beta\right)$.*

Proof. Let X_i be a random variable that is 1 if the i-th document matches the given ℓ patterns and 0 otherwise, and let $X = \sum_{i=1}^{D} X_i$. By Lemma 28 we have $\mathbb{E}[X] = \frac{D}{2^{p\ell}}$. By the Chernoff bound $\Pr[X \geq \beta] = \Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \frac{e^\beta}{(\beta/\mu)^\beta}$ where $\mu = D/2^{p\ell}$ and $1 + \delta = \beta/\mu$. The lemma follows easily afterwards. □

We now want to apply the union bound and use Lemma 29 and Observation 2. To satisfy the second requirement of Lemma 26, it suffices to have

$$2^{\ell(p+\sigma)} O\left(\left(\frac{eD}{\beta 2^{p\ell}}\right)^\beta\right) < 1/3. \tag{4.3}$$

By Lemma 26 we have $S(n) \geq \frac{|\mathcal{Q}|t}{\ell^2 2^{\mathcal{O}(\beta)}}$. Remember that by Corollary 4, the output size of each query is $t = \Omega(D2^{-2p})$. We set D such that $t = \Theta(Q(n))$. We now plug in t and $|\mathcal{Q}|$.

$$S(n) \geq \frac{|\mathcal{Q}|t}{\ell^2 2^{\mathcal{O}(\beta)}} = \Omega\left(\frac{2^{2\sigma+2p} D 2^{-2p}}{\ell^2 2^{\mathcal{O}(\beta)}}\right) = \Omega\left(\frac{(n/D)^2 D}{\ell 2^{\mathcal{O}(\beta)}}\right) = \Omega\left(\frac{n^2}{Q(n) 2^{2p} \ell^2 2^{\mathcal{O}(\beta)}}\right)$$

Since $D2^\sigma = n$ (the input size must be n) and $D = \Theta(Q(n)2^{2p})$. This gives us that $S(n)Q(n) = \Omega\left(\frac{n^2}{\ell^2 2^{2p} 2^{\mathcal{O}(\beta)}}\right)$, subject to satisfying inequality (4.3). Choosing $p = \Theta(\beta) = \Theta(\sqrt{\log(n/Q(n))})$ and $\ell = \Theta(\log n)$ we satisfy the condition and get the trade-off: $S(n)Q(n) = \Omega\left(\frac{n^2}{2^{\mathcal{O}(\sqrt{\log(n/Q(n))})} \log^2 n}\right)$

Though we do not explicitly use σ in the bound, one can verify that $\sigma = \Theta(\log(n/Q(n)))$ and thus we can assume that each character fits in one memory cell.

Recently in the literature there has been bounds on the form $\mathcal{O}(\sqrt{nk}\log^{\mathcal{O}(1)} n + k)$ with linear space. The trade-off proved here can be used to prove that is optimal within polylogarithmic factors. Suppose $Q(n) = \mathcal{O}((nk)^{1/2-\alpha} + k)$, for a constant $0 < \alpha < 1/2$. We can obtain a lower bound, by making sure that the search cost $nk^{1/2-\alpha}$ is dominated by the output size which means $k \geq (nk)^{1/2-\alpha}$ or $k \geq n^{(1/2-\alpha)/(1/2+\alpha)} = n^{(1-2\alpha)/(1+2\alpha)}$. Plugging this in for the query time gives the trade-off $S(n) \geq \Omega(n^{\frac{1+6\alpha}{1+2\alpha}}/2^{\Theta(\sqrt{\log(n)})})$.

We have now proven the claims of Theorem 15.

4.9.1 Forbidden Pattern lower bound modifications

One quickly sees that the inputs and queries designed for the 2P problem do not work to prove lower bounds for the Forbidden Pattern case (i.e. one positive pattern and one negative pattern). To overcome this, we design a slightly different input instance and the bounds are slightly different though still not more than polylogarithmic factors off from the upper bounds.

The documents are created very similar to before, except each document now comes in two parts. The first part is the same as before, i.e. 2^σ subsequences of 3 characters where the i-th subsequence is #, follow by **i**, followed by a trailing bits. Now we extend the alphabet to $\Sigma = \Sigma_1 \cup \Sigma_2 = [\mathbf{2^{\sigma+1}}]$, and the second part of the document will only contain symbols from $\Sigma_2 = \{\mathbf{2^\sigma}, \mathbf{2^\sigma + 1}, \ldots, \mathbf{2^{2\sigma} - 1}\}$. The second part of a document is a uniformly random subset with m characters from Σ_2, however, as before, we prefix each character with #. The ordering of the characters in the second part does not matter. The queries consist of a positive pattern and a negative pattern. The positive patterns are created in the same way as before. The negative pattern is just # follow by one character from Σ_2.

