Succinct Data Structures

Thesis submitted in
partial fulfillment of the
Degree of Doctor of Philosophy (Ph.D)

by

S. Srinivasa Rao
Theoretical Computer Science Group,
Institute of Mathematical Sciences,
Taramani, Chennai–600 113.

UNIVERSITY OF MADRAS
Chennai 600 005

December 2001

DECLARATION

I declare that the thesis entitled **"Succinct Data Structures"** submitted by me for the Degree of Doctor of Philosophy is the record of work carried out by me during the period from August 1997 to December 2001 under the guidance of Dr. Venkatesh Raman and has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other similar institution of higher learning.

December 2001 S. Srinivasa Rao

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113

CERTIFICATE

I certify that the thesis entitled **"Succinct Data Structures"** submitted for the Degree of Doctor of Philosophy by Mr. S. Srinivasa Rao is the record of work carried out by him during the period from August 1997 to December 2001 under my guidance and supervision, and that this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this University or any other University or Institution of Higher Learning.

I further certify that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of problems dealt with.

Venkatesh Raman
Thesis Supervisor

December 2001

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113

Abstract

The main aim of this thesis is to develop space efficient structures for some of the most fundamental problems in the area of data structures. In particular, we focus on the design of data structures that use almost optimal space while supporting the operations efficiently. The concrete problems we consider are: the representations of *suffix trees*, *suffix arrays*, *static dictionaries* supporting rank, *cardinal trees*, a list of numbers supporting *partial sum* queries, *dynamic bit vectors* and *dynamic arrays*. We look at these problems in the extended RAM model which supports all arithmetic and bitwise boolean operations on words in constant time. We assume an appropriate word size arising naturally from the problem instance. We also consider the problem of static dictionary in the bitprobe model.

We give the first space efficient suffix tree representation that answers all indexing queries in optimal time. For a text of length n over an alphabet Σ, this structure takes $n \lg n + O(n)$ bits of space and and supports searching for a a pattern of length m in $O(m \lg |\Sigma|)$ time. The main idea here is to use the succinct representation of binary trees to represent the tree structure of a suffix tree. For binary texts we also develop two index structures with better space complexity, but supporting a restricted set of indexing queries.

Extending the ideas of Grossi and Vitter, we give a compressed suffix array implementation that takes $o(n \lg n)$ bits of space and supports lookup (finding the i^{th} element in the suffix array, for any given i) in constant time. Using this representation, we give the first indexing structure with $o(n \lg n)$ bits of space that supports all indexing queries in optimal time.

We consider the static dictionary problem that also supports the rank operation for the elements present. This has applications in representing higher degree cardinal trees, which in turn have several applications including representing suffix trees with large alphabets directly without converting them into binary trees. We give a rank-dictionary for a subset of size n from a universe of size m that uses $n \lceil \lg m \rceil + O(\lg \lg m)$ bits of space and supports the membership and rank (for the elements present) operations in constant time. We then show a way of representing a set of dictionaries to support rank queries on individual dictionaries. Using the ideas of Benoit et al., we show that this immediately gives a structure to represent a cardinal tree using almost optimal space that supports all the navigational operations in constant time.

For the static dictionary problem in the bitprobe model, we develop a scheme to store two-element subsets of the universe $U = \{1, \ldots, m\}$ using $3m^{2/3}$ bits of space which can be used to answer membership queries using two adaptive bit probes. We show that this bound is tight for a restricted class of schemes. We then generalize this two-probe two-element scheme to a scheme to store n-element sets with $o(m)$ bits of space that answers queries using $\lg\lg n + 2$ adaptive probes. All these schemes are constructive.

We also look at some dynamic data structure problems where the operations are allowed to change the input data. We mainly focus on two classical inter-related problems: partial sums and dynamic arrays. For the partial sums problem on a sequence of n elements we give a space optimal structure that supports partial sum queries in $O(\log_b n)$ time and updates in $O(b)$ time, for any parameter $b \geq \lg n / \lg\lg n$. For the searchable partial sums problem, which also supports select queries, we give an optimal space structure that supports all the operations in $O(\lg n / \lg\lg n)$ worst-case time. As a special case of partial sums, we consider the dynamic bit vector problem where we give a structure that uses $o(n)$ bits of extra space for a given bit vector of length n, and supports rank and select operations in $O(\log_b n)$ time and flip in $O(b)$ amortized time.

For the dynamic array problem, we first give a structure that uses $o(n)$ extra words of space and supports updates in $O(n^\epsilon)$ worst case time and accesses in $O(1)$ worst-case time, for any fixed positive constant $\epsilon \leq 1$. Using this structure, we obtain a dynamic array structure that supports both query and update operations in optimal $O(\lg n / \lg\lg n)$ amortized time using $o(n)$ bits of extra space.

Acknowledgements

I am greatly indebted to Venkatesh Raman for his valuable guidance, encouragement and support throughout the years. He introduced me to the research area of succinct data structures, and many of the main ideas of the thesis were developed in collaboration with him. This thesis would not have been possible without his supervision and support. His wide range of interests has helped me learn a lot from him. I also thank my doctoral committee members — V. Arvind, Meena Mahajan and K. V. Subrahmanyam for their helpful suggestions and advice.

I have also had the pleasure of working with Rajeev Raman. His ideas and insights have greatly contributed to the thesis. I would also like to thank him for supporting me to work with him, at the university of Leicester.

I thank Ian Munro, Erik Demaine, Jaikumar Radhakrishnan and S. Venkatesh for the fruitful discussions I had with them which have also contributed to this thesis. I also wish to thank Sarmad Abbasi for the interesting technical discussions we had.

I am deeply grateful to V. Arvind, Kamal Lodaya, Meena Mahajan, Madhavan Mukund, R. Ramanujam, Anil Seth and K. V. Subrahmanyam, for their valuable teaching which has helped me understand several things, and also for their encouragement and support throughout.

I thank Prof. R. Balasubramanian, Director, IMSc, for providing all the facilities and a good research atmosphere. I would also like to thank all the office and library staff for being helpful at various stages of my work.

Thanks to all my colleagues and fellow-students in the theoretical computer science groups at IMSc and CMI — Swarup Kumar Mohalik, N. V. Vinodchandran, S. V. Nagaraj, P. Madhusudan, Deepak D'Souza, S. P. Suresh and B. Meenakshi, for the interesting discussions we had, academics and otherwise. I also thank Madhu for proof-reading part of the thesis.

I also thank my friends in Matscience — Balaji, Gyan Prakash, Ravindra, Sabu, Sridhar, Sumithra, Suri and others, for making my stay pleasant and wonderful.

I am grateful to my parents and my brothers Ramakrishna and Ravi, for the love, support and encouragement they have given me all along.

Finally, I would like to thank Jyoti for all her support and patience throughout.

Contents

Chapter 1

Introduction

1.1 Motivation and Background

The aim of this thesis is to develop succinct structures for some of the most fundamental problems in data structures. A data structure is called *succinct* if it uses an amount of space that is optimal to within lower order additive terms and still supports the required operations in asymptotically optimal time. Recently, there has been a surge of interest in the study of *succinct* data structures [BCD$^+$99, BM99, Cla96, GV00, Jac89b, MR97, MRS01].

Although the cost of memory is decreasing and the processor speeds are increasing day by day, the amount of textual data to be processed (such as dictionaries, encyclopedias, newspaper archives, web and genetic databases) is also increasing at a much higher rate. So, one is still interested in compact representations of data that support efficient retrieval. These representations have useful applications in portable devices (like mobile phones and smart cards) where the amount of memory is limited. Furthermore, it is an interesting and challenging problem from a theoretical view-point to develop data structures that use information-theoretically optimal space and support operations in asymptotically optimal time.

This thesis deals with succinct representations for the following data structures:

- suffix trees and suffix arrays used for text indexing
- static dictionaries supporting membership and rank

- trees of higher degree
- static dictionaries in the bitprobe model
- a list of numbers to support partial sum and update operations
- a bit vector supporting rank, select and flip operations and
- dynamic arrays.

Suffix Trees

String matching is one of the most fundamental problems in text processing. A suffix tree [Wei73] for a given string is a compact representation of all the suffixes of the text which enables efficient pattern searches. Suffix trees have been used to solve the string matching problem efficiently. They have also been applied to other fundamental string problems such as finding the longest repeated substring [Wei73], finding all squares or repetitions in a string [AP85], approximate string matching [LV89], text compression [RPE81], compressing assembly code [FWM84], inverted indices [Car75], and analyzing genetic sequences [CHM+86].

Classical representations of suffix trees have the drawback that the space required by the index is much higher than the space required to store the text itself, which makes it prohibitive in some applications. The problem of space efficient representation of index structures has been widely studied [Kär95, KU96, CM96, FG96, Irv95, Mäk00, GV00, FM00, Sad00]. We give the first $n \lg n + O(n)$-bit suffix tree structure, for a given text of length n, that supports indexing queries efficiently. We also give two improved index structures for binary texts, but with less functionality.

Suffix Arrays

A suffix array is an array storing the positions of all the suffixes in the given text in their lexicographic order. Given a text of length n, its suffix array can be stored using $n \lceil \lg n \rceil$ bits by explicitly storing all its entries in an array. This can be used to retrieve the i^{th} entry in constant time. But the disadvantage of this structure is that this uses more space than necessary, which could be prohibitive in some applications. One such application is the representation of a suffix tree; the storage requirement of a suffix tree includes, as one of the components, the position indices at the leaves, which is nothing but a suffix array.

Grossi and Vitter [GV00] have given an $O(n)$-bit representation of a suffix array, for a given binary text of length n, that supports indexing into the array in $O(\lg^{\epsilon} n)$ time, for any fixed positive constant $\epsilon < 1$. Using this, they have obtained an index structure that takes $O(n)$ bits of space and supports index queries, except finding all the occurrences, in optimal time, for binary texts. We give a suffix array representation that takes $O(n \lg^{\epsilon} n)$ bits and supports indexing into the array in constant time. Using this we obtain the first index structure for a binary text that takes $o(n \lg n)$ bits of space and supports *all* indexing queries in optimal time.

Static Dictionaries

Given a subset S of size n from a universe U of size m, a static dictionary for S is a representation of it that supports queries of the form 'Is x in S?', for any $x \in U$. This is a fundamental data-structure problem that has been widely studied in various models. Fredman et al. [FKS84] have given the first linear space ($O(n)$ words) data structure for this problem in the extended RAM model. Since then various structures have been proposed reducing the space further [SS90, FNSS92, BM99, Pag01a]. We give a representation that uses $n \lceil \lg m \rceil + O(\lg \lg m)$ bits of space and supports the membership and rank (for the elements present) operations in constant time. We also give a space efficient representation for a set of dictionaries to support rank queries on individual dictionaries.

Cardinal Trees

Trees form an important structure in many computing applications. However, their explicit representation using pointer based methods has a space requirement that is much higher than the optimum. Recent work [Jac89b, Cla96, MR97, Ben98, BDMR99, MRS01] has focused on space efficient representations of trees that support navigational operations efficiently.

We consider the problem of representing cardinal trees. A k-ary cardinal tree is a generalization of a binary tree. It is a rooted ordered tree in which the children of any node are uniquely labeled from the set $\{1, \ldots, k\}$. Benoit et al. [BDMR99] have given a representation that, for a k-ary tree on n nodes, uses $2n + n \lceil \lg k \rceil + o(n)$ bits and supports 'parent', 'i^{th} child', 'degree' and 'subtree size' operations at any given node in constant time, and finding a child labeled i in $O(\lg \lg k)$ time. Using our multiple dictionary representation, we obtain a k-ary cardinal tree representation takes $2n + n\lceil \lg k \rceil + o(n) + O(\lg\lg k)$ bits of space that supports *all* navigational operations in constant time.

Static Dictionaries in the Bitprobe Model

Minsky and Papert [MP69] have proposed the bitprobe model and studied the static membership problem in that model. Recently, Buhrman et al. [BMRV00] have studied this problem further and have given several upper and lower bounds for randomized and deterministic schemes. For the deterministic case, they have shown that the simple bit vector scheme and the static dictionary structure of Fredman et al. [FKS84] are optimal. They have also shown the existence of a deterministic scheme with $o(m)$ bits of space that answers membership queries using constant number of probes, where m is the size of the universe.

We first give an explicit two-probe adaptive scheme with $O(m^{2/3})$ bits of space, improving the existential scheme given in [BMRV00] that takes $O(m^{3/4})$ bits. We show that our bound is optimal for a restricted class of schemes. From a generalization of this scheme, we obtain an adaptive scheme with $o(m)$ bits of space that answers queries using $O(\lg\lg n)$ probes, where n is the size of the set.

Partial Sums

Given a sequence of n elements each in the range $\{0, \ldots, 2^k - 1\}$,where k is $O(\lg n)$, the partial sums problem is to maintain the sequence under the operations of incrementing and decrementing the elements and finding the partial sum up to a given index. Fredman and Saks [FS89] have shown a lower bound of $\Omega(\lg n/\lg\lg n)$ time for these operations. Dietz [Die89] has given a structure that supports all the operations in $O(\lg n/\lg\lg n)$ time using $O(n \lg n)$ bits. We modify Dietz's structure to obtain a data structure that uses $kn+o(kn)$ bits of space. This structure supports, apart form partial sum and update operations, the operation of finding the index of an element with a given partial sum, all in $O(\lg n/\lg\lg n)$ time. We also show trade-offs between the query and update times.

As a special case, we consider a dynamic bit vector supporting *rank*, *select* and *flip* operations. A bit vector supporting *rank* and *select* is a fundamental building block for several succinct static data structures. They have been used, for example, in the succinct representations of binary trees [Jac89b, Ben98, MR97], static dictionaries [BM99], and compressed representations of suffix arrays [GV00].

Hence the dynamic bit vector problem is likely to have potential applications in making these static data structures dynamic.

Dynamic Arrays

An array is perhaps the most primitive data structure. Almost all high-level programming languages and assembly languages support an array data type. An array is a static data type; it does not allow for element insertions and deletions, but only allows element replacements and accesses based on rank (position). There are, nevertheless, several applications where one would like to maintain a sequence of elements under insertions and deletions and still be able to access them based on their rank. This is in fact the motivation for the inclusion of a *dynamic array* data type in the Java language.

Standard implementations of a dynamic array either use the most obvious scheme of storing the elements in an array or use a balanced search tree to maintain the sequence of elements. The first implementation supports accesses in constant time and updates (inserts and deletes) in $O(n)$ time, where n is the number of elements in the current sequence. The second implementation supports both accesses and updates in $O(\lg n)$ time using $O(n)$ words of extra space.

The *tiered vector* data structure of Goodrich and Kloss [GI99] uses $O(n^{1-\epsilon})$ words of extra space and supports updates in $O(n^{\epsilon})$ amortized time while supporting accesses in constant worst-case time, for any fixed positive constant $\epsilon \leq 1$. We first give a structure that achieves the same space and time bounds with no amortization. Using this structure, we obtain a dynamic array structure that supports both query and update operations in optimal $O(\lg n / \lg\lg n)$ amortized time using $o(n)$ bits of extra space.

1.2 Our Contributions

- A suffix tree representation for a text of length n over an alphabet Σ that can be stored using $n \lg n + O(n)$ bits such that given a pattern of length m, the number of occurrences of the pattern in the string can be found in $O(m \lg |\Sigma|)$ time. Finding all the occurrences of the pattern takes an additional $O(occ)$ time, where occ is the number of occurrences. Using this representation as a substructure, we give the following data structures for the indexing problem for binary texts (i.e., $|\Sigma| = 2$):

- An indexing structure that uses $\frac{n}{2}\lg n + O(n)$ bits of space and supports finding an occurrence of the pattern, if it exists, in $O(m)$ time.
- An indexing structure that requires $o(n \lg n)$ bits of space and answers whether the given pattern occurs in the text in $O(m)$ time.

- A compressed suffix array representation for a given binary text of length n that can be stored using $O(nt(\lg n)^{1/t})$ bits of space and answers *lookup* queries in $O(t)$ time, for any parameter $1 \leq t \leq \lg\lg n$. Using this structure and the ideas of Grossi and Vitter [GV00], we give an indexing data structure for a binary text of length n that uses $O(n \lg^{\epsilon} n)$ bits of space and answers indexing queries in $O(m/\lg n)$ time, for any fixed positive $\epsilon < 1$. Finding all the occurrences of the pattern requires an additional $O(occ)$ time, where occ is the number of occurrences of the pattern in the text.

- A static dictionary data structure to represent a subset of size n from a universe of size m using $n\lceil \lg m \rceil + O(\lg\lg m)$ bits of space that supports *membership* and *rank* (for the elements present in the set) operations in constant time. We then use the ideas of *universe reduction* and *sharing primes* to get a structure that stores a set of dictionaries with total cardinality n over a universe of size m, using $n\lceil \lg m \rceil + O(\lg\lg m)$ bits of space which supports rank (and membership) queries on individual dictionaries in constant time. Using this structure and the ideas of Benoit et al. [BDMR99], we give an n-node k-ary cardinal tree representation that uses $2n + n\lceil \lg k \rceil + o(n) + O(\lg\lg k)$ bits of space and supports 'parent', 'i^{th} child', 'child labeled j', 'degree' and 'subtree size' operations at any given node in constant time.

- Explicit constructions for static dictionaries in the bitprobe model when the set size and the number of probes are small compared to the universe size. We give a 2-probe adaptive scheme for sets of size at most 2 that takes $3m^{2/3}$ bits of space, where m is the size of the universe, and also show that this scheme is optimal for a restricted class of storage schemes. We then generalize this to an adaptive scheme for storing sets of size at most n that takes $m^{k/(k+1)}\left(\lg(k+1) + \frac{1}{k}\lg n + kn^{1/k}\right)$ bits of space and answers membership queries using $\lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil + 1$ bit probes, for any parameter $1 \leq k \leq \lg n - 1$.

- Dynamic data structures for some fundamental problems in data structures, namely maintaining partial sums, dynamic bit vector and dynamic array.
  - For the partial sums problem, we give a succinct structure that supports *sum* in $O(\log_b n)$ time and *update* in $O(b)$ time, for any parameter $b \geq \lg n / \lg \lg n$. For the searchable partial sums problem, we give an optimal space structure that supports all the operations in $O(\lg n / \lg \lg n)$ worst case time.
  - For the dynamic bit vector problem, we give a structure that uses $o(n)$ bits of extra space and supports *rank* and *select* operations in $O(\log_b n)$ time and *flip* in $O(b)$ amortized time, for any parameter $b \geq \lg n / \lg \lg n$.
  - For the dynamic array problem, we give two structures: one using $O(n^{1-\epsilon})$ extra words of space that supports updates in $O(n^{\epsilon})$ worst case time and queries in $O(1)$ worst-case time, for any fixed positive constant $\epsilon \leq 1$, and another that supports both query and update operations in optimal $O(\lg n / \lg \lg n)$ amortized time using $o(n)$ bits of extra space.

1.3 Model of Computation

We work with an extended word RAM model throughout the thesis, except in Chapter 4 where we work with a bitprobe model. The word RAM [Hag98] has $2^{O(w)}$ registers each of which stores a w-bit word. All operations on words that are usually available on modern computers, namely memory access, flow control, comparisons, basic arithmetic operations (including multiplication and division), bitwise shift and Boolean operations can be performed in constant time. In this model, space is measured in terms of the number of words or bits used by a structure and time in terms of the number of unit-cost operations performed.

In the bitprobe model, the space is counted in terms of the number of bits used by the data structure and time in terms of the number of bits probed from the data structure. Time for any other computation is not counted.

The word-size of a RAM is typically assumed to be large enough to hold an input word and so we assume an appropriate word size naturally arising from the problem instance. In particular, for the problems of suffix array and suffix tree representations with text length n, the dynamic partial sums problem on n numbers and for the dynamic bit vector problem for an n-bit vector, we assume a word size of $\lg n$ bits. For the static dictionary problem with universe size m, we assume a word size of $\lg m$ bits and for the representation of k-ary cardinal trees on n nodes, we assume a word size of $max\{\lg n, \lg k\}$ bits. For the dynamic array problem, we assume a word size of $\lg n$ bits where n is the maximum limit on the length the array is allowed to grow.

1.4 Notations

For a real number x, $\lceil x \rceil$ denotes the smallest integer greater than or equal to x and $\lfloor x \rfloor$ denotes the largest integer less than or equal to x.

For integers x and a, we define $div(x, a) = \lfloor x/a \rfloor$ and $mod(x, a) = x - a \cdot div(x, a)$. We also use the notations $(x \bmod a)$ and $(x \text{ div } a)$ to denote $mod(x, a)$ and $div(x, a)$ respectively.