Fact. *There are $2^{2\sigma+p}$ different queries.*

Proof. A simple counting argument. □

Fact. *The probability a fixed query hits a random document is $2^{-p}(1 - \frac{m}{|\Sigma_2|})$ and the expected number of documents returned by a query is $\frac{D}{2^p}(1 - \frac{m}{|\Sigma_2|})$*

Proof. The proof follows easily from the construction. □

For ℓ^2 queries to have any output, there must either be ℓ distinct positive patterns or ℓ distinct negative patterns. In the former case, the rest of the proof from Section 4.9 goes through. In latter case there are ℓ distinct negative patterns, i.e. ℓ distinct characters. If any of the ℓ characters appear in a document, that document is not in the intersection.

Fact. *Let P be a set of ℓ negative patterns. The probability that a random document contains at least one pattern from P is $\frac{\binom{|\Sigma_2|-\ell}{m}}{\binom{|\Sigma_2|}{m}} = \Theta(\frac{1}{2^{pl}})$, for some choice of m.*

Proof. Suppose we choose $1 - \frac{m}{\Sigma_2}$ to be 2^{-p}, then it follows that $\Sigma_2 - m = \frac{\Sigma_2}{2^p}$. Using these equations it follows that

$$\frac{\binom{|\Sigma_2|-\ell}{m}}{\binom{|\Sigma_2|}{m}} = \frac{(\Sigma_2 - m)(\Sigma_2 - m - 1)\cdots(\Sigma_2 - m - \ell + 1)}{\Sigma_2(\Sigma_2 - 1)\cdots(\Sigma_2 - \ell + 1)} \leq \frac{1}{2^{pl}}$$

□

Now one can go through the remaining steps from Lemma 28 and onwards to get the same results. However the values change slightly. We see now $n = D(m + 2^\sigma)$, which is still $\Theta(D2^\sigma)$, since m is bounded by 2^σ. As noted above the number of queries has changed to $|Q| = 2^{2\sigma+p}$. We get the following when plugging in the values in the framework.

$$S(n) \geq \frac{|Q|t}{\ell^2 2^{\mathcal{O}(\beta)}} = \frac{2^{2\sigma+p}D2^{-2p}}{\ell^2 2^{\mathcal{O}(\beta)}} = \frac{(n/D)^2 D}{\ell^2 2^{\mathcal{O}(\beta)} 2^p} = \frac{n^2}{Q(n)\ell^2 2^{\mathcal{O}(\beta)} 2^{3p}}$$

By choosing the same values for β, p, and ℓ as before, we get the same trade-off up to constant factors, the only difference is the denominator has 2^{3p} factor instead of 2^{2p}.

Adapting to two negative patterns. The description above defines the hard instance for one positive pattern and one negative pattern. The change was to split each document in two parts, one regarding the positive element and one regarding the negative element. In exactly the same way a hard instance can be designed for two negative patterns: simply use the negative construction for the both parts of the documents. There are two minor details that also need to be addressed in the case of two negative patterns. The first is, that the length of each document is "long enough", which easily follows since m is at least a constant fraction of 2^σ. The second part, as also noted above, is that the queries lose yet another p bits, which means we can now only create $2^{2\sigma}$ queries. Similarly to above, this means the denominator will have a 2^{4p} factor rather than 2^{3p} (one positive pattern, one negative pattern) or 2^{2p} (two positive patterns).

4.10 Two Patterns Semi-Group Lower Bound

Following the same strategy as in the previous section and with some more work we can get the following theorem:

Theorem 16. *Answering 2P queries in the semi-group model requires*

$$S(n)Q^2(n) = \Omega(n^2/\log^4 n).$$

The general strategy for proving the lower bound is as follows. We create queries and documents where the answer for each query is sum of "a lot of" documents. Then we aim to prove that any subset of $\text{poly}(\log n)$ documents are unlikely to have more than a few patterns in common, which subsequently implies, any pre-computed sum that involves more than polylogarithmic weights is only useful for few queries; let's call such a pre-computed sum "crowded". We charge one unit of space for every "crowded" sum stored. At the query time however, since the answer to queries involves adding up "a lot of" weights, to reduce the query time, the query algorithm must use at least one "crowded" sum. But we argued that each "crowded" sum can be useful only for a small number of queries which means to be able to answer all the queries, the data structure should store many "crowded" sums, giving us a lower bound.

This strategy is very similar to the strategy employed by Dietz el al. [50] to prove a lower bound for the offline set intersection problem of processing a sequence of updates and queries. However we need to cast the construction in terms of documents and two-pattern queries. The results we get are for static data structure problems where as they care about an algorithm for processing a sequence of queries/updates. This means the results are not easily comparable since there are other parameters, such as the number of queries and updates. They do also provide a lower bound for dynamic online set intersections with restrictions on space. However, since it is dynamic the results are also not directly comparable. Both their result and our result suffer the penalty of some $\text{poly}\log(n)$ factors, but ours are slightly more sensitive. Their log factors are only dependent on n (the number of updates, which for us is the number of elements) where as some of our log factors depend on $Q(n)$ instead (the query time), which makes a difference for fast query times. However when considering the pointer machine model we have to be much more careful than they are in considering the size of the intersection of many sets. For the pointer machine it is absolutely crucial that the size is bounded by a constant, whereas they can get away with only bounding it by $\mathcal{O}(\log n)$ and only suffer $\log n$ factor in the bounds. If the intersection cannot be bounded better than $\varepsilon \log n$ for pointer machine lower bounds, the bound gets a penalty of n^{ε}, which is really detrimental and would make the lower bounds almost non-interesting. Luckily we are able to provide a better and tighter analysis in this regard.

Documents and Queries. Our construction will be very similar to the one used in the reporting case and the only difference is that the number of trailing bits, p will be constant. To avoid confusion and for the clarity of the exposition, we will still use the variable p.

Lower bound proof We now wish to bound how many documents can be put in a "crowded" sum and be useful to answer queries. To be specific, we define a "crowded" sum, as a linear combination of the weights of at least β documents. We would like to show that such a combination is useful for a very small number of queries (i.e., patterns). The same technique as the previous section works and we can in fact continue from inequality (4.3).

We make sure that $2^{p\ell} \geq D^2$ or $2\log D \leq p\ell$. We also choose $\beta > ce$ for some $c > 1$ and simplify inequality 4.3 to get, $2^{\ell(p+\sigma)} \leq 2^{p\ell\beta/2} \Rightarrow p + \sigma \leq p\beta/2$, or

$$\sigma \leq p\beta/2. \tag{4.4}$$

If the above inequality holds, then, no sets of β documents can match ℓ different patterns. Furthermore, remember that with high probability the number of documents that match every query is $\frac{D}{2\cdot 2^{2p}}$. Observe that to answer a query q faster than $\frac{D}{2\cdot 2^{2p}\beta}$ time, one has to use at least one crowded sum s. If there is a document that is used in s but it does not match q, then the result will be incorrect as in the semi-group model there is no way to "subtract" the extra non-matching weight from s. Thus, all the documents whose weights are used in s must match q. However, we just proved that documents in a crowded sum can only match ℓ patterns, and thus be useful for ℓ^2 queries. So, if inequality (4.4) holds, either $Q(n) \geq \frac{D}{2\cdot 2^{2p}\beta}$ or $S(n) \geq \frac{2^{2\sigma+2p}}{\ell^2}$; the latter follows since $2^{2\sigma+2p}$ is the total number of queries and the denominator is the maximum number of queries where a crowded sum can be used for. To exclude the first possibility, we pick D such that

$$Q(n) < \frac{D}{2\cdot 2^{2p}\beta}. \tag{4.5}$$

Recall that $n = D2^{\sigma}$. We pick $\sigma = \log(n/D)$, $p = \mathcal{O}(1)$, which forces $\beta \geq \log(n/D))$ and $\ell \geq \log D = \Theta(\log(n/2^{\sigma})) = \Theta(\log Q(n))$. We also pick D as small as possible while satisfying (4.5). This leads to the following trade-off:

$$S(n) \geq \frac{2^{2\sigma+2p}}{\ell^2} = \frac{n^2/D^2}{\Theta(\log^2 Q(n))} = \frac{n^2}{\Theta(Q^2(n)\log^2 n \log^2 Q(n))} \Rightarrow$$

This yields $S(n)Q(n)^2 = \Omega(n^2/\log^4 n)$ and concludes the proof of Theorem 16.

4.11 Lower Bound Implications

Set Intersection. We refer to the introduction for the definition of this problem (and its variants). Using a data structure that solves the set intersection problem, one can solve the hard instance we gave for the 2P problem: Notice that all our patterns have a fixed length. This means in a suffix tree built on the set of documents, each pattern matches a set of leaves disjoint from all the other patterns. For each pattern one identifies the set of documents in the corresponding leaf set as the input for the Set Intersection problem. Verifying the correctness is rather straightforward.

Corollary 5. *The reporting (resp. searching) variant of the set intersection requires* $S(n)Q(n) = \Omega(n^2/(2^{\mathcal{O}(\sqrt{\log(n/Q(n))}}\log^2 n))$ *(resp.* $S(n)Q^2(n) = \Omega(n^2/\log^4 n)$*) in the pointer machine (resp. semi-group) model.*

Furthermore these trade-offs are actually achievable within polylogarithmic factors. In fact in addition to Set Intersection all the Two-Pattern trade-offs are also achievable.