We use the notation $[m]$ to denote the set $\{0, 1, \ldots, m-1\}$, for any integer $m \geq 1$.

The notation $\lg x$ denotes the logarithm of x to the base 2, $\ln x$ denotes the natural logarithm of x, and e denotes the base of the natural logarithm.

1.5 Overview of the Thesis

In Chapter 2 we look at some space efficient representations of suffix trees, other indexing structures and compressed suffix arrays. Chapter 3 deals with static dictionary data structures and their applications to cardinal tree representation. In Chapter 4 we consider the static dictionary problem in the bitprobe model and give some upper and lower bounds. Chapter 5 describes some dynamic data structures for the partial sums, dynamic bit vector and dynamic array problems. We conclude with a summary and discussion of open problems in Chapter 6.

Chapter 2

Succinct Data Structures for Indexing

2.1 Introduction

In this chapter, we look at some data structures used for the exact string matching problem. Given a text string T and a pattern string P over a finite alphabet Σ, the (exact) string matching problem is to find the occurrences of the pattern P in the text T. When the text T is given in advance and is allowed to be preprocessed, it is known as the *indexed* (or off-line or static) string matching problem. A data structure designed for this problem is called an *index* for the given text. The string matching problem has been well studied and various data structures have been proposed to solve this efficiently.

Given a text T and a pattern P, we consider four types of queries: *existential*, *search*, *counting* and *enumerative*. An existential query returns a boolean value that says if P is contained in T. A search query returns a position of occurrence of P in T if it exists. A counting query returns the number of occurrences of P in T and an enumerative query outputs the list of positions where P occurs in T. Throughout this chapter, we assume that the given text T is a string of length n and that the given pattern P is a string of length m both over an alphabet Σ, if not mentioned explicitly. By a binary string or a bit vector, we mean a string over the alphabet $\{0, 1\}$. Also, we use *occ* to refer to the number of occurrences of the given pattern in the text.

2.1.1 Background

A *suffix tree* is the most commonly used text index structure for the string matching problem, which was first described by Weiner [Wei73]. A suffix tree represents all the suffixes of a given string in a space efficient manner such that string matching queries can be answered efficiently. A suffix tree, in its standard form [McC76, Wei73], takes $O(n \lg n)$ bits of space and supports search (and hence existential) queries in $O(m \lg |\Sigma|)$ time and enumerative queries in $O(m \lg |\Sigma| + occ)$ time. It can also be used to support counting queries in $O(m \lg |\Sigma|)$ time using some extra space. One can also represent a suffix tree using $O(n|\Sigma| \lg n)$ bits of space where search and counting queries can be supported in $O(m)$ time. The search algorithm uses the fact that a pattern P occurs in a text T if and only if P is a prefix of some suffix of T (see Section 2.2.1 for details).

Standard representations of suffix trees for a text of length n take about $5n$ words or pointers, where each word/pointer is a $\lg n$ bit string, and the original text is retained. Reducing the space requirement (by a constant factor) has been a main theme in the developments on the indexing structures. We briefly describe some of them here.

Suffix arrays [GBYS92, MM93] are widely used for indexing large texts. A suffix array stores the array containing pointers to all the suffixes in the lexicographic order along with some auxiliary information to aid in searching. This fairly standard representation of a suffix array, along with the auxiliary structure, takes about $2n$ words of space. Search and counting queries can be supported using a suffix array in $O(m + \lg n)$ time and enumerative queries in $O(m + \lg n + occ)$ time.

Colussi and De Col [CC96] have proposed the *augmented suffix array* as a space efficient alternative to the suffix tree structure. It consists of a suffix array for every block of $\lg n$ suffixes and a sparse suffix tree for every $(\lg n)^{th}$ suffix (beginning of each block), in the lexicographic order of the suffixes. This structure requires $n \lg n + O(n)$ bits of space and can be used to support search and counting queries in $O(m \lg |\Sigma| + \lg \lg n)$ time. Enumerative queries take an additional $O(occ)$ time.

Some other well known index structures include: the *suffix cactus* proposed by Kärkkäinen [Kär95], the *sparse suffix tree* proposed by Kärkkäinen and Ukkonen [KU96], efficient suffix trees for the secondary storage developed by Clark and Munro [CM96], the *string B-tree* developed by Ferragina and Grossi [FG96] and the *suffix binary search tree* designed by Irving [Irv95].

All these index structures either require at least $2n$ words of space or require $\omega(m)$ time to answer search queries, in the worst case. So, in [Mut97], Muthukrishnan asked whether there exists a data structure that uses only $n + o(n)$ words and answers search queries in $O(m)$ time. We propose some index structures answering his question. The model used is the extended RAM model with word size $O(\lg n)$ bits (where n is the length of the given text).

2.1.2 Overview

The next section first reviews the suffix tree data structure and the succinct binary tree representation and describes algorithms to support the additional operations. It then explains how the succinct binary tree representation can be used to obtain our first space efficient suffix tree structure taking $n + o(n)$ words of space. In Section 2.3, we give two space efficient index structures for binary texts. In particular, Section 2.3.1 gives a structure which takes $\frac{n}{2} \lg n + O(n)$ bits of space and answers search queries in $O(m)$ time. Section 2.3.2 describes a structure to answer the existential queries in $O(m)$ time, which takes $o(n \lg n)$ bits of space. In Section 2.4, we describe our compressed suffix array representation and explain how this can be used to obtain an indexing structure with $o(n \lg n)$ bits of space which can be used to support enumerative queries in $O(m/\lg n + occ)$ time.

2.2 Space Efficient Suffix Trees

In this section, we first outline the suffix tree data structure and explain how they can be represented space efficiently using succinct representation of binary trees.

2.2.1 Suffix Trees

Suffix trees [McC76] are the most fundamental data structures used for indexing which admit efficient online string searches. They have also been applied to other fundamental string problems such as finding the longest repeated substring, approximate string matching, text compression, compressing assembly code, inverted indices, and analyzing genetic sequences.

Definition 2.2.1 *Given a set of n strings $\{S_1, S_2, \ldots, S_n\}$ over a finite alphabet Σ, a trie for the set is an edge-labeled rooted directed tree with vertex set V =*

$\{w | w$ *is a prefix of* $S_i, 1 \leq i \leq n\}$ *and edge set* $E = \{w \overset{a}{\rightarrow} wa | w, wa \in V, a \in \Sigma\}$.

If the sum of the lengths of all the n strings in the set is s, then the trie for the set of strings could, in the worst case, have $O(s)$ nodes with at most n external nodes (leaves). To reduce the number of (internal) nodes, one can compress a trie by attaching a node with a single child to its parent and concatenating the edge labels of the two edges. For several applications, we would also like to have each string in the set to be represented by a leaf. This leads us to the following definition of a *compressed trie* [Gus97]:

Definition 2.2.2 *Given a set of n strings $\{S_1, S_2, \ldots, S_n\}$, a compressed trie for the set is a rooted ordered tree with n leaves numbered* 1 *to n. Each internal node other than the root, has at least two children and each edge is labeled with a nonempty substring of some string in the set. No two edges out of a node can have edge-labels beginning with the same character. For the leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spell-out the string S_i.*

Definition 2.2.3 *A suffix tree for a text string T is a compressed trie for all the suffixes of T.*

This definition of a suffix tree does not guarantee that a suffix tree exists for any given string T (i.e., a compact trie may not exist for a given set of strings). The problem is that if one suffix of T matches a prefix of another suffix of T then the path for the first suffix would not end at a leaf and hence no suffix tree obeying the above definition is possible. To avoid this problem, we assume that the last character of T appears nowhere else in T. To achieve this in practice, we can add a character to the end of T that is not in the alphabet that the string is taken from. In this chapter, we assume (unless explicitly stated) that the given text string is appended by a special character \$ at the end.

Since we have n leaf nodes and at most $n - 1$ internal nodes (as each internal node has at least two children), the tree has at most $2n - 1$ nodes and hence at most $2n - 2$ edges. Since the sum of the lengths of all the edge labels in a suffix tree could be $\Theta(n^2)$, the tree size (space required to store a representation) could also be of the same order. One can reduce the size of the tree to $O(n)$ words [McC76, Mor68, Wei73] by storing the edge labels efficiently. As each edge label is a substring of the text, one can represent the label using a pointer into the text

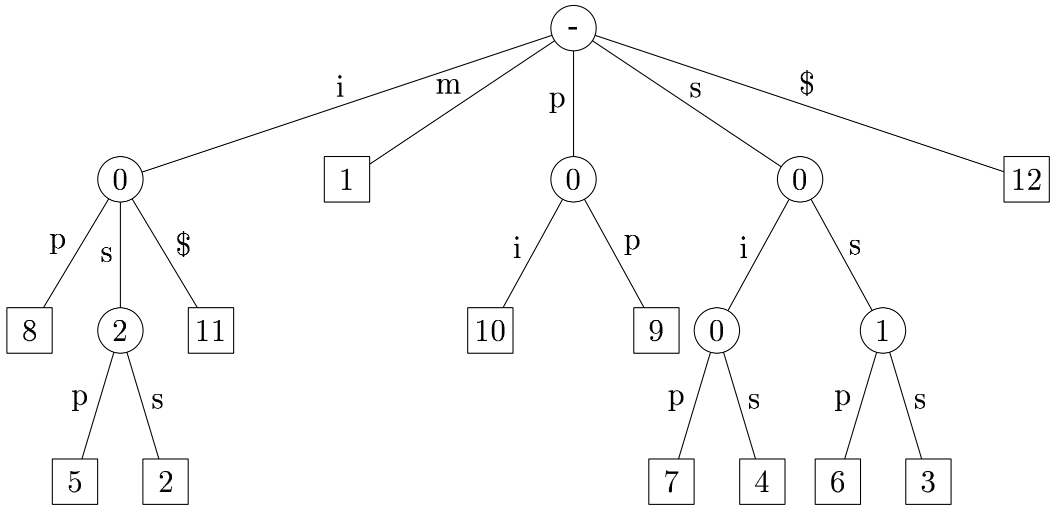

Figure 2.2.1: Suffix tree for the string “mississippi$”

where the substring corresponding to the edge label starts, and the length of the edge label, called the *skip value.* Thus, each edge label can be stored using two words, and with this new representation of edge labels, the tree can be stored using $O(n)$ words.

For the purpose of pattern matching, it is enough to store the starting character of an edge label and the skip value, as explained below. We associate this skip value with the node pointed to by the edge and store it with that node. Also, we do not need to store the skip value associated with a leaf. Thus, with each internal node (except the root), we store the skip value associated with it and with each external node (leaf), we store a pointer to the starting position of the suffix represented by that leaf. See Figure 2.2.1.

Given a pattern p and the suffix tree for a string x over an alphabet Σ, to search for an occurrence of p in x, we start at the root of the tree and follow the path labeled by the search string. At any node, we take the branch (edge) that matches the current character of the pattern and skip those many characters in the pattern as specified by the skip value at that node (this could be 0 at some nodes). Finding the branch that matches the current character of the pattern can be performed in $O(\lg|\Sigma|)$ time, by storing the first characters of the edge labels in sorted order, at each node.

The search is continued until either the pattern is exhausted (at a node or in an edge), or the current character of the pattern has no match at the current node. In the latter case, there is no occurrence of the pattern in the text. In the former case, let v be the node at which the search has ended (this could even be a leaf). If the pattern is exhausted in an edge, i.e., the skip value at a node is more than the length of the pattern left, then take v to be the node pointed to by that edge. Now, the position value stored in any leaf of the subtree rooted at v gives a *possible* starting point of the pattern in the text. (This only gives a possible starting position in the text since we might have skipped characters in the middle while matching the pattern in the suffix tree). Also, the pattern occurs in the text if and only if it occurs at all the positions in the text, stored at the leaves in the subtree rooted at v. We start at a position given by any of the leaves in the subtree rooted at v and confirm if the pattern exists in the text starting from that position. Clearly, the above procedure takes $O(m \lg |\Sigma|)$ time to find an occurrence of the pattern in the text, if it exists. Counting queries can be answered in the same amount of time by storing, with each internal node, the number of leaves in the subtree rooted at that node. Enumerative queries can be answered with an additional $O(occ)$ time by traversing the leaves of the subtree rooted at v (the node where the search has ended successfully).

2.2.2 Storage Requirement of Suffix Trees

The storage requirement for a suffix tree comprises of the space to store the following quantities [CM96]:

1. the tree (trie) structure,
2. the first characters of the edge-labels,
3. the skip values at the internal nodes,
4. the number of leaves in the subtree rooted at each internal node (to support counting queries) and
5. the position indices at the leaves (suffix array).

Thus, a straightforward representation of each of these items requires approximately $5n \lg n$ bits of space for a given text of length n. In the next section, we

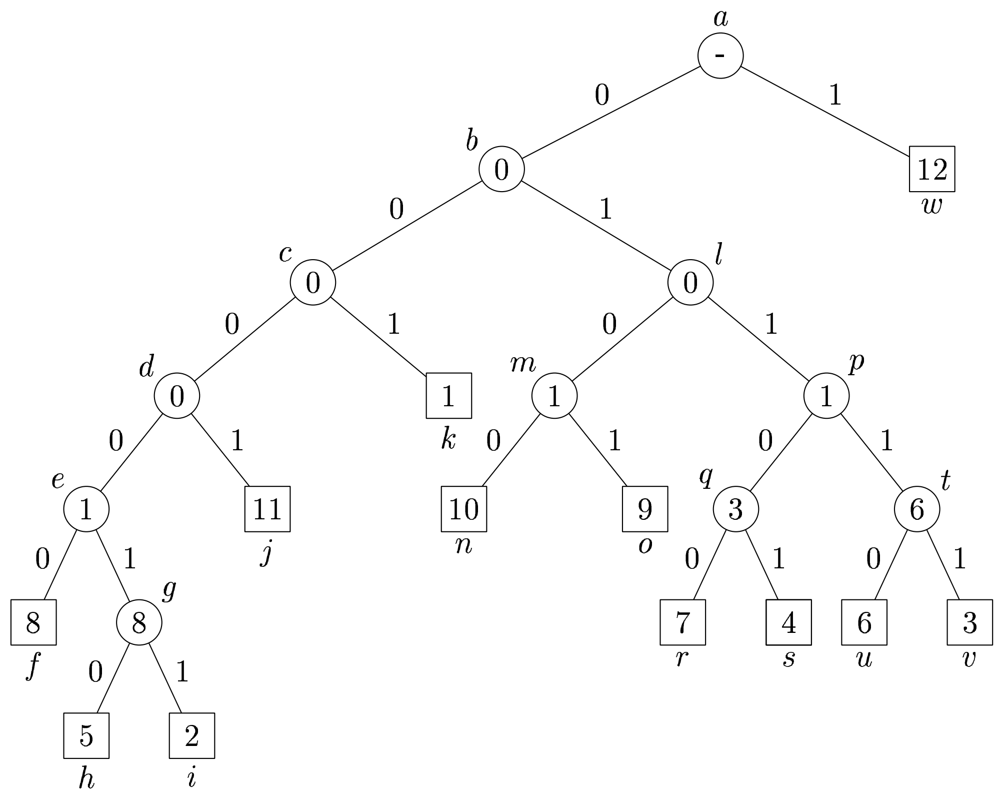

Figure 2.2.2: Suffix tree of for the string "mississippi$" using the binary encoding of the characters: i - 000, m - 001, p - 010, s - 011, $ - 100

show how each of these components can be stored succinctly. In particular, we give a suffix tree structure that takes $n \lg n + O(n)$ bits of space.

We first observe that, even if the given text is on a binary alphabet, its suffix tree will be a ternary tree (due to the extra $ character at the end). So first we will consider a binary alphabet by converting each symbol of the alphabet Σ and the symbol $ into binary using an encoding that assigns a binary string of length $\lceil \lg(|\Sigma| + 1) \rceil$ bits to each character, preserving the lexicographic order. Thus, given a text string $x\$$ of length n, we encode all the suffixes of it in binary and construct a compressed trie for them, which will be a binary tree. (This is the same as the PAT tree of Gonnet et al. [GBYS92].) This trie representing all the suffixes of the text in binary, is a special case of a binary tree in which all the internal nodes have exactly two children (since all nodes with a single child have been compressed). Since there are n suffixes, there will be $n - 1$ internal nodes and n external nodes in the trie. Thus we can represent it as a $2n - 1$ node binary tree. See Figure 2.2.2.

2.2.3 Succinct Binary Tree Representation

The starting point for our representation is the $2n + o(n)$ bit encoding of an n node static binary tree [MR97]. This structure supports finding the left child, right child or the parent of a node and reporting the size of the subtree rooted at a given node, all in constant time. We will, however, require a few more primitive navigational operations to support our suffix tree operations.

First, we review the succinct representation of binary trees and describe algorithms to support additional operations.

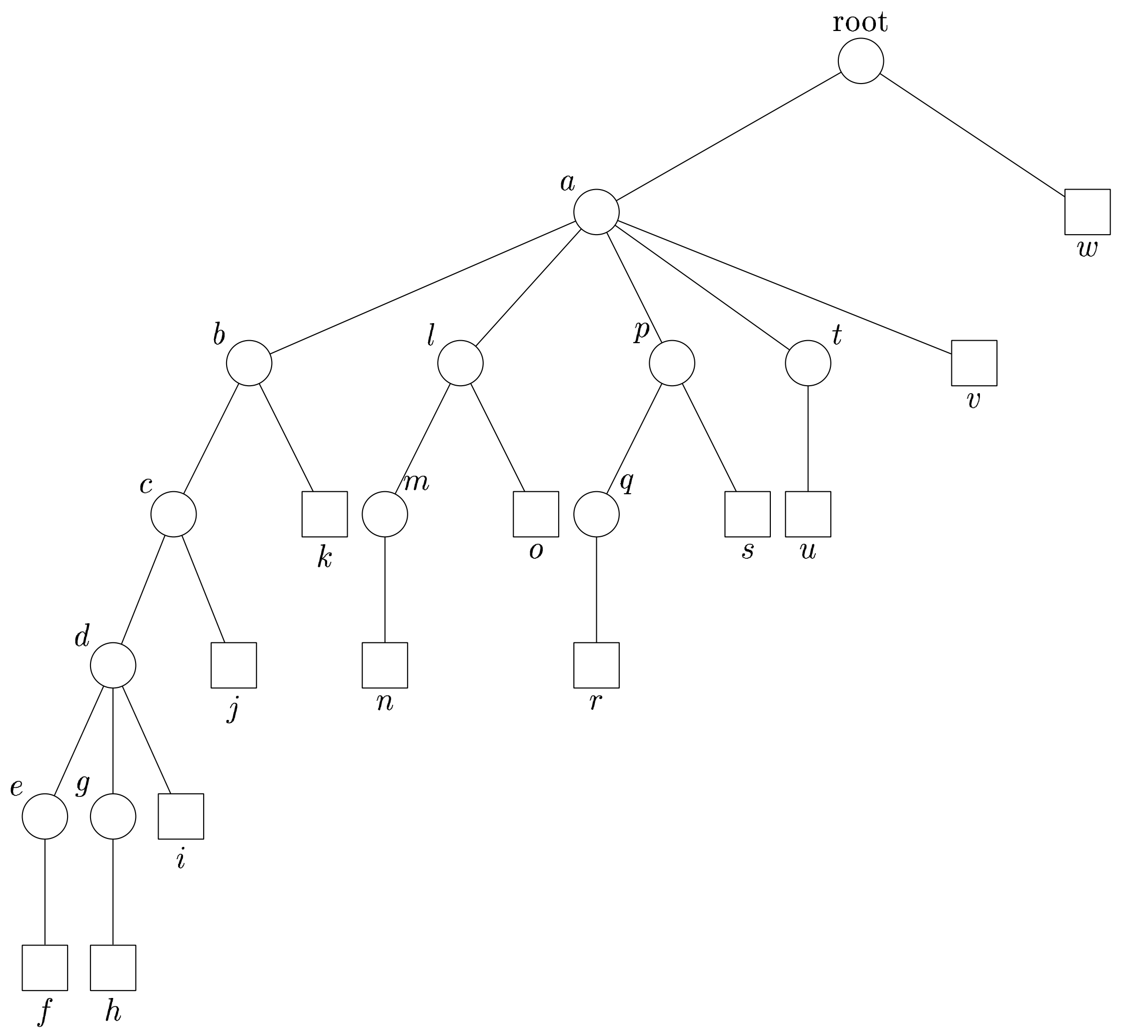

Figure 2.2.3: The rooted ordered corresponding to the binary tree of Figure 2.2.2. The parenthesis representation corresponding to this ordered tree is given below: ((((((( )) (( )) ( )) ( )) ( )) ((( )) ( )) ((( )) ( )) (( )) ( )) ( ))

A general rooted ordered tree on n nodes can be represented by a balanced string of $2n$ parentheses as follows: Perform a preorder traversal of the tree starting at the root; write an open parenthesis when a node is first encountered, going down the tree and a closing parenthesis while going up after traversing the subtree.

One cannot use this procedure directly to represent a binary tree, as it is not possible to distinguish a node with a left child but no right child from one with a right child but no left child. So, Munro and Raman [MR97] use the well known isomorphism between the class of binary trees and the class of rooted ordered trees to convert the given binary tree into a general rooted ordered tree and then represent the rooted ordered tree using the above parenthesis representation. Note that, this conversion preserves the left to right ordering of the leaves. See Figure 2.2.3. In the ordered tree there is a root which does not correspond to any node in the binary tree. Beyond this, the left child of a node in the binary tree corresponds to the leftmost child of the corresponding node in the ordered tree, and the right child in the binary tree corresponds to the next sibling to the right in the ordered tree. A node is identified, by convention, by its corresponding left parenthesis. Munro and Raman [MR97] show that using $o(n)$ additional bits, the standard operations of *left child*, *right child* and *parent* of a given node can be supported in constant time. They also show that the size of the subtree rooted at a given node can be found in constant time.

To use this tree representation to represent a suffix tree, we need to support several additional operations in constant time. Let x be any node in the given binary tree. (We identify a node in the binary tree by its preorder number.) We define the following operations:

- *leafrank*(x): return the number of leaves to the left of node x in the preorder numbering of the nodes
- *leafselect*(j): return the j^{th} leaf in the left to right ordering of the leaves
- *leafsize*(x): return the number of leaves in the subtree rooted at node x
- *leftmost*(x): return the leftmost leaf in the subtree rooted at node x and
- *rightmost*(x): return the rightmost leaf in the subtree rooted at node x.

Beginning with Jacobson [Jac89a], much of the work on navigating succinct representations of trees [Ben98, BDMR99, Mun96, MR97] has relied on the operations *rank* and *select* defined on binary strings. Given a binary string (bit vector) of length n and an index i, $1 \le i \le n$, these operations are defined as:

- $\mathit{rank}_1(i)$: the number of 1's up to and including the position i and
- $\mathit{select}_1(i)$: the position of the i^{th} 1, if it exists.

One can analogously define the operations $\mathit{rank}_0(i)$ and $\mathit{select}_0(i)$ for the 0's in the bit string. We refer to all these operations as the *rank* and *select* operations on the bit string. The *rank* and *select* operations on a given bit vector of length n can be supported in constant time using auxiliary structures of size $o(n)$ bits [Jac89b, Cla96, Mun96]. We call these auxiliary structures, the *rank* and *select* directories for the given bit vector.

Given a binary string of length n and a binary pattern p of fixed length m, one can generalize the *rank* and *select* operations as follows:

- $\mathit{rank}_p(i)$: the number of (possibly overlapping) occurrences of the pattern p up to and including the position i in the given binary string and
- $\mathit{select}_p(i)$: the position of the i^{th} occurrence of p in the given binary string.

The following theorem shows that these operations can also be supported in constant time using $o(n)$ bits of extra space, when the length of p is at most $\epsilon \lg n$ for some fixed positive constant $\epsilon < 1$.

Theorem 2.2.1 *Given a binary string of length n, and a binary pattern p of length $m \le \epsilon \lg n$, for any fixed positive constant $\epsilon < 1$, the operations $\mathit{rank}_p(i)$ and $\mathit{select}_p(i)$ can be supported in constant time using $o(n)$ bits, in addition to the space required for the given binary string.*

Proof. The algorithm to support the rank_1 operation [Cla96, Jac89a, Mun96] uses the following basic idea: Divide the given bit string into blocks of size $\lceil \lg n \rceil^2$ each and store the number of 1's up to the first element of every block in an array, using $\lfloor n / \lceil \lg n \rceil \rfloor$ bits of space. Within a block of size $\lceil \lg n \rceil^2$, keep a recursive structure by dividing the blocks into sub-blocks of size $\lceil (\lg n)/c \rceil$ for some integer constant $c \ge 2$ and store the rank information for the first element of each sub-block with respect to the beginning of the block. These sub-block sizes are small enough so that we can keep a precomputed table of answers for all possible distinct sub-blocks of the required size, using $o(n)$ bits. The precomputed table stores the number of ones up to a given position for each sub-block of length $\lceil(\lg n)/c\rceil$ and for each position in the sub-block. These entries are stored in the lexicographic order so that given a bit vector corresponding to a sub-block and a position in the sub-block, one can index into this table for the corresponding entry in constant time.

This structure can easily be adapted to keep the $rank_p$ information for every block and for every sub-block with respect to its block. The precomputed table, in this case, stores an entry for every possible triple consisting of a block b of size $\lceil(\lg n)/c\rceil$, a bit string x of size $m-1$, and a position i in the block. The table entry stores the number of occurrences of the pattern p in the prefix of length $|x|+i$ of the string xb. This also takes care of occurrences of the pattern in a span of two consecutive blocks. This table requires $O(2^m 2^{(\lg n)/c} 2^{\lg\lg n} \lg\lg n)$ bits. By choosing c to be $\lceil\frac{1}{1-\epsilon}\rceil + 1$, the size of the table becomes $o(n)$ bits. Again the entries are stored in the lexicographic order to enable constant time access.

We now, briefly describe the structure for computing $select_1$ [Cla96], which uses three levels of auxiliary directories. The first level auxiliary directory records the position of every $(\lceil\lg n\rceil \lceil\lg\lg n\rceil)^{th}$ one bit. This requires at most $\left\lfloor\frac{n}{\lceil\lg\lg n\rceil}\right\rfloor$ bits of space. Let r be the size of a range between two consecutive values in the first level auxiliary directory. If $r \geq (\lceil\lg n\rceil \lceil\lg\lg n\rceil)^2$ then we will explicitly store the positions of all the one bits in that range, which requires $\lceil\lg n\rceil^2 \lceil\lg\lg n\rceil$ bits of space which is at most $\left\lfloor\frac{r}{\lceil\lg\lg n\rceil}\right\rfloor$ bits, in the second level auxiliary directory. Otherwise, we subdivide the range and record the position, relative to the start of the range, of each $(\lceil\lg r\rceil \lceil\lg\lg n\rceil)^{th}$ one bit in the second level auxiliary directory, which takes at most $\left\lfloor\frac{r}{\lceil\lg\lg n\rceil}\right\rfloor$ bits.

Let r' be the size of a subrange between two consecutive values in the second level auxiliary directory. If $r' \geq \lceil\lg r'\rceil \lceil\lg r\rceil \lceil\lg\lg n\rceil^2$, then store positions of all the one bits in the subrange, with respect to the beginning of the subrange, explicitly, which requires at most $\left\lfloor\frac{r'}{\lceil\lg\lg n\rceil}\right\rfloor$ bits. Otherwise, one can show that $r' < 16\lceil\lg\lg n\rceil^4$. Computing *select* on a range of $l = O((\lg\lg n)^4)$ bits is performed using table lookup. For each possible bit string of length l and each value i in the range $1 \ldots l$ we record the position of the i^{th} one in the bit string, in a precomputed table. The storage used for the auxiliary directories and the lookup tables is $\frac{3n}{\lceil\lg\lg n\rceil} + O(2^l l \lg l)$ which is $o(n)$ bits, since $l = O((\lg\lg n)^4) = o(\lg n)$. Note that we know where the appropriate directory bits at each level are located and how to interpret them based on the value of i and the preceding directory levels. This gives a structure for supporting $select_1$ in constant time.

Now, to support $select_p$ on a given bit string, we again store the three levels of auxiliary directories for the pattern p, as in the case of $select_1$. The table structure now records, for each possible bit string of length $l = O((\lg\lg n)^4)$ and each value i in the range $1 \ldots l$, the position of the i^{th} occurrence of the pattern p, if it exists, in the bit string. One can easily verify that this structure takes $o(n)$ bits of space. This gives a structure to support the $select_p$ operation on the given binary string in constant time for any pattern p of length at most $\epsilon \lg n$, where $\epsilon < 1$ is any fixed constant. ∎

Now, we give a succinct representation of a static binary tree which supports several navigational operations in constant time, using the above theorem.

Theorem 2.2.2 *A static binary tree on n nodes can be represented using $2n + o(n)$ bits such that given a node x, in addition to finding its parent, left child, right child and the size of the subtree rooted at node x, we can also support leafrank(x), leafselect(j), leafsize(x), leftmost(x) and rightmost(x) operations in constant time, for $1 \leq j \leq l$ where l is the number of leaves in the tree.*

Proof. The fact that *parent, left child, right child and the subtree size* are supported in constant time is already known [MR97]. To support the other operations, we first convert the binary tree into an equivalent rooted ordered tree, as before.

Any leaf in the binary tree is a leaf in the corresponding rooted ordered tree, but not vice versa. In fact, any leaf in the rooted ordered tree is a leaf in the binary tree only if it is the last child of its parent. In other words, leaves in the binary tree correspond to the rightmost leaves in the rooted ordered tree. Also the left to right ordering of the leaves of the subtree rooted at any node is preserved between the binary tree and its corresponding rooted ordered tree. See figures 2.2.2 and 2.2.3.

In the parenthesis notation, a rightmost leaf corresponds to an open-close pair followed by a closing parenthesis, '())'. Thus to compute *leafrank*(x) we need to find $rank_p(x)$, where p is the pattern ()), in the parenthesis sequence corresponding to the tree (recall that a node x is denoted by its corresponding left parenthesis in the parenthesis representation). Similarly *leafselect*(j) is nothing but $select_p(j)$ where p is the pattern ()). Also *leafsize*(x) is the difference between $rank_p(x)$ and

$rank_p(c(x))$ where $c(x)$ denotes the position of the closing parenthesis corresponding to the parent of x. Hence from Theorem 2.2.1, these operations can be supported in constant time.

The leftmost leaf of the subtree rooted at a node in the binary tree is the leaf whose *leafrank* is one more than the *leafrank* of the given node. Thus it can be found in constant time using the expression: $leftmost(x) = select_p(rank_p(x) + 1)$, where p is the pattern $())$. The rightmost leaf of a subtree rooted at the node in the binary tree is the rightmost leaf of its parent in the general tree. Now, the rightmost leaf of a node in the general tree is the leaf preceding the closing parenthesis of the given node. Thus, $rightmost(x) = select_p(rank_p(close(parent(x)) - 1))$, where *close* gives the position of the corresponding closing parenthesis of a given opening parenthesis, which takes constant time for evaluation using the *rank* and *select* operations. ∎

2.2.4 Putting things together for the Suffix Tree Representation

Given a string $x\$$ of length n, we encode all its suffixes in binary using an encoding that assigns a bit string of length $k = \lceil \lg(|\Sigma| + 1) \rceil$ for each character in the alphabet and the character \$ as explained in Section 2.2.2. We then construct a compressed trie for them, which will be a binary tree on $2n - 1$ nodes. We represent this $2n - 1$ node binary trie with $4n + o(n)$ bits using the representation given in Section 2.2.3. (We could in fact represent the tree using only $2n + o(n)$ bits by storing only the internal nodes of the tree. Since all the internal nodes have two children, we can associate the external positions of this tree with the leaves of the original tree, in order. But listing the external nodes explicitly has some advantages for later modifications.) This will take care of the storage for the first component of suffix tree representation (Section 2.2.2).

We follow the convention that the edge pointing to the left child always has label starting with a 0 and the one to the right child has label starting with a 1. This eliminates the need for storing the first character of the edge label explicitly. Since the *leafsize* operation gives, in constant time, the number of leaves rooted at a given internal node, we need not store the fourth component. Next, we show that the third component of the representation, the skip values, need not be stored explicitly and that they can be obtained online whenever needed. So only the fifth component taking $n \lceil \lg n \rceil$ bits accounts for the higher order term and we get an $n \lg n + O(n)$ bit suffix tree structure.

Now we explain how we can find the skip value at a node without storing it explicitly. Given an internal node v in the suffix tree, let L_v denote the longest common prefix of all the suffixes, in binary, associated with the leaves in the subtree rooted at v. This is nothing but the concatenation of the edge labels (encoded in binary) in the path from root to the node v. Let $l(v)$ be the length of L_v. Note that $l(v)$ can be computed using the expression, $l(v) = \mathit{lcp}(\mathit{leftmost}(v), \mathit{rightmost}(v))$, where $\mathit{lcp}(x, y)$ returns the length of the longest common prefix of the strings x and y. We first observe that the skip value associated with an internal node v is nothing but $l(v) - l(\mathit{parent}(v)) - 1$. Thus, to find the skip value at an internal node v, we first go to the leftmost and rightmost leaves in the subtree rooted at v and start comparing the text starting at these positions until there is a disagreement. We don't have to compare the suffixes from the starting position. We already know that L_v is a common prefix of these strings. So we can start matching them from position $l(v) + 1$. The number of bits matched is the skip value at that node. Finding the leftmost or rightmost leaf of the subtree rooted at a node, in the tree representation takes constant time using the $\mathit{leftmost}(x)$ and $\mathit{rightmost}(x)$ operations of Theorem 2.2.2.

To search for a pattern, we start at the root as before and start matching the pattern with a path in the suffix tree. Navigating in the suffix tree is possible using the tree representation. Note that, from an internal node it is always possible to continue the search as each internal node has both left and right children whose edge labels start with the characters 0 and 1 respectively. At internal nodes, we don't skip any bits in the pattern, but actually match portion of the compressed string with the pattern. The search algorithm stops when either the pattern is matched with string L_v corresponding to a node v or the pattern does not match the portion of a skipped string.

If the pattern does not match the prefix of a skipped string, then there is no occurrence of the pattern in the given text string. Otherwise, if the end of the pattern is encountered, then all the suffixes in the subtree rooted at the node (which could be a leaf) at which the search has ended match the pattern. Once we confirm that the pattern exists in the text, the number of leaves in the subtree rooted at the node where the search ended, gives the number of occurrences of the pattern in the text.

This can be found in constant time using the *leafsize* operation of Theorem 2.2.2. Also, we can output all the occurrences of the pattern in time linear in the number of occurrences (after confirming that the pattern exists in the text) as follows: Let v be the node at which the search has ended. Find $i = \mathit{leafrank}(\mathit{leftmost}(v))$. The i^{th} element in the suffix array gives the leftmost starting point of an occurrence of the pattern in the text. Output all the values in the suffix array from i to $i + \mathit{leafsize}(v)$.

The time to find a skip value, k or the skip (bit) string is $O(k)$. The sum of the skip values computed during the search for a pattern is at most the length of the pattern (i.e., at most $m \lg |\Sigma|$). So, the total time spent in figuring out skip values is only $O(m \lg |\Sigma|)$. Looking carefully at the calls to *rank* and *select* which are implicit in this approach, one can see that we have at most increased the search cost by a constant factor by getting rid of the storage required for the skip values. This increase is due to the repeated *leftmost* and *rightmost* calls.

Thus we have

Theorem 2.2.3 *A suffix tree for a text can be represented using $n \lg n + O(n)$ bits in which one can answer search and counting queries in $O(m \lg k)$ time where k is the size of the alphabet. Enumerative queries require an additional $O(occ)$ time.*

The above representation can be built in $O(n)$ time as once we build the suffix tree which takes $O(n)$ time [Wei73], the succinct tree representation can be built in $O(n)$ time.

Instead of converting the suffixes to binary we could also directly represent the k-ary suffix tree using a succinct representation of a k-ary cardinal tree. We will explore this in Chapter 3, to get another implementation of a suffix tree.

2.3 Succinct Index Structures for Binary Texts

In this section, we consider the case when the given text is over a binary alphabet. Note that for the binary alphabet case, one can support search and count queries efficiently, by simply performing a binary search on the suffix array (i.e., the array of pointers to the suffixes, stored in the lexicographic order). Using the power of the word RAM to read and compare $O(\lg n)$ bits in constant time, one can compare the pattern (of length m) with any suffix of the text in $O(m/\lg n)$ time. Since there are n suffixes in the suffix array, a binary search on the suffix array requires $O(\lg n)$

comparisons of the pattern with the suffixes. Thus, this gives us a simple structure that takes $n\lceil\lg n\rceil$ bits of space and supports search and count queries in $O(m)$ time. This section shows that we can do even better by presenting two indexing structures that take less than $n\lg n$ bits of space.

2.3.1 A Structure using $\frac{n}{2}\lg n + O(n)$ Bits

Here, we give an index structure that takes at most $\frac{n}{2}\lg n + O(n)$ bits of space for a given binary string of length n and supports search queries in $O(m)$ time. But this structure does not support counting and enumerative queries efficiently.

The main idea is to store only the suffixes starting with either a 0 bit or a 1 bit, whichever bit appears less number of times in the given bit string and store some extra information to aid searching for patterns starting with the other bit.

Without loss of generality, suppose there are more number of 0's than 1's in the given bit string $T[1 \ldots n]$. The structure consists of a sparse suffix tree for all the suffixes starting with 1. (Here, first we take all the suffixes with the end-markers, convert them into binary and then construct the suffix tree so that the resulting suffix tree is a binary tree with at most $n/2$ leaves.) We order the subtrees of a node such that the left subtree contains a leaf which has at least as many consecutive zeroes preceding its starting position as any other leaf in that subtree. In other words, the leftmost leaf of any node has the maximum number of consecutive zeroes preceding it, among all the leaves of the subtree rooted at that node.

Since the order of the subtrees is not the same at each node, we store one bit for each internal node indicating whether or not its subtrees are interchanged with respect to the original ordering (to indicate whether the label of the edge pointing to the left child starts with a 0 or a 1). This requires $O(n)$ bits of space. We keep these bits in a separate bit vector and index into this bit vector using the internal node numbering obtained from the tree representation. The space occupied by the sparse suffix tree (using the representation given in Section 2.2.4) is at most $\frac{n}{2}\lg n + O(n)$ bits as there are at most $n/2$ suffixes starting with 1, since there are more zeroes than ones.

Now given a pattern, if it starts with a 1, we can use the sparse suffix tree directly to find its occurrences. Otherwise, let the pattern be $0^l x$ for some integer $l \geq 1$, where the first bit of x is a 1. Search for x in the sparse suffix tree. If the search fails then the given pattern does not exist in the text. Otherwise, let v be the node at which the search has ended (i.e., the node corresponding to the string x in the sparse suffix tree) and let i be the position pointed to by the leftmost leaf of v. If $T[i-l \dots i-1]$ is identical to 0^l, then the given pattern occurs at position $i-l$ in the text. Otherwise, there is no occurrence of the pattern in the given text T. The total time taken by the search procedure is $O(m)$. Thus we have,

Theorem 2.3.1 *Given a binary text of length n, there exists an index structure that uses $\frac{n}{2}\lg n + O(n)$ bits of space and supports search queries in $O(m)$ time.*

But this structure can not be used to report all the occurrences efficiently, in general. If the given pattern starts with a 1 bit, then all the leaves in the subtree rooted at the node at which the search for the pattern has ended (successfully) will give the positions of all the occurrences of the pattern in the text. Thus in this case, enumerative queries can be answered with an additional $O(occ)$ time. But if the given pattern is of the form $0^l 1x$ for some $l \geq 1$, then we first search for the pattern $1x$ in the sparse suffix tree. If the search ends successfully at node v, then one can list all the occurrences of the pattern in the text by visiting each of the leaves in the subtree rooted at v and listing a leaf labeled i if $T[i-l \dots i-1]$ is identical to 0^l. One can speed up this process by skipping all the nodes to the right of a leaf i in the subtree rooted at a node for which i is the leftmost leaf. But this, in general, will not improve the asymptotic complexity of the process.

2.3.2 A Structure using $o(n \lg n)$ Bits

In this section we develop a structure that takes $O(n \lg n / \lg\lg n)$ bits of space for a given binary text of length n and answers existential queries in $O(m)$ time. The structure does not support search and counting queries efficiently, in general, though in some cases it is possible to support these queries in constant time.

The structure consists of a sparse suffix tree and two tables. The first table stores, for all binary strings of length at most b (to be fixed later), whether it appears as a substring in the given text string. This precomputed table is used to find whether patterns of length at most b are present. A sparse suffix tree is constructed for every b^{th} suffix of the given text string, i.e., suffixes starting at positions $ib+1$, $0 \leq i \leq n/b - 1$. If the length of the text is not a multiple of b, we consider the text with the required number of 0's inserted in the beginning to make it a multiple of

b. We interpret the given binary text of length n as a string of length n/b over a 2^b-ary alphabet and construct a suffix tree for this string.