Set Intersection Upper bound Suppose we are aiming for query time $Q(n)$. We store every set in sorted order and this obviously takes only linear space. The observation is that if a set S_i has at most $Q(n)/\log n$ elements, then we can answer the query i,j in $\mathcal{O}(Q(n))$ time by binary searching every element of S_i in S_j. Thus, we only need to focus on "large" sets, that is, those larger than $Q(n)/\log n$. There can at most be $t = n\log n/Q(n)$ large sets. Directly store the answer for every pair of large sets. In the searching variant (semi-group model), this takes t^2 space. For the reporting variant, assume the sets are indexed increasingly by size. The space usage is

$$\sum_{i<j}\min\{|S_i|,|S_j|\} \leq \sum_{i=0}^{t-1}(t-i)|S_{i+1}| \leq \frac{Q(n)}{\log n}O\left(\left(\frac{n\log n}{Q(n)}\right)^2\right) = O\left(\frac{n^2\log n}{Q(n)}\right).$$

Theorem 17. *The reporting (resp. searching) variant of the set intersection can be solved in* $\mathcal{O}(Q(n)+k)$ *(resp.* $Q(n)$*) time using* $\mathcal{O}(n^2\log^2 n/Q^2(n))$ *(resp.* $\mathcal{O}(n^2\log n/Q(n))$*) space.*

Two-Sided High-Dimensional Queries Another implication is for high dimensional two-sided range queries, a special case of the fundamental problem of orthogonal range searching (see [4, 5, 35, 38, 39, 41, 42] for the best upper bounds, lower bounds, history and motivation). In this problem, we would like to preprocess a set of points in d-dimensional space such that given an axis-aligned query rectangle $q = (a_1, b_1) \times \cdots \times (a_d, b_d)$, we can report (or count) the points contained in the query. We consider a variant of the problem in

which the input is in d-dimensions but the query box q has two sides, that is, all a_i's and b_i's are set to $-\infty$ and $+\infty$, respectively, except only for two possible values.

To give a lower bound for this problem, we consider our construction for the 2P problem, and a suffix tree built on the set of documents. Consider a non-delimiter character $\mathbf{i}$ and its corresponding range of leaves, $S_{\mathbf{i}}$, in the suffix tree: we are guaranteed that each document appears exactly once in $S_{\mathbf{i}}$. We create D points in $(2^\sigma - 1)$-dimensional space by picking the i-th coordinate of the points based on the order of the documents in $S_{\mathbf{i}}$. With a moment of thought, it can be verified that we can use a solution for 2-sided orthogonal range searching to solve our hard instance for the 2P problem. By reworking the parameters we obtain the following.

Theorem 18. *For a set of m points in d-dimensional space, answering 2-sided orthogonal range reporting (resp. searching) queries in $m^{1-o(1)} + \mathcal{O}(k)$, (resp. $m^{1-o(1)}$) time in the pointer machine (resp. semi-group) model requires $\Omega((md)^{2-o(1)}/m)$ (resp. $\Omega((md)^{2-o(1)}/m^2)$) space.*

Two Pattern Simple Upper Bounds

For the sake of completeness we also include the simple upper bounds for 2P, FP, and 2FP that match the trade-offs from the lower bounds within polylogarithmic factors. These upper bounds have already been described by other authors and we claim no novelty here. The only observation left to make, is that the data structures for the counting versions also work with arbitrary weight assignments.

We first describe the general scheme and then move on to describe how to adapt to reporting versus counting. The first part deals with the reporting case and the second part deals with the searching/counting variant. In the second part we shortly describe upper bounds that work in the semi-group model for both the 2P and 2FP problems. Due to working with the semi-group model we are not allowed to do subtractions. We show that the problem of finding all documents where both P_1 and P_2 occur and the problem of finding documents where P_1 occurs but P_2 does not occur can still both be solved in such a model. Also notice that the solution described will work for any assignment of weights, which is also assumed for the lower bound to hold.

General Scheme As input we have a set of documents $D = \{d_1, d_2, \ldots, d_D\}$. We build a suffix tree over the concatenation of the documents, and each leaf is annotated with which document that suffix starts in (and the weight if applicable) of that document. Let ℓ_i be the i-th leaf and let the document it corresponds to be d_{ℓ_i}. For each $i \in \{1, \ldots, D\}$ we build a range emptiness structure, where the input to the ith structure is $\{j \mid d_{\ell_j} = i\}$. That is, for every document we build a range emptiness structure where the points are the

leaf numbers in the suffix tree corresponding to that document. Now suppose we are aiming for a query time of $\mathcal{O}(Q(n))$. We mark every $Q(n)$-th leaf and mark all their common ancestors. This leads to $\mathcal{O}(n/Q(n))$ marked nodes in the suffix tree. Now depending on the situation (reporting, searching, forbidden pattern, etc.) we have a different strategy. E.g. in the searching cases we store some information for every pair of marked nodes that depends on the query type we are to support.

Two matching patterns - Reporting The lower bound says that $S(n)Q(n) = \Omega(n^2/\log^{\mathcal{O}(1)} n)$, thus we can store $\mathcal{O}(n)$ information for all marked nodes in our tree and still be within polylogarithmic factors of the optimal solution.