This suffix tree is stored using a simple cardinal tree representation given in [Ben98, BDMR99]. (A cardinal tree of degree k is a rooted tree in which each node has k positions for an edge to a child. See Chapter 3 for more details.) In this representation, a cardinal tree of degree k with n nodes can be stored using $nk + o(n)$ bits and the tree navigational operations (like finding the parent or the i^{th} child of a node) can be supported in constant time. In our case, the degree is 2^b and the number of nodes is $O(n/b)$. Thus the space required to store this suffix tree is $O(2^b n/b)$ bits.

The second table is indexed by a node v in the sparse suffix tree and two strings p and s of length at most b. The table stores a 1 if there exists a leaf in the subtree rooted at node v which points to a suffix such that $L_v s$ is a prefix of it and the substring p precedes that suffix in the given text, and stores a 0 otherwise; here L_v is the string obtained by concatenating the edge labels in the path from root to the node v, as defined in Section 2.2.4.

The space occupied by the first table is 2^b bits. The space occupied by the sparse suffix tree is $O(n \lg n/b)$ bits for the leaf pointers and $O(2^b n/b)$ bits for the tree representation. The space required to store the second table is $O(\frac{n}{b}2^{2b})$ bits. Thus, choosing b to be $\frac{1}{2}\lg\lg n$ makes the overall space occupancy of the structure to be $O(n \lg n/\lg\lg n)$ bits.

To search for a given binary pattern p of length m, if its length is at most b then we can know the answer from the first table. Otherwise, we will repeat the following search procedure b times, varying i from 0 to $b-1$.

Let p_i be the prefix of p of length i bits and s_i be the suffix of p of length $(m - i - \lfloor\frac{m-i}{b}\rfloor b)$ bits. Match the substring of p of length $(\lfloor\frac{m-i}{b}\rfloor)$ characters (where each character is of length b bits) starting at bit position i, in the suffix tree. If there is no match in the suffix tree then skip the current iteration. Otherwise, let v be the node at which the match has ended. If the search ends in an edge, take v to be the node pointed to by the edge. If the length of L_v (defined in Section 2.2.4) is equal to $(\lfloor\frac{m-i}{b}\rfloor b)$ bits, then find table entry corresponding to node v, prefix p_i and suffix s_i. If it is 1, then output *yes* and halt; otherwise skip the current iteration. If L_v has length more than $(\lfloor\frac{m-i}{b}\rfloor b)$ bits (this happens when v is a compressed node), then find the $(\lfloor\frac{m-i}{b}\rfloor + 1)^{st}$ character, say x, in L_v (a character here is an element of a 2^b sized alphabet and hence corresponds to a b bit string) and check if s_i is a prefix of x (note that s is a binary string of length at most b). If not skip the current iteration. Otherwise find table entry corresponding to node v, prefix p_i and suffix λ (the empty string). If it is 1, then output *yes* and halt; otherwise skip the current iteration. Finally, if we have not answered *yes* in any of the b iterations, then we answer *no*.

Each iteration of the above search procedure takes $O(m/b)$ time and thus the total time for searching for a pattern in this structure is $O(m)$, where m is the length of the pattern. Thus we have

Theorem 2.3.2 *Given a binary text of length n, there exists an index structure that uses $O(n \lg n / \lg\lg n)$ bits of space and supports existential queries in $O(m)$ time.*

The first table, which stores whether a pattern of length at most b occurs in the text, can also be used to store a pointer to an occurrence and also the number of occurrences of the pattern using $o(n)$ bits of space. Thus, when the pattern length is at most $\epsilon \lg n$ for some fixed positive constant $\epsilon < 1$, it is possible to support the search and counting queries in constant time.

We have presented two index structures for binary texts, one which supports search queries and another which supports existential queries, in $O(m)$ time. Note that for binary texts, the lower bound on query time is $\Omega(m/\lg n)$ in the word RAM model with word size $O(\lg n)$, as we need $\Omega(m/\lg n)$ time to read the pattern.

Also, till recently, the $\Omega(n \lg n)$ bits of space to store the pointers to the starting positions of the suffixes, was considered unavoidable for any data structure based on searching the set of all suffixes [Kär99] (that answers search queries in $O(m)$ time). But Grossi and Vitter [GV00] have given an indexing structure that takes $O(n)$ bits of space for a binary string of length n which answers search and counting queries in $o(m)$ time. However, for supporting enumerative queries, this structure requires $O(m/\lg n + occ\ \lg^{\epsilon} n)$ time, for any fixed positive constant $\epsilon < 1$. Their main contribution is an $O(n)$ bit representation of a *compressed suffix array* for a given binary text.

In the next section, we give a structure that improves the structures given in the last two sections, both in terms of time and space, using the ideas of Grossi and Vitter [GV00]. We obtain the first $o(n \lg n)$ bit structure that supports enumerative queries in $O(m/\lg n + occ)$ time.

2.4 Compressed Suffix Arrays

In this section, we look at some space efficient implementations of another well known indexing structure called *suffix array*. We also describe how this can be used to obtain a space efficient suffix tree representation.

2.4.1 Definitions and Background

Definition 2.4.1 *A suffix array for a text string T of length n is an array of integers in the range 1 to n, specifying the lexicographic order of the n suffixes of string T.*

Gonnet et al. [GBYS92] propose this lexicographically ordered array of suffixes of a string as a useful data structure for solving string processing problems. Using this array one can search for a pattern of length m in $O(m \lg n)$ time using a simple binary search on the suffixes of the text, where n is the length of the text. Manber and Myers [MM93] augment this structure with longest common prefix information of some specific suffixes in the array, and they named it the 'suffix array'. This fairly standard representation of suffix array (along with the augmented structure) takes about $2n$ words of space [CC96]. Given the suffix array for a text string of length n, searching for a pattern of length m is done using a binary search and can be performed in $O(m + \lg n)$ time in the worst case (in the case of a binary text, one can actually support search in $O(m)$ time, as explained in Section 2.3).

In this section, by a suffix array we mean just the list of pointers to the suffixes in their lexicographic order, without the augmented array of longest common prefixes. We adopt the convention [GV00] that the given text T is a binary string of length $n-1$ over the alphabet $\{a, b\}$, and is terminated in the n^{th} position by a special symbol '#', and that the symbols are ordered as $a < \# < b$. This ordering was chosen to have the one-one correspondence between the text and its suffix array. (If we choose any of the other two orderings, for example, then the strings $a^{n-1}\#$ and $b^{n-1}\#$ will have the same suffix array.) Let SA be the suffix array of T; that is $SA[i]$ is the position in T of the i^{th} suffix in the lexicographic order of the suffixes.

Given a text of length n, its suffix array can be stored using $n \lceil \lg n \rceil$ bits by explicitly storing all its entries in an array. This can be used to retrieve the i^{th} entry in constant time. But the disadvantage of this structure is that this uses more space than necessary (as explained below).

Given a suffix array of a binary text, one can recover the original text from it. Thus, there is a one-to-one correspondence between the binary strings and their suffix arrays. Since the number of binary strings of length n is 2^n, information theory gives an $n-1$ bit space lower bound for representing a suffix array (by the previous assumption that the last character is fixed).

Given a binary text, we define its *compressed suffix array* as a data structure that represents the suffix array for the text to support the following operation efficiently:

lookup(i)**:** return $SA[i]$, i.e., the position in T of the i^{th} suffix in the lexicographic order.

Grossi and Vitter [GV00] have given two implementations of compressed suffix arrays, one that takes $O(n)$ bits of space and $O(n)$ preprocessing time so that each call to *lookup* takes $O(\lg^{\epsilon} n)$ time, for any fixed constant $0 < \epsilon \leq 1$, and another that takes $O(n \lg\lg n)$ bits and $O(n)$ preprocessing time, so that calls to *lookup* take $O(\lg\lg n)$ time. One can easily generalize their implementations to one that takes $O(nt)$ bits of space and supports *lookup* queries in $O(t(\lg n)^{1/t})$ time for any parameter $1 \leq t \leq \lg\lg n$. We give another implementation that takes $O(nt(\lg n)^{1/(t+1)})$ bits of space and supports *lookup* queries in $O(t)$ time for any parameter $1 \leq t \leq \lg\lg n - 1$. In particular this gives a compressed suffix array representation taking $o(n \lg n)$ bits that supports *lookup* in constant time.

Given a suffix array SA, define the functions, Ψ_k, for $k \geq 0$, as follows:

$$\Psi_k(i) = \begin{cases} j \text{ if } SA[j] = SA[i] + k \\ \\ 0 \text{ if } SA[i] + k > n \end{cases}$$

This is a simple generalization of the function Ψ defined by Grossi and Vitter [GV00], which 'corresponds' to the function Ψ_1.

We denote a sequence $s_1 s_2 s_3 \ldots s_n$ by $\{s_i : 1 \leq i \leq n\}$.

2.4.2 Our Representation

Now, we outline a method for representing a suffix array. Assume that the length of the text T is a multiple of l, where l is some parameter to be fixed later; otherwise append T with # symbols so that its length becomes a multiple of l. (This does not effect the asymptotic analysis of the structure as explained later).

1. Store those values in the suffix array SA which are multiples of l in another array SA_1 in the same order as they appear in SA, after dividing them by l.

2. Store a bit vector B of n bits, such that $B[i] = 1$ if $SA[i]$ is a multiple of l (i.e., if $SA[i]/l$ is stored in SA_1) and $B[i] = 0$ otherwise. Also store a rank directory for this bit vector (described in Section 2.2.3).

3. For each i, $1 \leq i \leq n$ store the value $l - (SA[i] \bmod l)$, that is the difference of $SA[i]$ from the next multiple of l, in an array d.

4. For each k, $1 \leq k \leq l - 1$ store the compressed representation of the subsequence $\{\Psi_k(i) : d[i] = k\}$, as described later in Section 2.4.3.

See Figure 2.4.4(a) for an example.

Now, given i, we can recover $SA[i]$ from the compressed representation as follows:

First, note that if the length of the text differs from the next multiple of l by a value r, then we would have appended r # symbols to the given text. All the suffixes starting with these symbols will occur in the middle of the suffix array, before any suffix starting with 1 and after every suffix starting with 0. Suppose there are n_0 zeroes in the given text. Then, then i^{th} suffix in the given text will be the i^{th} suffix in the appended text also, if $i \leq n_0$. Otherwise, if $i > n_0$, then the i^{th} suffix in the given text will be the $(i + r)^{th}$ suffix in the appended text. Thus, we can appropriately translate the operation by storing the values of r and n_0, using $O(\lg n)$ bits.

Let $k = d[i]$. Find $\Psi_k(i)$ from the representation of Ψ_k in Section 2.4.3 (note that $\Psi_0(i) = i$, from the definition). Find the rank r of $\Psi_k(i)$ using the rank directory for B. It is easy to see that $SA[i] = l * SA_1[r] - k$.

The array SA_1 can be stored using $\frac{n}{l}\lceil \lg(\frac{n}{l}) \rceil$ bits of space. The bit vector B and its rank directory occupy $n + o(n)$ bits of space. The array d requires $n \lceil \lg l \rceil$ bits. So the space requirement for the above compressed suffix array representation is $\frac{n}{l}\lceil \lg(\frac{n}{l}) \rceil + n \lceil \lg l \rceil + n + o(n)$ bits plus the space required to store the compressed representations of $\{\Psi_k(i) : d[i] = k\}$, for $1 \leq k \leq l - 1$.

2.4.3 Representing Ψ_k Compactly

The following lemmas are used in representing the Ψ_k function compactly.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T :	***a***	***b***	***b***	***a***	***b***	***b***	***a***	***b***	***b***	***a***	***b***	***b***	***a***	***b***	***a***	***a***	***a***	***b***	***a***	***b***	***a***	***b***	***b***	***a***	***b***	***b***	***b***	***a***	***b***	***b***	***a***	#
SA :	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25
B :	0	1	0	0	0	0	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
d :	1	0	1	3	3	1	0	2	1	0	3	3	0	0	2	2	0	2	1	3	2	1	0	1	3	1	2	0	3	2	2	3

	1	2	3	4	5	6	7	8
SA_1 :	4	7	1	6	8	3	5	2

(a)

$$\Psi_1 : 2\ 14\ 23\ 28\ 7\ 10\ 13\ 17$$

$$L_0 = \{2, 14, 23, 28\};\ L_1 = \{7, 10, 13, 17\}$$

$$\Psi_2 : 17\ 2\ 14\ 23\ 28\ 7\ 10\ 13$$

$$L_0 = \Phi;\ L_1 = \{17\};\ L_2 = \{2, 14, 23, 28\};\ L_3 = \{7, 10, 13\}$$

$$\Psi_3 : 2\ 23\ 10\ 13\ 17\ 14\ 28\ 7$$

$$L_0 = \Phi;\ L_1 = \Phi;\ L_2 = \{2, 23\};\ L_3 = \{10, 13\};$$

$$L_4 = \Phi;\ L_5 = \{17\};\ L_6 = \{14, 28\};\ L_7 = \{7\}$$

(b)

Figure 2.4.4: Compressed suffix array representation

Lemma 2.4.1 *Given a subset S of size n from the universe $U = \{1, \ldots, m\}$, $n \leq m$, it can be represented using $n \lg(m/n) + O(n)$ bits such that given i, $1 \leq i \leq n$, the i^{th} smallest element in S can be found in constant time.*

Proof. Represent the top $\lceil \lg n \rceil$ bits of all the elements in the increasing order using a bit vector of length $3n$. This is done as follows: Consider the sequence obtained by taking the top $\lceil \lg n \rceil$ bits of all the elements in the increasing order. This will be a nondecreasing sequence of n elements from the set $[2n]$. This sequence is then represented as a bit vector B by storing a 0 followed by j 1s, for $1 \leq i \leq 2n$, where j is the multiplicity of number i in the sequence. Also store the rank and select directories (of Section 2.2.3) for B using $o(n)$ bits. Store the remaining $\lceil \lg m \rceil - \lceil \lg n \rceil$ bits of each element in an array A in the increasing order of the elements.

To find the i^{th} smallest element using this representation, we first find its top $\lceil \lg n \rceil$ bits from the bit vector B using its rank and select directories. More specifically, we compute $rank_0(select_1(i))$ in the bit vector B, which gives the top $\lceil \lg n \rceil$ bits of the i^{th} element. We then, find the remaining $\lceil \lg m \rceil - \lceil \lg n \rceil$ bits from the i^{th} element of the array A. ∎

Corollary 2.4.2 *Let $L = L_0 \cdot L_1 \cdot \ldots \cdot L_{k-1}$ be the sequence obtained by concatenating the lists $L_0, L_1, \ldots, L_{k-1}$, where each L_i is a sorted sequence of n_i numbers in the range $[1, \ldots, m]$ and let $\sum_{i=0}^{k-1} n_i = n$. Then the sequence L can be represented using $n \lg(mk/n) + O(n)$ bits such that given i, the i^{th} element in L can be retrieved in constant time.*

Proof. Store the set $S = \{j * m + x | x \in L_j\}$ using the representation of Lemma 2.4.1, which uses $n \lg(mk/n) + O(n)$ bits of space. Then the i^{th} element in the sequence L is the same as the i^{th} smallest element in the set S, which can be found in constant time using the representation. ∎

Lemma 2.4.3 *The sequence $\{\Psi_k(i) : 1 \leq i \leq n\}$ is the concatenation of 2^k sorted lists $L_0^k, L_1^k, \ldots L_{2^k-1}^k$, where $L_j^k = \{\Psi_k(i)$: the value in binary of the k symbols appearing before the position $SA[\Psi_k(i)]$ in the text is equal to $j\}$.*

Proof. First we will show that each of the lists L_j^k is sorted. To show this, it is enough to show that: if the suffixes starting at the positions $SA[\Psi_k(i)]$ and $SA[\Psi_k(j)]$ in the text have the same k symbols preceding them, then $i < j \Rightarrow \Psi_k(i) < \Psi_k(j)$.

Suppose not, i.e., for $i < j$ let $SA[\Psi_k(i)]$ and $SA[\Psi_k(j)]$ have the same k symbols preceding them and $\Psi_k(i) > \Psi_k(j)$. Since $SA[\Psi_k(i)] = SA[i] + k$ (unless $SA[i] + k > n$), it follows that the positions $SA[i] + k$ and $SA[j] + k$ have the same k symbols preceding them in the text and the suffix starting at position $SA[i] + k$ is lexicographically larger than the suffix starting at position $SA[j] + k$. But this implies that the suffix starting at position $SA[i]$ is lexicographically larger than the suffix starting at position $SA[j]$ (as they have the same first k symbols), which is a contradiction since SA is a suffix array and $i < j$.

Now, to show that the sequence $\{\Psi_k(i) : 1 \leq i \leq n\}$ is the concatenation of the lists $L_0^k, L_1^k, \ldots, L_{2^k-1}^k$ (i.e., each element in L_i^k appears before any element in L_j^k in the Ψ_k sequence, for $i < j$), it is enough to show the following: If $\Psi_k(i_1) \in L_{j_1}^k$ and $\Psi_k(i_2) \in L_{j_2}^k$, for $j_1 \neq j_2$, then $i_1 < i_2 \Rightarrow j_1 < j_2$. Now $i_1 < i_2$ implies that the suffix starting at position $SA[i_1]$ in the text is lexicographically smaller than the suffix starting at position $SA[i_2]$. Hence, the value of the k symbols appearing before the suffix starting at position $SA[\Psi_k(i_1)]$ is less than the value of the k symbols appearing before the suffix starting at position $SA[\Psi_k(i_2)]$ (since they are not equal as $\Psi_k(i_1)$ and $\Psi_k(i_2)$ belong to different lists), which implies $j_1 < j_2$. ∎

Remark 2.4.4 The same proof goes through even if we take only a subsequence of Ψ_k. Suppose we take all the multiples of a number p in SA in the order they appear and store them in an array SA_1 after dividing each element by p. Then the sequence, $\{\Psi_k(i) : 1 \leq i \leq n/p\}$, defined on the array SA_1 can be partitioned into 2^{kp} sorted lists, since the sequence $\{\Psi_k(i) : 1 \leq i \leq n/p\}$ for the array SA_1 'corresponds' to the subsequence $\{\Psi_{kp}(i) : SA[i] \text{ is a multiple of } p, 1 \leq i \leq n\}$ of the sequence $\{\Psi_{kp}(i) : 1 \leq i \leq n\}$ for the array SA.

Now we show how to represent the subsequences of Ψ_k compactly. There are exactly n/l values, where each value is in the range $[1, \ldots, n]$, to be stored in each Ψ_k (in item 4 of the representation of suffix array in Section 2.4.2), namely the sequence $\{\Psi_k(i) : d[i] = k\}$, assuming n is a multiple of l. As observed above, each Ψ_k can be partitioned into 2^k sorted lists, $L_0^k, L_1^k, \ldots, L_{2^k-1}^k$. See Figure 2.4.4(b) for an example.

So the subsequence of Ψ_k that is to be stored, in item 4 of the representation of a suffix array in Section 2.4.2, can be stored as follows:

- Store a bit vector V_k such that $V_k[i] = 1$ if $d[i] = k$ and 0 otherwise, for $1 \leq i \leq n$. Store the rank directory for this bit vector.

- Store the concatenation of the lists $L_0^k, L_1^k, \ldots, L_{2^k-1}^k$ (which is precisely the sequence $\{\Psi_k(i) : d[i] = k\}$) using the representation of Corollary 2.4.2.

Given i, we can find $\Psi_k(i)$, if $d[i] = k$, as follows: Find $r = rank_1(i)$ in the bit vector V_k using its rank directory. Now find the r^{th} element in the sequence $L_0^k L_1^k \ldots L_{2^k-1}^k$ using its representation, which gives the value of $\Psi_k(i)$.

The space required to store the bit vector V_k and its rank directory is $n + o(n)$ bits. Representation of the lists $L_0^k, L_1^k, \ldots, L_{2^k-1}^k$ (using Corollary 2.4.2) requires $\frac{n}{l}\lg(2^k n/(n/l)) = \frac{n}{l}(k + \lg l)$ bits. Thus the space required to store compressed representations of Ψ_k for $1 \leq k \leq l-1$ is $O(nl)$ bits.

Thus, using this compressed representation of Ψ_k in the suffix array representation, we get

Theorem 2.4.5 *There is an implementation of a compressed suffix array that takes* $\frac{n}{l}\lceil\lg(\frac{n}{l})\rceil + O(nl)$ *bits of space and supports lookup queries in constant time, for any parameter* $1 \leq l \leq \lg n$.

Choosing $l = \sqrt{\lg n}$, we get

Corollary 2.4.6 *There is an implementation of a compressed suffix array that takes* $O(n\sqrt{\lg n})$ *bits of space and supports lookup queries in constant time.*

We generalize this approach in the following section.

2.4.4 A Recursive Structure

The array SA_1 in the above representation can again be compressed using the same technique used to compress SA in the previous section. We store all the multiples of l in SA_1 in another array SA_2 and construct the Ψ_k's for $1 \leq k \leq l-1$. But in this case, each Ψ_k splits into 2^{kl} sorted lists. Since there are n/l^2 different entries in each Ψ_k, the space required to store the representations of all the Ψ_k's is again $O(nl)$ bits. We will continue this up to t levels, each level occupying $O(nl)$ bits of space. Thus the space required to store the compressed representations in all the t levels is $O(tnl)$ bits. At the end of t levels, we will store the remaining 'suffix array' SA_{t+1} explicitly. Since there are n/l^t entries left after t levels, this requires $\frac{n}{l^t}\lg(n/l^t)$ bits. Hence the overall space required by this recursive structure is $\frac{n}{l^t}\lg(n/l^t)+O(tnl)$ bits. Clearly, this representation can be used to support *lookup* queries in $O(t)$ time, as we need to spend a constant time at each of the t levels.

Choosing $l = (\lg n)^{1/(t+1)}$, we get

Theorem 2.4.7 *There is an implementation of a compressed suffix array that takes $O(tn(\lg n)^{1/(t+1)})$ bits of space and answers lookup queries in $O(t)$ time, for any parameter $1 \leq t \leq \lg\lg n - 1$.*

In particular for $t = \lg\lg n - 1$ we get a compressed suffix array representation that takes $O(n\lg\lg n)$ bits which can be used to answer *lookup* queries in $O(\lg\lg n)$ time. This scheme is the same as one of the schemes presented in [GV00].

By choosing $t = (1/\epsilon) - 1$ in the above theorem, we get

Corollary 2.4.8 *There is an implementation of a compressed suffix array that takes $O(n\lg^\epsilon n)$ bits of space and answers lookup queries in $O(1)$ time, for any fixed positive constant $\epsilon \leq 1$.*

2.4.5 Application to Suffix Tree Representation

By replacing the suffix array with the representation of Corollary 2.4.8, in the suffix tree structure of Theorem 2.2.3, we get an $O(n\lg^\epsilon n)$ bit suffix tree representation for a binary text of length n that supports search queries in $O(m)$ time and enumerative queries in $O(m + occ)$ time.

Using their $O(n)$ bit compressed suffix array representation, Grossi and Vitter [GV00] have given an index structure that takes $O(n)$ bits of space and supports search and counting queries in $O(m/\lg n + \lg^\epsilon n)$ time and enumerative queries in $O(m/\lg n + occ\lg^\epsilon n)$ time, for any fixed positive constant $\epsilon < 1$. This structure uses the suffix tree structure of Theorem 2.2.3 with some modifications (with the suffix array replaced by their compressed suffix array representation). We first briefly describe these modifications.

The main idea is to use a perfect hash function h to skip $O(\lg n)$ nodes in the trie in constant time, by storing shortcut links. The nodes of the trie are enumerated in preorder starting from the root. Then they build hash tables in which the pair $\langle j, b\rangle$ is stored at position $h(i, x)$ where node j is a descendent of node i, string x is of length at most $\lg n$ and b is a non-negative integer. These parameters must satisfy the following conditions:

- j is the node identified by starting out form node i and traversing downward the trie according to the bits in x.
- b is the unique integer such that the string corresponding to the path from i to j has prefix x and length $|x| + b$; this condition does not hold for any ancestor of j.

In order to reduce the number of shortcut links, they set up two hash tables T_1 and T_2. The first hash table T_1 stores entries $T_1[h(i,x)] = \langle j, b\rangle$ such that all strings x are of length $\lg n$. Initially, all possible shortcut links from the root are created. Then, the shortcut links from each of the descendents are recursively created. The second table T_2 is created analogously with strings of length $\sqrt{\lg n}$. The number of shortcut links in each of these tables is upper bounded by the number of nodes in the trie.

To search for a pattern in this trie augmented with the shortcut links, they take the first $\lg n$ bits in the pattern and branch from the root using T_1. If the hash lookup in T_1 succeeds and gives pair $\langle j, b\rangle$, then they try to match the next b bits in the pattern in $O(1 + b/\lg n)$ time and then recursively search in node j with the remaining pattern. If the hash lookup fails because there are fewer than $\lg n$ bits left in the pattern, they switch to T_2 and take only the next $\sqrt{\lg n}$ bits in the pattern to branch further. Finally, when the branching fails again, they match the remaining at most $\sqrt{\lg n}$ bits in the standard way, one bit at a time. Thus, a search query requires $O(m/\lg n + \sqrt{\lg n})$ time using this structure. In any case they require $O(\lg^\epsilon n)$ time to index into the suffix array (using their compressed suffix array implementation). Thus, this structure requires $O(m/\lg n + occ \lg^\epsilon n)$ time to support enumerative queries.

We now describe our modifications to the structure of Grossi and Vitter, to get a structure that supports enumerative queries in optimal time.

We replace their compressed suffix array representation by the representation of Corollary 2.4.8 so that we can index into the suffix array in constant time. To improve the search time, we store another precomputed table T_3 which stores for each possible trie of $\sqrt{\lg n}$ nodes and for each possible pattern of length at most $\sqrt{\lg n}$, the node at which a search for the pattern would end in the trie. This table requires $2^{O(\sqrt{\lg n})}$ which is $o(n)$ bits. Using this augmented structure, we can support search queries in $O(m/\lg n)$ time and enumerative queries in $O(m/\lg n + occ)$ time. Thus we have,

Theorem 2.4.9 *There exists an indexing structure for a binary text of length n, that uses $O(n \lg^{\epsilon} n)$ bits of space, for any fixed positive constant $\epsilon \leq 1$, and supports enumerative queries in $O(m/\lg n + occ)$ time.*

Chapter 3

Static Dictionaries Supporting Rank

3.1 Introduction and Motivation

A static dictionary is a data type for storing a subset S of size n from a finite universe U of size m so that membership queries can be answered efficiently. This problem has been widely studied and various structures have been proposed to support membership in constant time [FKS84, FNSS92, BM99, Pag01a] in the extended RAM model. More specifically, Fredman et al. [FKS84] have given a structure that takes $n \lg m + O(\lg\lg m + n\sqrt{\lg n})$ bits of space and answers membership queries in constant time. The space complexity of this structure was reduced to $n \lg m + O(\lg\lg m + n)$ bits by Schmidit and Siegel [SS90]. Brodnik and Munro [BM99] have given a structure that takes $\mathcal{B} + O(\mathcal{B}/\lg\lg\lg m)$ bits of space and supports membership queries in constant time, where $\mathcal{B} = \lceil \lg \binom{m}{n} \rceil$ is the information theoretic lower bound on the space required to store any set of size n from a universe of size m. Pagh [Pag01a] has further improved the space complexity to $B + o(n) + O(\lg\lg m)$ bits. Our focus here is to also support the *rank* operation which asks for the number of elements in the set less than or equal to the given element.

Note that the general problem of supporting the *rank* for every element in the universe is equivalent to the well studied static predecessor problem, if the space allowed is $O(n)$ words. Ajtai [Ajt88] has shown that if the word length is sufficiently small (i.e., $O(\lg n)$ bits) and only $n^{O(1)}$ words of memory are used to represent any set of n elements, then worst-case constant time for predecessor queries is not possible. Some improved lower bounds have been obtained for this problem [Mil94, MNSW98] until Beame and Fich [BF99] essentially closed the problem by giving a structure that uses $n^{O(1)}$ words of space and performs predecessor queries in $\Theta\left(\min\left\{\lg\lg m/\lg\lg\lg m, \sqrt{\lg n/\lg\lg n}\right\}\right)$ time, and by providing a matching lower bound. Our focus here is to support rank queries for only the elements that are present in the given set, in constant time.

We define the *dictionary with rank* problem, which is to represent a subset S of a finite universe U so that following operation can be supported in $O(1)$ worst-case time:

rank(x)**:** Given $x \in U$, return -1 if $x \notin S$ and $|\{y \in S | y < x\}|$ otherwise.

Our motivation for studying the rank operation comes from the problem of succinct representation of a cardinal tree, which is a generalization of a binary tree (see Section 3.4 for a formal definition). The cardinal tree representation of Benoit et al. [BDMR99] uses $2n + n\lceil\lg k\rceil + o(n)$ bits of space for a k-ary cardinal tree. This structure supports all the navigational operations in constant time except finding the child of a node with a given label, which takes $O(\lg\lg k)$ time. This representation implicitly gives a static dictionary structure that supports membership and rank queries in $O(\lg\lg m)$ time, where m is the size of the universe. We first observe that, to support all the operations in constant time using the same amount of space, a dictionary with rank structure taking $n\lg m + o(n)$ bits for a set of size n over a universe of size m, suffices.

Note that using a dictionary with rank, one can determine the membership of an element in constant time by checking whether or not its rank is -1. Thus, a dictionary with rank is a generalization of a *static dictionary*, which only supports (yes/no) membership queries on S. The most space efficient static dictionary is due to Pagh [Pag01a] which requires $\lceil\lg\binom{m}{n}\rceil + o(n) + O(\lg\lg m) = n\lg(me/n) - \Theta(n^2/m) + o(n) + O(\lg\lg m)$ bits of space. However, this approach, as well as previous ones, is based on *minimal perfect hashing* [FKS84, Meh82, SS90], and does not maintain the ordering of the elements of S. Pagh [Pag01a] has also given another structure that answers membership and rank queries (for every element in the universe) in constant time using $\mathcal{B} + O(m\lg\lg m/\lg m)$ bits of space.

In the next section we give a space efficient static dictionary structure that answers membership and rank queries in constant time. We first give a structure that builds up on the enhancement of Pagh [Pag01a] of the FKS dictionary [FKS84] in Section 3.2.2. Then we use a space saving technique to remove the $O(n)$ term in the space complexity to get a structure that takes $n \lg m + O(\lg\lg m)$ bits of space and supports rank queries in constant time in Section 3.2.3. Section 3.3 gives the details of representing a set of dictionaries (over a common universe) to support rank queries on individual dictionaries. In Section 3.4, we outline the k-ary cardinal tree representation of Benoit et al. [BDMR99] and explain how the multiple dictionary structure can be used to improve the running time from $O(\lg\lg k)$ to $O(1)$ for finding the child labeled i, if exists, for any i.

In what follows, if f is a function defined from a finite set X to a finite totally ordered set Y, by $||f||$, we mean $\max\{f(x)|x \in X\}$.

3.2 Dictionary with Rank

In this section, we first describe some known static dictionary structures and then use them to obtain a dictionary with rank which takes $n \lg m + O(\lg\lg m)$ bits of space.

3.2.1 A Static Dictionary using $n \lg m + O(n + \lg\lg m)$ bits

Fredman et al. [FKS84] have given a static dictionary structure that takes $n \lg m + O(\lg\lg m + n\sqrt{\lg n})$ bits of space and supports membership in $O(1)$ time. Schmidt and Seigel [SS90] have improved this space complexity to $n \lg m + O(\lg\lg m + n)$ bits. We refer to this structure as the FKS dictionary in the later sections.

We use the following lemma due to Fredman et al. [FKS84] in reducing the universe size.

Lemma 3.2.1 ([FKS84]) *Given a set S of size n from the universe $[m]$, there exists a prime $p \leq n^2 \lg m$, such that the function $h(x) = x \bmod p$ maps the set S injectively into the set $[p]$.*

The original FKS construction to store a set S, as described by Schmidt and Siegel [SS90], has four basic steps:

1. A function $h_{k,p}(x)$ is found that maps S into $[n^2]$ without collisions. It suffices to choose $h_{k,p}(x) = (kx \bmod p) \bmod n^2$ with suitable $k < p < n^2 \lg m$, where p is a prime. Here, the values k and p depend on the set S.

2. Next, a function $h_{\kappa,r}(z)$ is found that maps $h_{k,p}(S)$ into $[n]$ so that the sum of the squares of the collision sizes is not too large. Again, it suffices to choose $h_{\kappa,r}(z) = (\kappa z \bmod r) \bmod n$, where r is any prime greater than n^2 and $\kappa \in \{1, \ldots, r-1\}$ so that $\sum_{0 \le j < n} |h_{\kappa,r}^{-1}(j) \cap h_{k,p}(S)|^2 < 3n$. We will choose r to be the smallest prime greater than n^2.

3. For each non-empty bucket i, a secondary hash function h_i is found that is one-one on the collision set. We choose $h_i(z) = (k_i z \bmod r) \bmod {c_i}^2$, where $k_i \in \{1, \ldots, r-1\}$ and c_i is the size of the collision set hashing to bucket i. The element $x \in S$, is stored in location $C_i + h_i(h_{\kappa,r}(h_{k,p}(x)))$, where $C_i = \sum_{j=0}^{i-1} {c_j}^2$. This locates all n items within a table $A^\star[1, \ldots, 3n]$ of size $3n$. The remaining $2n$ locations of A^* are set 0.

4. Finally the entries in non-zero locations of table A^* is stored (without any vacant locations) in an array $A[1, \ldots, n]$ in the same order.

The composite hash function requires the following parameters:

- k, p, κ and r for the functions $h_{k,p}$ and $h_{\kappa,r}$
- a table $K[0, \ldots, n-1]$ storing the parameters k_i for secondary hash functions h_i
- a table $C[1, \ldots, n]$ listing the locations (values) C_i and
- a compression table $D[1, \ldots, 3n]$, where $D[j]$ gives the index, within A, of the item (if any) that hashes to the value j in $A^\star$.

A straightforward representation of each of these parameters requires $O(n \lg n + \lg\lg m)$ bits of space. Schmidt and Siegel [SS90] have shown a way of storing this composite hash function using $O(n + \lg\lg m)$ bits of space. We briefly describe their representation below.

First, they observed that up to $\lfloor \lg n \rfloor + 1$ secondary hash functions are sufficient (instead of having one secondary hash function for every bucket) to hash the elements of the set. Moreover, these hash functions, stored in an array $B[1, \ldots, \lfloor \lg n \rfloor + 1]$ can be chosen in such a way that $B[1]$ is a perfect hash function for at least half the buckets, $B[2]$ is a perfect hash function for at least half of the remaining buckets and so on. (Existence of such a set of hash functions can be easily shown from the results of FKS [FKS84].) The array B takes $O((\lg n)^2)$ bits. Now, for each secondary bucket i, we need to store the index j of the hash function $B[j]$ that hashes the bucket i (to represent table K). We store this as follows: for $1 \leq i \leq n$, if the i^{th} bucket is nonempty, then store a 0 followed by j 1s if $B[j]$ hashes bucket i, and a 0 otherwise. Since index j appears at most $n/2^j$ times, for $1 \leq j \leq \lfloor \lg n \rfloor + 1$ in the above representation, the total length of this bit-string is $O(n)$. We store the *rank* and *select* directories using $o(n)$ bits (described in Section 2.2.3) for this bit vector. Using this and the array of multipliers (i.e., the secondary hash functions), given an i, we can find the multiplier (hash function) associated with the bucket i in constant time.

The parameters k and p require $O(\lg n + \lg \lg m)$ bits each and κ and r take $O(\lg n)$ bits each. The table C, which contains the values C_i, is encoded as follows. First the values c_i^2 are stored in a table T_0 in unary notation (in the order of increasing i, separated by 0's), which is of length at most $4n$. We also store the *rank* and *select* directories for this bit vector using $o(n)$ bits. Now, given an i, C_i is nothing but the rank of the i^{th} 0 (i.e., $C_i = rank_1(select_0(i))$), which can be found in constant time. For the compression table D, we store a bit-string of length $3n$ where the i^{th} bit is a 0 if $A^\star[i]$ is empty and 1 otherwise. We store the *rank* directory for this bit vector using $o(n)$ bits. When an element is hashed to a location in D, the rank of the bit in that location in the bit vector representation of D gives the location of the element in the array A.

This gives us the following:

Theorem 3.2.2 ([FKS84, SS90]) *There is a static dictionary structure that takes* $n \lg m + O(n + \lg \lg m)$ *bits of space and answers membership queries in constant time.*

3.2.2 A Dictionary with Rank taking $n \lg m + O(n + \lg\lg m)$ bits

Pagh [Pag01a] has observed that each bucket j of the hash table may be resolved with respect to the part of the universe hashing to bucket j. Thus we can save space by compressing the hash table part (i.e., table A above) of the data structure, storing in each location not the element itself, but only a *quotient* information that distinguishes it from the part of U that hashes to this location. The quotient function, slightly modified from that of Pagh's, is as follows:

$$q_{k,p}(x) = \big((x \text{ div } p). \lceil p/n^2 \rceil + (k.x \bmod p) \text{ div } n^2\big) . \lceil r/n \rceil + (\kappa.z \bmod r) \text{ div } n$$

where $z = (k.x \bmod p) \bmod n^2$ and the parameters k, p, κ and r are as defined in the description of the FKS hash function. It is easy to see that $q_{k,p}(x)$ for $x \in U$ is $O(m/n)$ (as $(\kappa.z \bmod r)$ div $n < r/n$, $(k.x \bmod p)$ div $n^2 < p/n^2$ and x div $p < m/p$).

Thus the total space required to store all the quotient values along with the hash function is $n \lg(m/n) + O(n + \lg\lg m)$ bits.

To search for a given element x, we first apply the composite hash function to determine the location to which the element x hashes to in the hash table and check whether the quotient value $q_{k,p}(x)$ appears in that location. If it appears in that location, then we answer that the element is present in the set. Otherwise, we answer that the element is not present.

This gives us the following:

Theorem 3.2.3 ([Pag01a]) *There is a static dictionary representation that takes $n \lg(m/n) + O(n + \lg\lg m)$ bits and answers membership queries in constant time.*

A common extension to the dictionary problem is that every element of the set S is associated with a *satellite* data from a set V. A membership query 'Is $x \in S$?' should then return the satellite data associated with x, if $x \in S$.

Now, with each element we can also store the satellite information associated with that element, in the hash table, using an extra $n \lceil \lg |V| \rceil$ bits of space. Thus we have

Lemma 3.2.4 *A static dictionary for a subset S of size n from the universe $[m]$, where each element is associated with a satellite data from a set V, can be stored* *using $n \lg(m|V|/n) + O(n + \lg\lg m)$ bits of space to answer membership queries in constant time.*

By storing the rank information of an element as its satellite data in the above dictionary, we get

Corollary 3.2.5 *There is a dictionary with rank for a subset of size n from the universe $[m]$ that uses $n \lg m + O(n + \lg\lg m)$ bits of space.*

3.2.3 Reducing the Space further

We now describe a method to remove the $O(n)$ term in the space bound of the above corollary. The main trick is to explicitly store only some partial rank information as a satellite information (except for a sparse number of elements for whom the full rank information is implicitly stored). This results in a saving of a linear number of bits proving the following lemma.

Theorem 3.2.6 *There is a dictionary with rank that stores a set S of size n from the universe $[m]$ using at most $n \lg m + O(\lg\lg m)$ bits of space.*

Proof. If $n \leq c$ for some constant c, to be fixed later, then list the elements of S explicitly, which takes $n \lceil \lg m \rceil$ bits of space. Using this representation, *rank* queries can be supported in constant time. Thus, assume that $n > c$.

Let $x_0 < \ldots < x_{n-1}$ be the elements of set S. Let $4 \leq r \leq n$ be an integer parameter (which will be fixed later) and let $n' = n - \lfloor n/r \rfloor - 1$.

We write down r using $\lceil \lg n \rceil$ bits and explicitly write down the keys of $B = \{x_0, x_r, x_{2r}, \ldots, x_{(n-n'-1)r}\}$ in sorted order using $(n - n') \lceil \lg m \rceil$ bits. Next, we store the set $S' = S \setminus B$ (of size n') using the dictionary of Lemma 3.2.4, with key x_i storing the satellite information $i \bmod r$ (i.e., the difference of its rank from the rank of its predecessor in B). The space occupied by this structure will be $n' \lg(mr/n') + a(n' + \lg\lg m)$ bits, for some constant a.