For each node in the suffix tree there is a unique highest descendant that is marked. Each node then stores a pointer to that node. If the node itself is marked a self pointer is stored. Each node defines a range of leaves and each leaf is associated with a document. For each marked node we store a pointer to an auxiliary data structure. The auxiliary data structure is the standard document retrieval data structure from [96]. The input for the auxiliary data structure is the set of documents corresponding to the leaves defined by the marked node. To process a query one searches for P_1 in the main structure which ends at a node v. Then we follow the pointer stored at v to its highest marked descendant v_{mark} and perform a query with P_2 in the auxiliary data structure stored at v_{mark}. The leaves in the range defined by v_{mark} is by definition a subset of the ones defined by v. This means that what is reported by the query in the auxiliary structure should be reported, since those documents contain both P_1 and P_2. But we might be missing some documents. To find the remaining documents a search for P_2 in the main structure is performed which ends in the node u. Now scan the leaves in v's range but not in v_{mark}'s range. Each leaf corresponds to a document and if that document has not already been reported, a range emptiness query is performed asking if that document appears in u's range. If so, the document is reported. The documents reported during the scan exactly satisfy that P_1 occurs in them (due to scanning leaves defined by v) and they also appear as leaf in u's range, meaning P_2 occurs in them. This concludes the description of the data structure and query algorithm.

The time to perform a query is $\mathcal{O}(|P_1| + |P_2| + Q(n) + k)$ where k is the output size. To bound the space usage we see that each auxiliary data structure uses $\mathcal{O}(n)$ space and there are $\mathcal{O}(\frac{n}{Q(n)})$ marked nodes. In addition we also store range emptiness structures which requires in total $\mathcal{O}(n)$ space as well as a suffix tree using $\mathcal{O}(n)$ space. In total we get $\mathcal{O}(n^2/Q(n))$ space, since $Q(n) \leq n$.

Two matching patterns - Counting Observe that a node in the suffix tree corresponds to an interval of leaves. Each pair of marked nodes then

corresponds to two intervals of leaves. The value stored for a pair of marked nodes is then the weighted sum of the documents that occur in both intervals. Now to answer a query the following steps are performed. First we search for each pattern, which gives two nodes that have two intervals $[l_{p_1}, r_{p_1}]$ and $[l_{p_2}, r_{p_2}]$. For those two nodes we find the highest marked descendant in their subtrees (say we store a pointer in every node to its highest marked descendant). This will also give us two intervals: $[l_{mark_{p_1}}, r_{mark_{p_1}}]$, $[l_{mark_{p_2}}, r_{mark_{p_2}}]$. For this pair a value is stored, which we read and let that be the *running sum*. Now this only corresponds to *parts* of the actual intervals we were interested in, since by construction $l_{p_i} \leq l_{mark_{p_i}}$ and $r_{p_i} \geq r_{mark_{p_i}}$. To find the remaining values to add, one can scan through the remaining leaves and for each leaf perform $\mathcal{O}(1)$ range emptiness queries to test whether that value is already part of the running sum and otherwise add it. I.e. run through the leaves with numbers in $[l_{p_1}, r_{p_1}] \setminus [l_{mark_{p_1}}, r_{mark_{p_1}}]$ and in $[l_{p_2}, r_{p_2}] \setminus [l_{mark_{p_2}}, r_{mark_{p_2}}]$.

Since every $n^{\varepsilon/2}$th leaf was marked it follows that $l_{mark_{p_i}} - l_{p_i} = \mathcal{O}(n^{\varepsilon/2})$ and similarly $r_{p_i} - r_{mark_{p_i}} = \mathcal{O}(n^{\varepsilon/2})$ which are the leaf numbers we scan. The query time becomes $\mathcal{O}(P_1 + P_2 + n^{\varepsilon/2})$ and the space becomes $\mathcal{O}((n/n^{\varepsilon/2})^2) = \mathcal{O}(n^{2-\varepsilon})$. Note that this matches the trade-off in the lower bound within a $poly(\log n)$ factor.

One forbidden pattern - Counting In this setting we are given two patterns P^+ and P^- and we are to report the sum of all documents where P^+ occurs but P^- does not occur. In the model where we proved the lower bound trade-off we were not allowed to subtract weights. This might seem like a hard restriction and our main observation is that subtractions are not required. Suppose for a moment subtractions were allowed, then the solution presented in Section 4.11 would also solve this version of the problem, except when scanning the leaves one has to subtract from the running sum rather than add. To fix this, another pointer is stored in every node that points to the nearest ancestor which is a marked node. For the query, the node that P^- ends in, one should follow the ancestor pointer instead. Now "too much" have been excluded by the ancestor of P^- from the running sum, and by iterating through the interval differences (similar to the previous case) one can find the weights that were excluded and add them to running sum. The analysis is completely analogous to before.