Thus the total space used is $n \lg m + a \lg\lg m + n'(\lg r - \lg n') + n - n' + an' + \lg n$ bits. This can be made less than $n \lg m + O(\lg\lg m)$ bits by choosing r to be $\lfloor n/2^{a+1} \rfloor$, when n is more than some constant. Choose c to be this constant. Thus, the space used in either case is $n \lg m + O(\lg\lg m)$ bits.

To answer *rank*(x) queries, we do a binary search to find the predecessor $y \in B$ of x. If $x \in B$, then the rank of x in S is r times the rank of x in B. Otherwise, we locate x in the dictionary for S'. If x is found in S', the value of the satellite data associated with x gives the difference of the rank of x from the rank of y (in S). Adding this to the rank of the y gives rank of x in S. Given an element $x \in [m]$, one can find the predecessor of x in B and also its rank in B, in constant ($O(a)$) time by performing a binary search on B, as $|B| = \lfloor n/r \rfloor + 1$ which is $O(2^a)$. Thus, *rank* queries can be supported in $O(1)$ time using this structure. ∎

Mehlhorn [Meh82] has shown that $\Omega(\lg \lg m)$ bits are needed to represent minimal perfect hash functions from $[m]$ to $[n]$. Thus there is little hope of removing the $O(\lg \lg m)$ term in the space bound altogether using the above approach (specifically, when n is $o(\lg \lg m)$). So, we use a different approach when a group of small sets are to be stored, as explained in the next section.

3.3 Multiple Rank-Dictionaries

In this section we look at the problem of representing a set of subsets from the same universe. More formally, we look at the problem of representing multiple rank-dictionaries: Given the sets $S_1, S_2, \ldots, S_s$ where each S_i is a subset of the universe $[m]$, support the operations $rank_i(x)$ which returns the rank of the element x in the subset S_i in constant time.

Theorem 3.3.1 *Let $S_1, \ldots, S_s$ all contained in $[m]$ be given sets with $\sum_{i=1}^{s} |S_i| = n$. Then this collection of sets can be represented using $n \lceil \lg m \rceil + o(n) + O(\lg \lg m)$ bits, supporting $rank_i(x)$ operation in constant time. Here $rank_i(x)$ returns the rank of the element x in the set S_i. We also assume that we have access to a constant time oracle which, given i, returns the value $\sum_{j=1}^{i-1} |S_j|$ given i.*

Proof. If we use the static dictionary representation of Theorem 3.2.6 to represent each set, then the space used by the all the dictionaries put together will be $n \lceil \lg m \rceil + O(s \lg \lg m)$ bits. More specifically, we represent the set S_i using the representation of Theorem 3.2.6 and pad it to $n \lceil \lg m \rceil + c \lceil \lg \lg m \rceil$ bits, for some constant c. Now, we concatenate the representations of the sets S_i for $1 \leq i \leq s$ (in that order). The the total space used by this representation is $n \lceil \lg m \rceil + cs \lceil \lg \lg m \rceil$ bits. Also the representation of the set S_i starts at the position $(\sum_{j=1}^{i-1} |S_j|) \lceil \lg m \rceil + (i-1)c \lceil \lg \lg m \rceil + 1$, which can be computed in constant time using the given oracle. But the total space used by this representation could be $n \lg m + \Omega(n)$ bits in the worst case.

Note that the $O(\lg\lg m)$ term for each set comes from the hash function part of the dictionary representation. One can get rid of this term when n is $\Omega(\lg\lg m)$, as explained in Section 3.2.3. But when n is $o(\lg\lg m)$ it is not possible to get rid of this term altogether (using hashing techniques), if we store each set using a separate dictionary. Hence, we use the idea of combining the sets into groups and storing a single hash function for all small sets in a group. The details are as follows:

We group the first $\lg n$ sets into the first group, the next $\lg n$ sets into the second group and so on. Thus given i, the group number to which the set S_i belongs, can be determined in constant time. We call a set S_i dense if $|S_i| \geq \sqrt{\lg n}$ and sparse otherwise.

Let $S = \cup_{i=1}^{s} S_i$. Suppose $n^2 < m$. (The case when $n^2 \geq m$ is much simpler and the details for this case are outlined at the end of this proof.) We first find a hash function f given by $f(x) = (kx \bmod p) \bmod n^2$, for some prime $p \leq n^2 \lg m$ and $k \leq p$, which maps S injectively into the set $[n^2]$. Existence of such p and k is shown by Fredman et al. [FKS84]. The values p and k can be stored using $O(\lg n + \lg\lg m)$ bits of space.

Now, to distinguish x from all the elements of the universe that map to $f(x)$ (under f), it is enough to store the following quotient information:

$$q(x) = (x \text{ div } p) \left\lceil p/n^2 \right\rceil + ((kx \bmod p) \text{ div } n^2).$$

Note that, each $q(x)$ can be stored using at most $\lg m/n^2 + 3$ bits.

Let S_i be a dense set, i.e., $n_i = |S_i| \geq \sqrt{\lg n}$. Then we store the set $f(S_i)$ using the representation described in the proof of Theorem 3.2.6, with the following modifications:

- The set B stores every r^{th} element in the sorted order of the set S_i (instead of $f(S_i)$).
- With each element $x \in S_i' = S_i \setminus B$, we now store its partial rank with respect to the set S_i (instead of $f(S_i)$), as the satellite data, in the dictionary for S_i'.

Note that the hash function in Theorem 3.2.6, in this case requires $O(n_i + \lg\lg ||f||)$ bits of space which is $O(n_i)$ bits as $||f|| = n^2$ and $n_i \geq \sqrt{\lg n}$. We also store the quotient values $q(x)$ for each $x \in S_i'$ in the sorted order of the elements of S_i.

Let ${n_i}' = |{S_i}'|$. The set B is stored using $(n_i - {n_i}' + 1)\lceil \lg m \rceil$ bits and the partial rank information for each element in the set ${S_i}'$ takes ${n_i}' \lg r$ bits. The space occupied by the dictionary for the set ${S_i}'$ will be ${n_i}' \lg(n^2/{n_i}') + an_i$ bits, for some constant a. Finally, the quotient values require at most ${n_i}'(\lg(m/n^2) + 3)$ bits of space. Thus, the total space used is $n_i \lg m + {n_i}'(\lg r + a + 3 - \lg {n_i}')$. By choosing $r = \lfloor n_i / 2^{a+4} \rfloor$, one can make the space complexity to be at most $n_i \lg m$ bits. This takes care of representing all the dense sets. We now, explain the representation of sparse groups.

Consider a group i. Let S^i be the union of all the sparse sets in the i^{th} group. Note that $|S^i| \leq (\lg n)^{3/2}$. Find a prime p_i such that the the function $g_i(x) = f(x) \bmod p_i$ is 1-1 on the set $f(S^i)$. Such a p_i whose value is at most $O(|S^i|^2 \lg ||f||)$ exists, from Lemma 3.2.1. Since $||f|| = n^2$ and $|S^i| \leq (\lg n)^{3/2}$, we can represent p_i $(\leq (\lg n)^4)$ using $O(\lg \lg n)$ bits.

We store these primes, indexed by their group number in a separate table. Each prime is stored in a field of $b = \Theta(\lg \lg n)$ bits. If S^i is empty (i.e., there is no sparse set in the i^{th} group), then the table contains a string of b 0s in the entry corresponding to that group. The total space required by this table is $O(n \lg \lg n / \lg n)$ which is $o(n)$ bits.

For a sparse set S^i_j in the i^{th} group, let $n^i_j = |S^i_j|$. If $n^i_j \leq C$, for some constant C to be determined later, we store the elements of S^i_j in the sorted order, which takes $n^i_j \lceil \lg m \rceil$ bits of space. Otherwise, for a sparse set S^i_j, we store an implicit representation of the set $\{g_i(x) | x \in S^i_j\}$. Since $||g_i|| = p_i$, this can be stored using $\lceil \lg \binom{p_i}{n^i_j} \rceil$ bits. Since $p_i \leq (\lg n)^4$ and $n^i_j < \sqrt{\lg n}$, the space used is $o(\lg n)$ bits. We also store a precomputed table (one for each possible size of the sparse sets) which, given the representation of a sparse set S and a value $x < p_i$, returns the *rank* of x in S, for all possible S and x. The space required by this table is $o(n)$ bits, as the representation of any S takes $O(\sqrt{\lg n} \lg \lg n)$ bits and x takes $O(\lg \lg n)$ bits.

Again, to distinguish the element x_r from all the elements of the universe that map to the value $g(x_r)$ (under g), it suffices to store the quotient value

$$q'(x_r) = q(x_r) \lceil n^2/p_i \rceil + f(x_r) \text{ div } p_i.$$

We store these quotient values in the sorted order of the elements of S^i_j. Thus the space used by a sparse set S^i_j is at most $\lg \binom{p_i}{n^i_j} + n^i_j(\lg m - \lg p_i + 4)$ bits (apart from the $o(n)$ bit precomputed table). The term $4n^i_j$ in the space is due to the ceilings in the quotient function. Hence, the space is at most $n_j^i \lg m$, if n_j^i is more than some constant. We choose C to be this constant.

Thus, the space used for a set S_i is at most $|S_i| \lceil \lg m \rceil$ bits in all the cases. If a set S_i takes less than this number of bits, we pad it to this length so that each set S_i takes exactly $|S_i| \lceil \lg m \rceil$ bits. Thus, the representation of a set S_i starts at the position $\sum_{j=1}^{i-1} |S_j| \lg m + 1$ in the representation.

Given $x \in [m]$, to find $rank_i(x)$ we do the following:

First, find the position where the representation of S_i is stored and also the size $n_i = |S_i|$ using the oracle. If $n_i \leq C$, then search the elements of S_i using binary search, as the elements of S_i are stored in sorted order. Otherwise, first read the values of p and k corresponding to the hash function f.

If $n_i < \sqrt{\lg n}$, find the prime p_l stored in the $l = (i/\lg n)^{th}$ location in the table of primes stored separately. Now find the value $g_l(x) = f(x) \bmod p_l$ and then find the rank of $g_l(x)$ using the implicit representation of the set $g_l(S_i)$ (stored at the position where the representation of S_i starts) using the precomputed table. The element x is not in S_i if the rank is -1. Otherwise, if the rank is r, then we know that $g_l(x) = g_l(x_r)$ where x_r is the r^{th} smallest element in the set S_i. Now, return r if $q'(x) = q'(x_r)$ and -1 otherwise. Note that the value $q'(x_r)$ is stored at the r^{th} position in the sequence of quotient values corresponding to the set S_i.

Otherwise $n_i \geq \sqrt{\lg n}$. Now, find the predecessor $y \in B$ of x and its rank l in B using a binary search. If $x \in B$, return $r * l$ as the rank of x. Otherwise, search for $f(x)$ in the dictionary for S'. If it is found, we read the satellite data r' associated with it, for some r' in the range $[1, \ldots, |S'|]$. Let $b = r * l + r'$. If $q(x) = q(x_b)$, return b otherwise return -1. Again, the value $q(x_b)$ is stored in the b^{th} position in the sequence of quotient values corresponding to the set S_i.

The first thing to note is that there are only $O(\lg^c n)$ tables (for some fixed positive constant c) for operations on small sets. Thus the space required by all tables put together is $o(n)$. The second thing is that in order to easily find the start of the representation of a set S_i, the representations of all sets S_i would need to be padded out to precisely $|S_i| \lceil \lg m \rceil$ bits, if necessary.

When $n^2 \geq k$, we follow exactly the same approach, except that we do not use the range reduction function, f for the set S. ∎

In the next section, we show an application of this result by giving an efficient representation of a cardinal tree.

3.4 Representing Cardinal Trees

By a *cardinal tree* (or trie) of degree k (or a k-ary cardinal tree), we mean a rooted tree in which each node has k positions for an edge to a child. A binary tree is a cardinal tree of degree 2. Since there are $\mathcal{C}_n^k \equiv \binom{kn+1}{n}/(kn+1)$ such trees, $(\lg(k-1) + k \lg \frac{k}{k-1})n$ bits is a good estimate of the lower bound on a representation, assuming n is large with respect to k. This bound is roughly $n(\lg k + \lg e)$ bits as k grows. Benoit et al. [BDMR99] have given a representation of a cardinal tree that takes $n \lceil \lg k \rceil + 2n + o(n)$ bits and supports finding the parent, i^{th} child, degree and the subtree size of a given node in constant time. Finding a child labeled i, using their representation, requires $O(\lg \lg k)$ time in the worst case. Using our representation of multiple dictionaries, we give a structure that supports all the five operations in constant time.

The cardinal tree encoding of Benoit et al. [BDMR99] has two parts. In the first part, they store the underlying ordinal tree of the given cardinal tree. (An *ordinal tree* is a rooted ordered tree in which the children of each node are ordered from left to right, with no explicit labels.) They use the succinct encoding of ordinal trees [MR97, BDMR99] (see also Section 2.2.3) which takes $2n + o(n)$ bits to store an ordinal tree on n nodes. This ordinal tree representation supports finding the parent, i^{th} child, degree and the subtree size of a given node, all in constant time. In this representation, a node is identified by its preorder numbering in the tree (i.e., the position of the node in the preorder traversal of the tree).

In the second part, they store the information about the labels of the children present at that node. In particular, they store the set of labels of the children present using a static dictionary, for each node. They use a static dictionary that takes $n \lceil \lg m \rceil$ bits of space to represent a set of size n from the universe $[m]$ and supports membership and rank operations in $O(\lg \lg m)$ time. This takes $d \lceil \lg k \rceil$ bits for a node with d children in a k-ary cardinal tree. Hence, to represent this child information for each node in a k-ary tree on n nodes, they use $(n-1) \lceil \lg k \rceil$ bits of space, since the sum of the degrees (i.e., the number of children) of all the nodes is $n-1$. Thus, their overall representation takes $n(\lceil \lg k \rceil + 2) + o(n)$ bits of space.

Now, to find the child labeled i of a node, we first find the rank r of the element i in the static dictionary corresponding to that node. If the rank is -1, then we answer that there is no such child. Otherwise, we return the r^{th} child of the node (using the ordinal tree representation).

We now describe our modification to the cardinal tree representation of Benoit et al. [BDMR99] to support all the operations in constant time. Let S_i be the set containing all the child labels of node i in the given tree, for $1 \leq i \leq n$. Note that $\sum_{i=1}^{n} |S_i| = n - 1$. Store these sets using the representation of Theorem 3.3.1 which takes $n \lceil \lg k \rceil + o(n) + O(\lg \lg k)$ bits of space. Also note that given i, the values, $\sum_{j=1}^{i-1} |S_i|$ and $|S_i|$ can be computed in constant time, using the ordinal tree representation stored. (This is because, the ordinal tree representation, a node is identified by its preorder numbering in the tree. One can easily verify that the preorder number of the leftmost child of a node numbered i is nothing but $1 + \sum_{j=1}^{i-1} d_j$, where d_j denotes the number of children of the j^{th} node. Also, note that $|S_j| = d_j$.) Thus, using this representation, given a node x and a child label l, one can find the rank of the child labeled l at x in constant time. Hence, this cardinal tree representation also supports finding a child labeled j of a given node in constant time. Thus we have,

Theorem 3.4.1 *There exists an $n \lceil \lg k \rceil + 2n + o(n) + O(\lg \lg k)$ bit representation of an k-ary cardinal tree on n nodes that supports, given any node, finding its parent, i^{th} child, child labeled j, degree and the size of the subtree rooted at the node in constant time.*

3.4.1 Application to Suffix Tree Representation

In Theorem 2.2.3 of Section 2.2, we gave a representation of a suffix tree for a given text, which is stored as a binary tree by converting the suffixes of the text into binary. Instead, if we directly store the suffix tree using the cardinal tree representation of Section 3.4, we get the following:

Theorem 3.4.2 *A suffix tree for a given text of length n over an alphabet Σ can be represented using $n(\lg n + \lg |\Sigma|) + O(n + \lg \lg |\Sigma|)$ bits of space using which, given a pattern of length m, one can search for an occurrence of the pattern in $O(m)$ time.*

Chapter 4

Static Dictionary in the Bitprobe Model

4.1 Introduction

In this chapter, we look at the static membership problem: given a subset S of up to n keys drawn from the universe $[m]$, store it so that queries of the form "Is x in S?" can be answered quickly. We study this problem in the bitprobe model. In this model space complexity is measured in terms of the number of bits used to store the data structure, and time complexity in terms of the number of bits of the data structure probed (i.e., looked at) in answering a query. All other computations are free.

A simple characteristic bit vector gives a solution to the problem using m bits of space in which membership queries can be answered using one bit probe. On the other hand, the structures given by Fredman et al. [FKS84], Brodnik and Munro [BM99], and Pagh [Pag01a] can be used to get a scheme that uses $O(n \lg m)$ bits of space in which membership queries can be answered using $O(\lg m)$ bit probes.

Buhrman et al. [BMRV00] have shown that both the above schemes are optimal. More specifically, they have shown that any scheme that uses s bits of space and answers membership queries using t bit probes satisfies that condition: $\binom{m}{n} \leq \max_{i \leq nt} \binom{2s}{i}$. (This inequality has been tightened further by Radhakrishnan et al. [RSV00].) In particular, this implies that any scheme that answers membership queries using one bit probe requires at least m bits of space and that any scheme using $O(n \lg m)$ bits of space requires $\Omega(\lg m)$ probes to answer membership queries, when $n \leq m^{1-\epsilon}$, for any fixed positive constant $\epsilon \leq 1$. They have also considered the intermediate ranges and have given some upper and lower bounds for randomized as well as deterministic versions of the problem. Their main result is that the optimal $O(n \lg m)$ bits (for $n \leq m^{1-\Omega(1)}$) of space is sufficient to answer queries using one bit, if the query algorithm is allowed to make errors on both sides (i.e., when the query element is either present or absent in the given set) with a small probability.

For the deterministic case, they have shown that the static dictionary structure given by Fredman, Komlos and Szemeredi [FKS84] can be modified to give a scheme that uses $O(nkm^{1/k})$ bits of space and answers membership queries by probing $O(\lg n + \lg\lg m) + k$ bits from the structure, for any parameter $1 \leq k \leq \lg m$. Using probabilistic arguments, they have also shown the existence of a scheme that uses $O(m^{3/t} n \lg m)$ bits and answers queries using t probes, for any t $(4 \leq t \leq O(\lg m))$. For $n = 2$, they have shown the existence of a scheme that takes $O(m^{3/4})$ bits of space and answers membership queries using two adaptive bit probes.

Our main contribution is some improved deterministic upper bounds for the problem using explicit constructions, particularly for small values of t.

4.1.1 Definitions and Notations

We reproduce the definitions of a storage scheme and a query scheme, introduced in [BMRV00]. An (n, m, s)-storage scheme, is a method for representing any subset of size at most n over a universe of size m as an s-bit string. Formally, an (n, m, s)-storage scheme is a map ϕ from the subsets of size at most n of the universe $[m]$ to $\{0, 1\}^s$. It is called an explicit storage scheme, if there is a Turing machine which given the subset D, outputs $\phi(D)$ in $s^{O(1)}$ time. A deterministic (m, s, t)-query scheme is a family of m boolean decision trees $\{T_1, T_2, \ldots, T_m\}$, of depth at most t. Each internal node in a decision tree is marked with an index between 1 and s, indicating an address of a bit in an s-bit data structure. All the edges are labeled by "0" or "1" indicating the bit stored in the parent node. The leaf nodes are marked "Yes" or "No". Each tree T_i induces a map from $\{0, 1\}^s \rightarrow \{\text{Yes, No}\}$. An (m, s, t)-query scheme is explicit, if there is a Turing Machine running in time $O(\lg m)^{O(1)}$, which on input x and with oracle access to $\phi(D)$ executes the correct sequence of probes according to the query scheme and accepts or rejects accordingly.

An (n, m, s)-storage scheme and a deterministic (m, s, t)-query scheme together form a deterministic (n, m, s, t)-scheme which solves the (n, m)-membership problem if $(\forall S, x \ s.t.\ |S| \leq n$, and $x \in U : T_x(\phi(S)) =$ 'Yes' if and only if $x \in S)$. An (n, m, s, t)-scheme is explicit, if the associated storage and query schemes are explicit. A non-adaptive deterministic scheme is a scheme, where in each decision tree of the query scheme, all the nodes on a particular level are marked with the same index. In this chapter, we fix our universe to be $[m]$ and hence refer to an (n, m, s, t)-scheme simply as an (n, s, t)-scheme.