Bibliography

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 434–443, 2014. 52, 55

[2] Peyman Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Symposium on Computational Geometry (SoCG)*, pages 339–346, 2012. 64, 89, 91

[3] Peyman Afshani and Jesper Sindahl Nielsen. Data structure lower bounds for set intersection and document indexing problems. 2015. 49, 50

[4] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010. 64, 101

[5] Peyman Afshani, Lars Arge, and Kasper Green Larsen. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In *Symposium on Computational Geometry (SoCG)*, pages 323–332, 2012. 64, 101

[6] Peyman Afshani, Edvin Berglin, Ingo van Duijn, and Jesper Sindahl Nielsen. Applications of incidence bounds in point covering problems. 2015. vi

[7] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $\mathcal{O}(n \log n)$ sorting network. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1983. 45

[8] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997. 45, 46

[9] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001. 63

[10] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences (JCSS)*, 57:74–93, 1998. 36, 39, 40, 41

[11] Sunil Arya, David M. Mount, and Jian Xia. Tight lower bounds for halfspace range searching. *Discrete & Computational Geometry*, 47(4): 711–730, 2012. 65

[12] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011. 50, 51

[13] Nikhil Bansal and Ryan Williams. Regularity lemmas and combinatorial algorithms. *Theory of Computing*, 8(1):69–94, 2012. 83

[14] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 748–759, 2011. 58, 61, 75

[15] Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013. 58

[16] Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected linear time sorting for word size $\Omega(\log^2 n \log\log n)$. In *Scandinavian Workshop on Algorithm Theory (SWAT), Proceedings*, pages 26–37, 2014. 35

[17] Gary Benson and Michael S. Waterman. A fast method for fast database search for all k-nucleotide repeats. *Nucleic Acids Research*, 22(22), 1994. 51

[18] Philip Bille, Anne Pagh, and Rasmus Pagh. Fast evaluation of union-intersection expressions. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 4835 of *Lecture Notes in Computer Science*, pages 739–750. Springer Berlin Heidelberg, 2007. 55

[19] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. In *Scandinavian Workshop on Algorithm Theory (SWAT), Proceedings*, pages 283–294, 2012. 55, 57, 60, 93

[20] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, and Avi Wigderson. A tradeoff between search and update time for the implicit dictionary problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 226 of *LNCS*, pages 50–59. Springer, 1986. 7

[21] Gerth Stølting Brodal. A survey on priority queues. In *Proc. Conference on Space Efficient Data Structures, Streams and Algorithms – Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer Verlag, Berlin, 2013. 19

[22] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002. 22

[23] Gerth Stølting Brodal, Casper Kejlberg-Rasmussen, and Jakob Truelsen. A cache-oblivious implicit dictionary with the working set property. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 6507 of *LNCS*, pages 37–48. Springer, 2010. 9, 15

[24] Gerth Stølting Brodal, Pooya Davoodi, and S. Srinivasa Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 63(4):815–830, 2012. 62, 82

[25] Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Finger search in the implicit model. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 527–536, 2012. 7

[26] Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Strictly implicit priority queues: On the number of moves and worst-case time. In *Algorithms and Data Structures Workshop (WADS)*, 2015. 7

[27] Gerth Stølting Brodal. Finger search trees. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 11. CRC Press, 2005. 9

[28] Andreas Broschart and Ralf Schenkel. Index tuning for efficient proximity-enhanced query processing. In *Workshop of the Initiative for the Evaluation of XML Retrieval (INEX)*, pages 213–217, 2009. 51

[29] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010. 50, 51

[30] Svante Carlsson and Mikael Sundström. Linear-time in-place selection in less than 3n comparisons. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 244–253, 1995. 24

[31] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Scandinavian Workshop*

on Algorithm Theory (SWAT), Proceedings, pages 1–13, 1988. 19, 20, 28

[32] Timothy M. Chan. Optimal partition trees. In *Symposium on Computational Geometry (SoCG)*, pages 1–10. ACM, 2010. 64

[33] Timothy M. Chan and Moshe Lewenstein. Clustered integer 3sum via additive combinatorics. http://arxiv.org/abs/1502.05204, 2015. 55

[34] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011. 83

[35] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011. 101

[36] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. In *Proceedings of Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2012. 83

[37] Timothy M. Chan, Stephane Durocher, Matthew Skala, and Bryan T. Wilkinson. Linear-space data structures for range minority query in arrays. In *Scandinavian Workshop on Algorithm Theory (SWAT), Proceedings*, pages 295–306, 2012. 83

[38] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, 1986. 101

[39] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3): 427–462, 1988. 101

[40] Bernard Chazelle. Lower bounds on the complexity of polytope range searching. *Journal of the American Mathematical Society*, 2(4):pp. 637–666, 1989. 65

[41] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990. 63, 64, 101

[42] Bernard Chazelle. Lower bounds for orthogonal range searching II. the arithmetic model. *Journal of the ACM (JACM)*, 37(3):439–463, 1990. 59, 64, 65, 101

[43] Bernard Chazelle and Burton Rosenberg. Simplex range reporting on a pointer machine. *Computational Geometry*, 5:237–247, 1995. 64, 90