We follow the convention that whenever the universe $[m]$ is divided into blocks of size b (or equivalently m/b blocks), the set of elements from the subset $B_i = \{(i-1)b, \ldots, ib-1\}$ of the universe belong to the i^{th} block, for $1 \leq i \leq \lfloor m/b \rfloor$ and the remaining (at most $b-1$) elements, $\{\lfloor m/b \rfloor b, \ldots, m-1\}$, belong to the last block. Given a subset S of the universe and also given that the universe is divided into b blocks, we call the i^{th} block empty if $S \cap B_i = \emptyset$ (i.e., if no element of the set S belongs to that block), and non-empty otherwise. We call the number of non-empty blocks with indices less than or equal to i as the rank of the i^{th} block among the non-empty blocks. By characteristic vector of the i^{th} block, we mean the bit vector of length m/b, where the j^{th} bit is a 1 if and only if $(i-1)b+j-1 \in S$, for $1 \leq j \leq b$.

We assume, without loss of generality, that the universe size m is a perfect square. (All the schemes which use this assumption can be easily modified to schemes which take the same amount of asymptotic space and answer the queries using the same number of probes, even if m is not a perfect square.) Also, given any element $x \in [m]$, we use the abbreviations $x^q = div(x, \sqrt{m})$ and $x^r = mod(x, \sqrt{m})$ (q and r stand for quotient and reminder respectively). We also refer to them as the first and second halves of x respectively. Also, to simplify the notation, we ignore integer rounding ups and downs at some places where they do not affect the asymptotic analysis.

The next section gives improved upper bounds for deterministic schemes. In particular, Section 4.2.1 gives some schemes for small values of n. Section 4.2.2 developes a non-adaptive scheme that answers queries using $O(\sqrt{n} \lg n)$ probes, taking $O(\sqrt{nm} \lg n)$ bits of space. In Section 4.2.3, we develop schemes with $o(m)$ bits that answer queries using either $\lg \lg n + 2$ adaptive probes or $O(\lg n)$ non-adaptive probes. In section 4.3, we give some space lower bounds for a restricted class of two probe schemes, matching some of our upper bounds.

4.2 Upper Bounds for Deterministic Schemes

4.2.1 Deterministic Schemes for small n

For $n = 1$, one can easily get an optimal $(1, tm^{1/t}, t)$-scheme [BMRV00] for any given t by splitting the bit representation of the given element into t (roughly) equal parts and storing the characteristic vectors of each part by considering it as subset of the set $[m^{1/t}]$. (The space used by this scheme is within a constant factor of the optimum; for example, for $n = 2$, this scheme takes $2\sqrt{m}$ bits whereas the lower bound is $\sqrt{2m} + O(1)$ bits. One can in fact achieve the optimum space by assigning a unique code $c \in \{0,1\}^s$ of weight t to each element of the universe, where s is the lower bound on the number of bits required.)

For $n = 2$, Buhrman et al. [BMRV00] have shown that any non-adaptive two probe scheme requires m bits of space. We first start with a simple $O(\sqrt{m})$-bit non-adaptive scheme for two element sets that uses four probes to answer queries. Given a subset $\{x, y\}$ of $[m]$, assume without loss of generality that $x < y$. The storage scheme consists of three tables T_1, T_2 and T_3 each of length $\sqrt{m}$. We set $T_1[x^q] = 01$, $T_1[y^q] = 10$ and all other locations in T_1 to 00 (if $x_q = y_q$, then set $T_1[x_q] = T_2[y_q] = 01$, say) and set $T_2[x^r]$ and $T_3[y^r]$ to one and all other bits in T_2 and T_3 to zero. Given a query element $z \in [m]$, the query scheme probes the locations $T_1[z^q]$, $T_2[z^r]$ and $T_3[z^r]$. If $(T_1[z^q] = 01$ and $T_2[z^r] = 1)$ or $(T_1[z^q] = 10$ and $T_3[z^r] = 1)$, then we answer 'Yes'; in all other cases we answer 'No'. Thus, this gives us a $(2, s, 4)$-scheme with $s = 4\sqrt{m}$.

Note that this immediately gives an adaptive $(2, 4\sqrt{m}, 3)$-scheme (using the same storage scheme). Also, using the same idea, for three-element sets one can get a non-adaptive $(3, 5\sqrt{m}, 5)$-scheme and an adaptive $(3, 5\sqrt{m}, 3)$-scheme.

The non-adaptive $(2, 4\sqrt{m}, 4)$-scheme is improved to a three probe scheme, using the idea of Steiner triples, by Buhrman et al.[BMRV00]. The storage scheme consists of the characteristic vectors of the subsets $\{x^q, y^q\}$, $\{x^r, y^r\}$ and $\{mod(x^q + x^r, \sqrt{m}), mod(y^q + y^r, \sqrt{m})\}$ of $[\sqrt{m}]$. Given a query element $z \in [m]$, the query scheme probes the locations z^q, z^r and $mod(z^q + z^r, \sqrt{m})$ respectively in the three vectors and answers 'Yes' if and only if all the probed locations are 1. Thus, this gives a $(2, s, 3)$-scheme with $s = 3\sqrt{m}$. Note that this scheme is inherently non-adaptive.

We now explore generalizations of these schemes in two different directions. In the first direction in Section 4.2.2, we keep the relation between time and space to be the same (i.e., $s = t\sqrt{m}$) and then try to reduce the number of probes (t), considering only non-adaptive schemes. We achieve a scheme with $t = O(\sqrt{n \lg n})$ and $s = t\sqrt{m}$. In Section 4.2.3, we develop schemes with $s = o(m)$ that answer queries using as few probes as possible. We achieve a non-adaptive (n, s, t)-scheme with $t = O(\lg n)$ and $s = o(m)$, for a large range of n.

4.2.2 Non-adaptive Schemes with $s = t\sqrt{m}$

The non-adaptive $(2, 4\sqrt{m}, 4)$-scheme can be easily generalized to get a non-adaptive (n, s, t)-scheme with $t = \lceil \lg(n+1) \rceil + n$ and $s = (n + \lceil \lg(n+1) \rceil)\sqrt{m}$. This can be improved by generalizing the non-adaptive $(2, 3\sqrt{m}, 3)$-scheme to the following:

Theorem 4.2.1 *There is an explicit non-adaptive (n, s, t)-scheme with $t = n + 1$ and $s = t\sqrt{m}$.*

Proof. The scheme has $t = n + 1$ tables each of size $\sqrt{m}$ bits. To each of the m elements we assign t bits, one in each table. These bits are be determined by an equation (4.2.1) below. To represent a given set of elements, for each element in the set we set its corresponding t bits to 1. All other bits are set to 0. To determine whether an element is present, we simply check whether all its assigned bits are 1.

Let m' be the smallest prime number greater than or equal to $\sqrt{m}$. Consider an element $x \in [m]$. Treating $\{x^q, x^r\}$ as a basis of a vector space over the finite field $\mathbf{Z}/m'$, consider the following pairwise linearly independent vectors (note that both x^q and x^r are less than m'):

$$x^q,\ x^r,\ x^q + x^r,\ 2x^q + x^r,\ 3x^q + x^r,\ \ldots,\ (m'-1)x^q + x^r, \tag{4.2.1}$$

where arithmetic is modulo m'. Because these vectors are pairwise linearly independent and the dimension of the vector space is two, given any two of these vectors we can obtain the unit vectors x^q and x^r from linear combinations. In other words, the values of any two of these vectors uniquely determine the values of x^q and x^r, and hence also x.

Recall that to each element we must assign a bit in each of the $t = n + 1$ tables. Now we can specify the bits: for the i^{th} table, we choose the bit with address given by the i^{th} vector in (4.2.1). Because there are only $m' + 1 \geq \sqrt{m} + 1$ such vectors, we must have $t = n + 1 \leq \sqrt{m} + 1$, i.e., $n \leq \sqrt{m}$. But note that the statement is trivial if $n \geq \sqrt{m}$: a bit vector would suffice to prove the bound.

So, we have an assignment of bits to elements. Now observe that (by the "two values determine all the others" property proved above) any two *distinct* elements share at most one vector in (4.2.1), and hence share at most one bit. Because there are only n elements and $t = n+1$ bits assigned to each element, not all the locations corresponding to any element would have been set to 1, if that element is not present. Thus, no element will be mistakenly thought to be in the set by examining the t bits. ∎

By choosing a larger vector field with $c + 1$ basis elements, one can generalize the above scheme to get the following:

Corollary 4.2.2 *There is an explicit non-adaptive (n, s, t)-scheme with $t = cn + 1$ and $s = tm^{1/(c+1)}$, for any integer $1 \leq c \leq \lg m - 1$.*

Now we develop a different idea to reduce the number of probes. The following lemma gives a scheme when the given set of elements satisfies some restrictions. This is used in the next theorem to get a structure for unrestricted sets.

Lemma 4.2.3 *There is an explicit non-adaptive $(n, s, 2\lceil\lg(n + 1)\rceil)$-scheme with $s = t\sqrt{m}$, if all the elements of the set to be represented have either distinct first halves or distinct second halves in their binary representations.*

Proof. Divide the universe into blocks of size $\sqrt{m}$. Suppose all the elements have distinct first halves in their binary representations. The case when all the elements have distinct second halves is symmetric. The storage scheme consists of two tables T_1 and T_2, corresponding to the two halves of the binary representations of the elements. We store a number in the range $[0, \ldots, n]$ (using $\lceil\lg(n + 1)\rceil$ bits) corresponding to each block in each of T_1 and T_2.

Let $x_1, x_2, \ldots, x_n$ be the sequence of elements of the given set in the increasing order. For each i, let ρ_i be the rank of x_i^r (i.e., the number of distinct values less than x_i^r) in the set $\{x_j^r | 1 \leq j \leq n\}$.

Now, set $T_1[x_i^q]$ and $T_2[x_i^r]$ to $\rho_i + 1$, for $1 \leq i \leq n$ and all other locations in T_1 and T_2 to 0. One can verify that this gives a valid storage scheme (since all the elements have distinct first halves, i.e., $x_i^q \neq x_j^q$ for $i \neq j$).

Given a query element x the scheme looks the values stored in $T_1[x^q]$ and $T_2[x^r]$. If these two values are equal and non-zero, it answers 'Yes'; otherwise it answers 'No'. One can easily verify that this scheme correctly answers the membership queries. ∎

This can be combined with the simple bit vector scheme to give the following result:

Theorem 4.2.4 *There is an explicit non-adaptive (n, s, t)-scheme with $t = 1 + 2\lceil \lg(n+1) \rceil + \lfloor n/2 \rfloor$ and $s = t\sqrt{m}$.*

Proof. The storage scheme consists of a bit vector B of length $\sqrt{m}$, two tables T_1 and T_2 of size $\sqrt{m}$, each storing a value in the range $[0, \ldots, n]$, and $\lfloor n/2 \rfloor$ bits vectors $V_1, \ldots, V_{\lfloor n/2 \rfloor}$, each of length $\sqrt{m}$, for a total space of $\sqrt{m}(1+2\lceil \lg(n+1) \rceil + \lfloor n/2 \rfloor)$ bits.

Divide the universe into blocks of size $\sqrt{m}$. Note that an element x belongs to the block numbered x^q. For $1 \leq i \leq \sqrt{m}$, the i^{th} bit in the bit vector B is set to 1 if and only if the block has at most one element from the given set. If a block contains exactly one element x from the given set, then store in $T_1[x^q]$ the rank of x^r in the set $\{y^r | y \in S'\}$, where S' is the set of all the elements that fall into such blocks containing exactly one element from the given set. If it has more than one element, then store its rank, among such blocks, in T_1. For each element, x that falls into a block with exactly one element, store the value $T_1[x^q]$ at location x^r in T_2. For each block with at least two elements, store the characteristic vector of that block in the vector V_i, where i is the value stored in T_1 in the corresponding block. Set all the unspecified values to zero.

Given an element x, to find whether x belongs to the given set, we read the bit $B[x^q]$, the values $T_1[x^q]$ and $T_2[x^r]$, and the bits $V_i[x_r]$, for $1 \leq i \leq \lfloor n/2 \rfloor$. If $B[x^q] = 1$, then we look at the values $T_1[x^q]$ and $T_2[x^r]$. If these values are equal and non-zero, then we answer that the element x is present. Otherwise, if $B[x^q] = 0$, then look at the bit read from the bit vector V_j, where j is the value read from table T_1 (in other words, the bit $V_{T_1[x^q]}[x^r]$). If it is 1, then we answer that the element x is present. In all the other cases, we answer that the element x is not present in the given set. ∎

Remark 4.2.5 One can use the same storage scheme to get an adaptive scheme by first reading the $1 + \lceil \lg(n+1) \rceil$ bits from B and T_1, and then reading either the $\lceil \lg(n+1) \rceil$ bits from table T_2 or one bit from the corresponding bit vector V_i

(depending on the value of the bit read from B). This gives us an explicit adaptive (n, s, t)-scheme with $t = 1 + 2\lceil \lg(n+1) \rceil)$ and $s = O(n\sqrt{m})$.

We now generalize the scheme of Theorem 4.2.4 to reduce the number of bit probes required to answer a query (and there by also reducing the storage space required). The main idea here is to distinguish blocks containing at most $k-1$ elements from the other blocks. We again store an 'indicator bit' for each block to store this information. For all those elements that fall into blocks containing at most $k-1$ elements, we store the ranks of their second halves in a table T_2.

The table T_1 corresponding to the first halves contains $k-1$ entries for each block. For each block containing at least k elements we store the rank of it among such blocks in its first entry of T_1 and also store its characteristic vector in a bit vector indexed by its rank. For blocks containing at most $k-1$ elements, we store the (at most $k-1$) values stored in T_2 corresponding to each of the elements falling into that block, in the entries of T_1 (in any order). All other unspecified entries (in all the tables) are set to 0.

Given an element, we read all the entries corresponding to that element (namely, the indicator bit, the entries in tables T_1 and T_2 and corresponding bits from each of the characteristic vectors). From the indicator bit, we find whether it belong to a block containing at most $k-1$ elements. If so, then we look at all the $k-1$ entries from T_1 and the entry read from T_2, and answer 'Yes' if and only if any of the non-zero entries read from T_1 is equal to the entry read from T_2. Otherwise, if the query element belongs to a block containing at least k elements, we look at the first entry, say i, read from T_1 and then look at the bit read from the i^{th} characteristic vector. We answer 'Yes' if and only if this bit is 1.

Thus we have

Theorem 4.2.6 *There is an explicit non-adaptive (n, s, t)-scheme with $t = 1 + k\lceil \lg(n+1) \rceil + \lfloor n/k \rfloor$ and $s = t\sqrt{m}$, for any parameter $k \geq 1$.*

Substituting $k = \lfloor \sqrt{n/\lg n} \rfloor$ in the above theorem, we get

Corollary 4.2.7 *There is an explicit non-adaptive $(n, s, O(\sqrt{n \lg n}))$-scheme with $s = t\sqrt{m}$.*

Note that this scheme performs better than the scheme of Theorem 4.2.1 both in terms of time and space, for large n. This gives the best known constructive scheme (in terms of time) for the given amount of space ($O(\sqrt{m})$ for fixed n). We give another scheme that takes less number of probes albeit using more space, in the next subsection by first giving an adaptive scheme and then making it non-adaptive.

4.2.3 Adaptive Schemes

The goal in this section is to get as small t as possible keeping the space to be $o(m)$.

For $n = 2$, if only two probes are allowed, Buhrman et al. [BMRV00] have shown that, any *non-adaptive* scheme must use m bits of space. They have also shown the existence of an *adaptive* scheme using 2 probes and $O(m^{3/4})$ bits of space. We improve it to the following:

Theorem 4.2.8 *There is an explicit adaptive* $(2, s, 2)$*-scheme with* $s = 3m^{2/3}$.

Proof. Divide the universe into blocks of size $m^{1/3}$ each. There are $m^{2/3}$ blocks. Group $m^{1/3}$ consecutive blocks into a superblock. There are $m^{1/3}$ superblocks of size $m^{2/3}$ each.

The storage scheme consists of three tables T, T_0 and T_1, each of size $m^{2/3}$ bits. Each element $x \in [m]$ is associated with three locations, $t(x)$, $t_0(x)$ and $t_1(x)$, one in each of the three tables, as defined below. Let $b = m^{2/3}$ and $b_1 = m^{1/3}$. Then, $t(x) = div(x, b_1)$, $t_0(x) = mod(x, b)$ and $t_1(x) = div(x, b)\ b_1 + mod(x, b_1)$. (I.e., in the bit representation of x, $t(x)$ is the value of the first two-thirds of the bits, $t_0(x)$ is the value of the last two-thirds of the bits and $t_1(x)$ is the value of the concatenation of the first one-thirds and the last one-thirds of the bits.)

Given an element $x \in [m]$, the query scheme first looks at $T(t(x))$. If $T(t(x)) = j$, it looks at $T_j(t_j(x))$ and answers 'Yes' if and only if it is 1, for $j \in \{0, 1\}$.

To represent a set $\{x, y\}$, if both x and y belong to the same superblock (i.e., if $div(x, b) = div(y, b)$), then we set the bits $T(t(x))$ and $T(t(y))$ to 0, all other bits in T to 1; $T_0(t_0(x))$ and $T_0(t_0(y))$ are set to 1 and all other bits in T_0 and T_1 are set to 0. In other words, we represent the characteristic vector of the superblock containing both the elements, in T_0, in this case.

Otherwise, if both the elements belong to different superblocks, we set $T(t(x))$, $T(t(y))$, $T_1(t_1(x))$ and $T_1(t_1(y))$ to 1 and all other bits in T, T_0 and T_1 to 0. In this case, each superblock has at most one non-empty block (containing one element). So in T_1, for each superblock, we store the characteristic vector of the only non-empty block in it, if it exists, and a sequence of zeroes otherwise. One can easily verify that the storage scheme is valid and that the query scheme answers membership queries correctly using two adaptive bit probes. ∎

The number of probes used by this scheme can be slightly improved by generalizing the adaptive $(2, 3m^{2/3}, 2)$-scheme to work for larger n, to get the following.

Theorem 4.2.9 *There is an explicit adaptive* $(n, s, 1+\lceil \lg(\lfloor n/2 \rfloor + 2) \rceil)$*-scheme with* $s = O(m^{2/3}(n/2 + \lg(n/2+2)+1))$.

Proof. The idea is to distinguish superblocks containing at least 2 elements from those containing at most one element.

In the first level, if a superblock contains at least 2 elements, we store its rank among all superblocks containing at least 2 elements, with all its blocks. Since there can be at most $\lfloor n/2 \rfloor$ superblocks containing at least 2 elements, the rank can be any number in the range $\{1, \ldots, \lfloor n/2 \rfloor\}$. For blocks which fall into superblocks containing at most one element, we store the number $\lfloor n/2 \rfloor + 1$, if the block is non-empty and a sequence of $\lceil \lg(\lfloor n/2 \rfloor + 2) \rceil$ zeroes, otherwise.

The second level consists of $\lfloor n/2 \rfloor + 1$ bit vectors of size $m^{2/3}$ each. We store the characteristic vector of the j^{th} superblock containing at least two elements in the j^{th} bit vector, for $1 \le j \le l$, where l is the number of superblocks containing at least 2 elements. We set all other bit vectors (indexed $l+1$ to $\lfloor n/2 \rfloor$) to zeroes. In the $(\lfloor n/2 \rfloor + 1)^{st}$ bit vector, for each superblock we store the characteristic vector of the only non-empty block in it, if it has exactly one non-empty block or a sequence of zeroes otherwise.

On query x, we look at the first level entry of the block corresponding to x. If it is 0, we answer 'No'. Otherwise, if it is a number k in the range $[1, \ldots, \lfloor n/2 \rfloor]$, we look at the corresponding location of x in the k^{th} bit vector in the second level (which stores the bit vector corresponding to the superblock containing x). Otherwise (if the number is $\lfloor n/2 \rfloor + 1$), we look at the corresponding location of x in the last bit vector and answer accordingly. ∎

Remark 4.2.10 The adaptive $(2, 4\sqrt{m}, 3)$-scheme given in Section 4.2.1 can be easily generalized, for larger n, to get an adaptive (n, s, t)-scheme with $t = \lceil \lg(n+1) \rceil + 1$ and $s = \sqrt{m}(n + \lceil \lg(n+1) \rceil)$. Note that, compared with this scheme, the savings in the number of probes achieved by the scheme of Theorem 4.2.9 is at most one (and the space usage is higher). But this scheme guides us in achieving further reduction in the number of probes.