[44] Bernard Chazelle, Micha Sharir, and Emo Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8:407–429, December 1992. 64

[45] Hagai Cohen and Ely Porat. On the hardness of distance oracle for sparse graph. http://arxiv.org/abs/1006.1117, 2010. 52, 55

[46] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010. 52, 53, 54, 55, 57

[47] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 2004. 55, 57, 93

[48] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw Hill, 3rd edition, 2009. 35

[49] Mark de Berg, Marc van Kreveld, Mark Overmars, and O. Schwarzkopf. *Computational Geometry — Algorithms and Applications*. Springer, 3rd edition, 2008. 63

[50] Paul F. Dietz, Kurt Mehlhorn, Rajeev Raman, and Christian Uhrig. Lower bounds for set intersection queries. *Algorithmica*, 14(2):154–168, 1995. 55, 99

[51] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. 40, 45

[52] Stefan Edelkamp, Amr Elmasry, and Jyrki Katajainen. Ultimate binary heaps, 2013. Manuscript. 20

[53] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280, 1999. 37, 73

[54] Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. *Journal of Computer and System Sciences (JCSS)*, 66(4):763–774, 2003. Special Issue on PODS 2001. 51, 54, 57, 80

[55] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):art. 20, 2007. 69

[56] Johannes Fischer, Travis Gagie, Tsvi Kopelowitz, Moshe Lewenstein, Veli Mäkinen, Leena Salmela, and Niko Välimäki. Forbidden patterns. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 327–337, 2012. 51, 53, 54, 57

[57] Gianni Franceschini. Sorting stably, in place, with $O(n \log n)$ comparisons and $O(n)$ moves. *Theory of Computing Systems*, 40(4):327–353, 2007. 19

[58] Gianni Franceschini and Roberto Grossi. Optimal worst case operations for implicit cache-oblivious search trees. In *Algorithms and Data Structures Workshop (WADS)*, volume 2748 of *LNCS*, pages 114–126. Springer, 2003. 7, 8, 9, 10

[59] Gianni Franceschini and J. Ian Munro. Implicit dictionaries with $O(1)$ modifications per update and fast search. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 404–413, 2006. 19, 20, 21

[60] Gianni Franceschini, Roberto Grossi, James Ian Munro, and Linda Pagli. Implicit B-Trees: New results for the dictionary problem. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 145–154. IEEE, 2002. 8, 9

[61] Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM (JACM)*, 30(1):80–94, 1983. 7

[62] Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012. 58

[63] François Le Gall. Powers of tensors and fast matrix multiplication. *CoRR*, abs/1401.7714, 2014. 83

[64] Michael T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *Journal of the ACM (JACM)*, 58(6):27, 2011. 36, 46

[65] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $\mathcal{O}(n \log n)$ time. *CoRR*, abs/1403.2777, 2014. 45

[66] Frank Gray. Pulse code communications, 1953. 26

[67] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997. 51

[68] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 366–398, 1998. 45

[69] Yijie Han and Mikkel Thorup. Integer sorting in $\mathcal{O}(n\sqrt{\log\log n})$ expected time and linear space. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002. 36, 47

[70] Wing-Kai Hon, Rahul Shah, and Jeffrey S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 713–722, 2009. 52, 58, 71, 75

[71] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. String retrieval for multi-pattern queries. In *International Symposium on String Processing and Information Retrieval*, pages 55–66, 2010. 53, 57

[72] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey S. Vitter. On position restricted substring searching in succinct space. *Journal of Discrete Algorithms*, 17:109–114, 2012. 66, 76

[73] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Document listing for queries with excluded pattern. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 185–195, 2012. 53, 57

[74] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey S. Vitter. Faster compressed top-k document retrieval. In *Proc. 23rd DCC*, pages 341–350, 2013. 53, 71

[75] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey S. Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM (JACM)*, 61(2):9, 2014. 52

[76] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM (JACM)*, 61(2):9, 2014. 53

[77] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989. 62

[78] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977. 19

[79] Bolette Ammitzbøll Jurik and Jesper Sindahl Nielsen. Audio quality assurance: An application of cross correlation. In *International Conference on Preservation of Digital Objects (iPRES)*, pages 196–201, 2012.

[80] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Word-packing algorithms for dynamic connectivity and dynamic sets. http://arxiv.org/abs/1407.6755, 2014. 55

[81] David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983. 36

[82] Donald E. Knuth. *The Art of Computer Programming*, volume 4A: Combinatorial Algorithms. Addison-Wesley Professional, 2011. 39

[83] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. 3SUM hardness in (dynamic) data structures. http://arxiv.org/abs/1407.6756, 2014. 52, 54, 55, 57, 59

[84] Daniel Larkin, Siddhartha Sen, and Robert Endre Tarjan. A back-to-basics empirical study of priority queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*, pages 61–72, 2014. 19

[85] Kasper Green Larsen, J. Ian Munro, Jesper Sindahl Nielsen, and Sharma V. Thankachan. On hardness of several string indexing problems. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 242–251, 2014. 49, 50

[86] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, chapter 3.4.3 Packing, Spreading, and Monotone Routing Problems. Morgan Kaufmann Publishers, Inc., 1991. 39

[87] Tom Leighton and C. Greg Plaxton. Hypercubic sorting networks. *SIAM Journal on Computing*, 27(1):1–47, 1998. 46

[88] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. 61

[89] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM (JACM)*, 48(3):407–430, 2001. 61

[90] Yossi Matias, S. Muthukrishnan, Süleyman Cenk Sahinalp, and Jacob Ziv. Augmenting suffix trees, with applications. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 67–78, 1998. 50

[91] Jiří Matoušek. Range searching with efficient hierarchical cuttings. *Discrete & Computational Geometry*, 10(2):157–182, 1993. 64

[92] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005. 82

[93] J. Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V. Thankachan. Top-k term-proximity in succinct space. In *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, pages 169–180, 2014. 49, 50

[94] James Ian Munro. An implicit data structure supporting insertion, deletion, and search in $\mathcal{O}(\log^2 n)$ time. *Journal of Computer and System Sciences (JCSS)*, 33(1):66–74, 1986. 9

[95] James Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences (JCSS)*, 21(2):236–250, 1980. 8

[96] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002. 50, 53, 103

[97] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):52, 2013. 50, 53

[98] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):art. 2, 2007. 52, 61

[99] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078, 2012. 52

[100] Gonzalo Navarro and Luís M. S. Russo. Fast fully-compressed suffix trees. In *Proc. 24th DCC*, pages 283–291, 2014. 61

[101] Gonzalo Navarro and Sharma V. Thankachan. Faster top-k document retrieval in optimal space. In *International Symposium on String Processing and Information Retrieval*, LNCS 8214, pages 255–262, 2013. 53

[102] Gonzalo Navarro and Sharma V. Thankachan. Top-k document retrieval in compact space and near-optimal time. In *International Symposium on Algorithms and Computation (ISAAC)*, LNCS 8283, pages 394–404, 2013. 53

[103] Gonzalo Navarro and Sharma V. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. *Theoretical Computer Science*, 542:83–97, 2014. 53

[104] Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Scandinavian Workshop on Algorithm Theory (SWAT), Proceedings*, pages 271–282, 2012. 63

[105] Harvey N.J.A. and Zatloukal K.C. The post-order heap. In *3rd International Conference on Fun with Algorithms*, 2004. 19, 20

[106] Mihai Pătraşcu. Succincter. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008. 73

[107] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 603–610, 2010. 55

[108] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. 56

[109] Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the thorup–zwick bound. *SIAM Journal on Computing*, 43(1):300–311, 2014. 52, 55

[110] Mihai Pătraşcu, Liam Roditty, and Mikkel Thorup. A new infinity of distance oracles for sparse graphs. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 738–747, 2012. 52, 55

[111] Mohammad Sohel Rahman and Costas S. Iliopoulos. Pattern matching algorithms with don't cares. In *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings Volume II*, pages 116–126, 2007. 56, 57

[112] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007. 62, 82

[113] Ralf Schenkel, Andreas Broschart, Seungwon Hwang, Martin Theobald, and Gerhard Weikum. Efficient text proximity search. In *SPIRE*, pages 287–299, 2007. 51

[114] Rahul Shah, Cheng Sheng, Sharma V. Thankachan, and Jeffrey Scott Vitter. Top-k document retrieval in external memory. In *Proceedings of*

European Symposium on Algorithms (ESA), LNCS 8125, pages 803–814, 2013. 52

[115] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences (JCSS)*, 18(2):110–127, 1979. 63

[116] Mikkel Thorup. Randomized sorting in $\mathcal{O}(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *Journal of Algorithms*, 42(2):205–230, 2002. 46

[117] Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM (JACM)*, 54(6), December 2007. 36

[118] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975. 35

[119] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973. 60

[120] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters (IPL)*, 17(2):81–84, 1983. 36

[121] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM (CACM)*, 7(6):347–348, 1964. 8, 18, 20

[122] Hao Yan, Shuming Shi, Fan Zhang, Torsten Suel, and Ji-Rong Wen. Efficient term proximity search with term-pair indexes. In *ACM Conference on information and knowledge management*, pages 1229–1238, 2010. 51

[123] Mingjie Zhu, Shuming Shi, Mingjing Li, and Ji rong Wen. Effective top-k computation in retrieving structured documents with term-proximity support. In *ACM Conference on information and knowledge management*, pages 771–780, 2007. 58

[124] Mingjie Zhu, Shuming Shi, Nenghai Yu, and Ji rong Wen. Can phrase indexing help to process non-phrase queries? In *ACM Conference on information and knowledge management*, pages 679–688, 2008. 51