This can be further generalized as follows. In the first level, we distinguish the superblocks having at least k elements (for some integer k) from those with at most $k-1$ elements in them. For superblocks having at least k elements, we store the rank of that superblock among all such superblocks, in all the blocks of that superblock. For the other blocks, we store the rank of the block among all non-empty blocks in that superblock, if the block is non-empty and a sequence of zeroes otherwise. The second level has $\lfloor n/k \rfloor + k - 1$ bit vectors of length $m^{2/3}$ each, where in the first $\lfloor n/k \rfloor$ bit vectors we store the characteristic vectors of the at most $\lfloor n/k \rfloor$ superblocks containing at least k elements in them (in the order of increasing rank) and pad the rest of them with zeroes. Each of the $(\lfloor n/k \rfloor + j)^{th}$ bit vectors, for $1 \leq j \leq k-1$, stores the characteristic vector of one block from every superblock. This block is the j^{th} non-empty block in that superblock, if that superblock contains at least j non-empty blocks and at most $k-1$ elements; we store a sequence of zeroes otherwise. The query scheme is straightforward. This results in the following:

Corollary 4.2.11 *There is an explicit adaptive $(n, s, 1 + \lceil \lg(\lfloor n/k \rfloor + k) \rceil)$-scheme with $s = O(m^{2/3}(n/k + \lg(n/k + k) + k))$.*

Choosing $k = \lceil \sqrt{n} \rceil$, we get

Corollary 4.2.12 *There is an explicit adaptive $(n, s, 2 + \lceil \frac{1}{2} \lg n \rceil)$-scheme with $s = O(m^{2/3}\sqrt{n})$.*

This can be slightly improved by choosing the block sizes appropriately as shown below:

Theorem 4.2.13 *There is an explicit adaptive $(n, s, 2 + \lceil \frac{1}{2} \lg n \rceil)$-scheme with $s = O(m^{2/3}(n \lg n)^{1/3})$.*

Proof. Divide the universe into blocks of size b and subdivide each block in turn into sub-blocks of size b_1 (where b and b_1 are to be determined later).

The structure consists of three levels. In the first level we store a bit vector of length m/b. We set the i^{th} bit to 1 if the i^{th} block has at least $\lfloor \sqrt{n} \rfloor$ elements, and to 0 otherwise.

The second level consists of a table of size m/b_1 each entry corresponding to a sub-block of the universe. If the block to which a sub-block belongs has at least $\lfloor\sqrt{n}\rfloor$ elements, we store the rank of that block, among all such blocks, in the table entry corresponding to that sub-block. Since there can be at most $\lceil\sqrt{n}\rceil - 1$ blocks with at least $\lfloor\sqrt{n}\rfloor$ elements, each entry can be stored using $\lceil\frac{\lg n}{2}\rceil$ bits.

Otherwise, if the block to which a sub-block belongs has at most $\lceil\sqrt{n}\rceil - 1$ elements, store its rank among all the non-empty sub-blocks in that block, if the block is non-empty; store a 0 otherwise. Since there can be at most $\lceil\sqrt{n}\rceil - 1$ non-empty sub-blocks, each entry again takes $\lceil\frac{\lg n}{2}\rceil$ bits.

The third level has two parts. In the first part, for each block with at least $\lfloor\sqrt{n}\rfloor$ elements, we store the bit vector of the block, in the order in which they appear. Since there are at most $\lceil\sqrt{n}\rceil - 1$ such blocks, this requires $b\lceil\sqrt{n}\rceil$ bits of space.

In the second part, for each block we allocate $b_1\lceil\sqrt{n}\rceil$ bits of space. For each block with at most $\lceil\sqrt{n}\rceil - 1$ elements, we store the bit vectors of all non-empty sub-blocks in the space allocated for that block. For the remaining blocks, we store all zeroes.

The space occupied for the overall structure is

$$s = \frac{m}{b} + \frac{m}{2b_1}\lg n + n^{1/2}(b + \frac{m}{b}b_1)$$

and the number of probes is $t = 1 + \lceil\frac{1}{2}\lg n\rceil + 1$. Thus, choosing

$$b = \frac{m^{2/3}(\lg n)^{1/3}}{2^{1/3}n^{1/6}} \text{ and } b_1 = \frac{m^{1/3}(\lg n)^{2/3}}{2^{2/3}n^{1/3}}$$

makes the space s to be $\frac{3}{2^{1/3}}m^{2/3}(n\lg n)^{1/3} + o(m^{2/3})$ bits. ∎

We generalize the scheme of Corollary 4.2.12 to the following:

Theorem 4.2.14 *There is an explicit adaptive* $(n, s, \lceil\lg(k+1)\rceil + \lceil\lg\lfloor n^{1/k}\rfloor\rceil + 1)$-*scheme with* $s = m^{k/(k+1)}\left(\lg k + \frac{1}{k}\lg n + kn^{1/k}\right)$, *for* $k \geq 1$.

Proof. We divide the universe into blocks of size b (to be determined later) and construct a complete b-ary tree with these blocks at the leaves. Let k be the height of this tree. Then, we have $m = b^{k+1}$ or $b = m^{1/(k+1)}$. Given a set S of n elements from the universe, we store it using a three level structure. We define the height of a node in the tree to be the length of the path (the number of nodes in the path) from that node to any leaf in the subtree rooted at that node. Note that the height of the root is $k + 1$ and that of any leaf is one.

In the first level we store an index in the range $[0, \ldots, k]$ corresponding to each block. Thus, the first level consists of a table B of size b^k where each entry is a $\lceil \lg k \rceil$ bit number. The index stored for an empty block is 0. For a non-empty block, we store the height $h \leq k$ of its ancestor (excluding the root) x of maximum height such that the total number of elements falling into all the blocks in the subtree rooted at node x is at least $\lfloor n^{(h-1)/k} \rfloor$. This will be a number in the range $[1, \ldots, k]$.

In the second level we store a number in the range $[0, \ldots, \lceil n^{1/k} \rceil - 1]$ corresponding to each block. Thus this level consists of a table T of size b^k, each entry of which is a $\lceil \lg n^{1/k} \rceil$ bit number. The number stored for an empty block is 0. For a non-empty block, we store the following:

Observe that given any node x at height h which has less than $\lfloor n^{(h-1)/k} \rfloor$ elements from the set, the number of its children which have at least $\lfloor n^{(h-2)/k} \rfloor$ elements from the set is less than $\lceil n^{1/k} \rceil$. Suppose the index stored for a block is l ($l \neq 0$). It means that the ancestor x of that block at height l has at least $\lfloor n^{(l-1)/k} \rfloor$ elements and the ancestor y at height $l+1$ has less than $\lfloor n^{l/k} \rfloor$ elements. Hence, y can have less than $\lceil n^{1/k} \rceil$ children which have at least $\lfloor n^{l/k} \rfloor$ elements. Call these the 'large' children of y. All the leaves in the subtree rooted at each large child of y are blocks with index l. With these blocks we store the rank of that child among all large children of y (from left to right) in the second level.

In the third level, we have k tables, each of size $\lceil n^{1/k} \rceil m/b$ bits. The i^{th} table stores the representations of all blocks whose first level entry (in table B) is i. We think of the i^{th} table as a set of $\lceil n^{1/k} \rceil$ bit vectors, each of length m/b. Each of these bit vectors in the i^{th} level stores the characteristic vector of a particular child for each node at height i of the tree, in the left to right order. More specifically, for each block (of size b) with first level entry i and second level entry j, we store the characteristic vector of that block in the j^{th} bit vector of the i^{th} table at the location corresponding to its block of size b^{k-i}. We store zeroes at all other locations not specified above.

Every element $x \in [m]$ is associated with $k+2$ locations $b(x)$, $t(x)$ and $t_i(x)$ for $0 \leq i \leq k-1$, as defined below: $b(x) = t(x) = div(x, b)$, $t_i(x) = mod(div(x, b^{k-i})b^i + mod(x, b^i), b^k)$. (These values of $t(x)$ and $t_i(x)$, for each i, can be interpreted using the bit representation of x, as in the proof of Theorem 4.2.8.)

Given an element x, to determine the membership of x, we first read $i = B(b(x))$ and $j = T(t(x))$ from the first two levels of the structure. If $j = 0$, we answer 'No'.

Otherwise, we read the j^{th} bit in the table entry at location $t_i(x)$ in table T_i and answer 'Yes' if and only if it is 1.

The space required for the structure is $b^k(\lceil \lg(k+1) \rceil + \lceil \frac{1}{k} \lg n \rceil) + \frac{m}{b} k \lfloor n^{1/k} \rfloor$ bits. Substituting $b = m^{1/(k+1)}$ makes the space complexity to be $m^{k/(k+1)}(\lceil \lg(k+1) \rceil + \lceil \frac{1}{k} \lg n \rceil + kn^{1/k})$. Also, this takes $\lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil + 1$ probes to answer a query. ∎

One can slightly improve the space complexity of the above structure by choosing non-uniform block sizes and making the block sizes (i.e., branching factors at each level in the above tree structure) to be a function of n (using similar ideas used in the proof of Theorem 4.2.13). More precisely, by choosing the branching factor of all the nodes at level i in the above tree structure to be b_i, where $b_i = m^{1-\frac{i}{k+1}} \left(\frac{\lceil \lg(k+1) \rceil + \lceil \frac{1}{k} \lg n \rceil}{\lceil n^{1/k} \rceil} \right)^{i/(k+1)}$, we get

Corollary 4.2.15 *There is an explicit adaptive* $(n, s, \lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil + 1)$*-scheme with* $s = (k+1) m^{k/(k+1)} \left(n(\lceil \lg(k+1) \rceil + \lceil \lg n^{1/k} \rceil) \right)^{1/(k+1)}$*, for* $k \geq 1$.

By setting $k = \lg n$, we get

Corollary 4.2.16 *There is an explicit adaptive* $(n, s, \lceil \lg \lg n \rceil + 2)$*-scheme with* $s = O(m/\lg m)$*, when* n *is* $O(m^{1/\lg \lg m})$.

In the adaptive scheme of Theorem 4.2.14, we first read $\lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil$ bits from the first two levels of the structure, and depending on these bits we look at one more bit in the third level to determine whether the query element is present. An obvious way to make this scheme non-adaptive is to read the $\lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil$ bits from the first two levels and all possible $k \lceil n^{1/k} \rceil$ bits in the next level and determine the membership accordingly. Thus we get an explicit non-adaptive (n, m, s, t)-scheme with $t = \lceil \lg(k+1) \rceil + \lceil \lg \lfloor n^{1/k} \rfloor \rceil + k \lceil n^{1/k} \rceil$ and $s = tm^{k/(k+1)}$. By setting $k = \lceil \lg n \rceil$ in this, we get

Corollary 4.2.17 *There is an explicit non-adaptive* $(n, s, O(\lg n))$*-scheme with* $s = O(m/\lg m)$*, when* n *is* $O(m^{1/\lg \lg m})$.

The schemes of Corollary 4.2.16 and Corollary 4.2.17 give the best known (in terms of the number of probes) explicit adaptive and non-adaptive schemes respectively for general n using $o(m)$ bits.

4.3 Lower Bounds

Buhrman et al. [BMRV00] have shown that for any (n, s, t)-scheme s is $\Omega(ntm^{1/t})$. As observed in Section 4.2.1, one can achieve this bound easily for $n = 1$. They have also shown that for $n \geq 2$ any two probe non-adaptive scheme must use at least m bits of space. In this section, we show a space lower bound of $\Omega(m^{2/3})$ bits for a restricted class of adaptive schemes using two probes, for $n \geq 2$. Combined with the upper bound of Theorem 4.2.8, this gives a tight lower bound for this class of restricted schemes. We conjecture that the lower bound applies even for unrestricted schemes. We also show a lower bound of $\Omega(m)$ bits for this restricted class of schemes for $n \geq 3$.

Any two-probe $O(s)$ bit adaptive scheme to represent sets of size at most 2 from a universe U of size m, can be assumed to satisfy the following conditions (without loss of generality):

1. It has three tables A, B and C each of size s bits.

2. Each $x \in U$ is associated with three locations $a(x)$, $b(x)$ and $c(x)$.

3. On query x, the query scheme first looks at $A[a(x)]$. If $A[a(x)] = 0$ then it answers 'Yes' if and only if $B[b(x)] = 1$; else if $A[a(x)] = 1$, then it answers 'Yes' if and only if $C[c(x)] = 1$.

4. Each location of A, B and C is looked at by at least two elements of the universe, unless $s \geq m$. (If a location is looked at by only one element, then set that location to 1 or 0 depending on whether the corresponding element is present or not; we can remove that location and the element out of our scheme.)

5. Let $A_i = \{x \in [m] : a(x) = i\}$, $B_i = \{b(x) : x \in A_i\}$ and $C_i = \{c(x) : x \in A_i\}$, for $1 \leq i \leq s$. Then $|B_i| = |A_i|$ or $|A_i| = |C_i|$, for all $1 \leq i \leq s$. I.e., all the elements probing a particular location in table A will all probe distinct locations in one of the tables, B or C. (Otherwise, let $x, y, x', y' \in A_i$, $x \neq y$ and $x' \neq y'$ be such that $b(x) = b(y)$ and $c(x') = c(y')$. Then we can not represent one of the sets $\{x, x'\}$ (when $x \neq x'$) or $\{y, y'\}$ (when $x = x'$).

6. There are at most two ones in B and C put together.

Consider following restrictions on the query scheme:

- R1. For $x, y \in [m], x \neq y$, $a(x) = a(y) \Rightarrow b(x) \neq b(y)$ and $c(x) \neq c(y)$. I.e., any pair of elements probing the same location in A will probe at distinct locations in both B and C.
- R2. For $i, j \in [s], i \neq j$, $B_i \cap B_j \neq \phi \Rightarrow C_i \cap C_j = \phi$. I.e., no two elements will probe the same locations in both B and C.
- R3. Either B or C is all zeroes.

We first show the following relationship between these restrictions:

Lemma 4.3.1 *For any adaptive* $(2, s, 2)$*-scheme, (R1 and R2)* $\Leftrightarrow$ *R3.*

Proof. R3 $\Rightarrow$ (R1 and R2) :

Let x and y $(x \neq y)$ be two elements in $[m]$ such that $a(x) = a(y)$ and $b(x) = b(y)$ (the case when $c(x) = c(y)$, for some x, y, is similar), so that condition R1 is violated. Consider an element $z \neq x$ such that $c(x) = c(z)$; such an element exists by condition 5 above. Now to represent the set $\{y, z\}$, we have set the bits $A[a(x)]$, $B[b(z)]$ and $C[c(y)]$ to 1. But this does not satisfy the condition R3. Thus, R3 $\Rightarrow$ R1.

Again, without loss of generality, let $a(x_1) = a(x_2) = i$, $a(y_1) = a(y_2) = j$, $b(x_1) = b(y_1)$ and $c(x_2) = c(y_2)$, for some elements $x_1, x_2, y_1, y_2 \in [m]$, so that condition R2 is violated. Then, to represent the set $\{x_2, y_1\}$ if we set $A[a(x_1)]$ to 0, then we have to set $A[a(y_1)]$, $B[b(x_2)]$ and $C[c(y_1)]$ to 1. On the other hand, if we set $A[a(x_1)]$ to 1, then we have to set $A[a(y_1)]$ to 0, and $B[b(y_1)]$ and $C[c(x_2)]$ to 1. In both cases, condition R3 is not satisfied. Thus, R3 $\Rightarrow$ R2.

(R1 and R2) $\Rightarrow$ R3 :

Consider any scheme which satisfies R1 and R2 but not R3. So, there exists a set $\{x, y\}$ such that $a(x) \neq a(y)$ for which the scheme stores this set as follows (without loss of generality): $A[a(x)] = 0$, $A[a(y)] = 1$, $B[b(x)] = 1$, $C[c(y)] = 1$, $A[a(z)] = 1$ for all z for which $b(z) = b(x)$, $A[a(z)] = 0$ for all z for which $c(z) = c(y)$ and all other entries are zeroes.

We now argue that this scheme can be converted into a scheme that satisfies R3 also.

Let $a(x) = i$ and $a(y) = j$. If $B_i \cap B_j = \phi$, then we can store this set as follows: $A[a(x)] = A[a(y)] = 0$ and all other entries in A as 1s, $B[b(x)] = B[b(y)] = 1$ and all entries in B and C as zeroes, satisfying R3. Condition R1 (and the fact that $B_i \cap B_j = \phi$) ensures that this is a valid scheme to represent the set $\{x, y\}$.

On the other hand, if $B_i \cap B_j \neq \phi$, then R2 implies that $C_i \cap C_j = \phi$. In this case, to store the set $\{x, y\}$ we can set $A(a(x)) = A(a(y)) = 1, C(c(x)) = C(c(y)) = 1$ and all other entries in A, B and C as zeroes, satisfying R3. ∎

Next we show that if an adaptive $(2, s, 2)$-scheme satisfies R3, then s is $\Omega(m^{2/3})$. Note that the scheme given in Theorem 4.2.8 satisfies all these three conditions. Thus, this lower bound is tight for the class of storage schemes satisfying R3.

Theorem 4.3.2 *If an adaptive* $(2, s, 2)$*-scheme satisfies condition R3, then s is* $\Omega(m^{2/3})$.

Proof. From Lemma 4.3.1, it is enough to show that if an adaptive $(2, s, 2)$-scheme satisfies (R1 and R2), then s is $\Omega(m^{2/3})$.

Observe that R1 implies

$$|A_i| = |B_i| = |C_i|, \ \forall i, 1 \leq i \leq s. \tag{4.3.2}$$

Hence

$$\sum_{i=1}^{s} |B_i| = \sum_{i=1}^{s} |A_i| = m. \tag{4.3.3}$$

By R2, the sets $B_i \times C_i$ are disjoint (no pair occurs in two of these Cartesian products). Thus, by Equation (4.3.2), $\sum_{i=1}^{s} |B_i|^2 \leq s^2$. By Cauchy-Schwarz, $s(\sum_{i=1}^{s} |B_i|/s)^2 \leq \sum_{i=1}^{s} |B_i|^2 \leq s^2$. By Equation (4.3.3), $\sum_{i=1}^{s} |B_i| = m$. Thus, $m^2/s \leq s^2$ *or* $s \geq m^{2/3}$. ∎

We now show that if an adaptive $(n, s, 2)$-scheme, for $n \geq 3$ satisfies R3, then $s \geq m$ (showing another tight lower bound for this class of restricted schemes).

Theorem 4.3.3 *If an adaptive* $(n, s, 2)$*-scheme for* $n \geq 3$ *satisfies condition R3, then* $s \geq m$.

Proof. We first observe that any two probe adaptive scheme satisfies conditions 1 to 5 of the adaptive schemes for sets of size at most 2. Consider an adaptive $(3, s, 2)$-scheme with $s < m$ satisfying R3. One can find up to five elements x, y, y', z and z' from the universe such that $a(y) = a(y')$, $a(z) = a(z')$, $b(x) = b(y)$ and

$c(x) = c(z)$. (Start by fixing x, y, z and then fix x' and y'.) Existence of such a situation is guaranteed by condition 5, as $s < m$.

Now to represent the set $\{x, y', z'\}$, we have to set either $B[b(x)]$ or $C[c(x)]$ to 1. If we set $B[b(x)]$ to 1, then we have to set $A[a(y)]$ to 0 and $C[c(y')]$ to 1. On the other hand, if we set $C[c(x)]$ to 1, then we have to set $A[a(z)]$ to 0 and $B[b(z')]$ to 1. In both the cases, condition R3 is not satisfied. Also note that to represent the set $\{x, y', z'\}$, yielding a contradiction. Hence, $s \geq m$. ∎

Chapter 5

Succinct Dynamic Data Structures

5.1 Introduction

In the last three chapters, we considered succinct structures for problems in which given an input, we would like to support the relevant operations efficiently using as less space as possible. In all these problems, the input is fixed before the data structure is constructed and it does not change during the operations. These are called static structures. In this chapter, we look at data structure problems where the input is allowed to change during the operations. These are called dynamic data structure problems.

We mainly look at succinct solutions to two interrelated classical dynamic data structuring problems, namely maintaining *partial sums* and *dynamic arrays*. We consider these problems in the extended RAM model with word size $\Theta(\lg n)$ bits, where n is the maximum size of the input (though, at some places, we give a more general result by assuming a word size of w bits, where w is a parameter).

In more detail, the problems considered are:

Partial Sums

This problem has two positive integer parameters: the *item size*, $k = O(\lg n)$ and the *maximum increment*, $\delta_{max} = \lg^{O(1)} n$. The problem consists in maintaining a sequence of n numbers $A[1], \ldots, A[n]$, such that $0 \leq A[i] \leq 2^k - 1$ under the operations:

- *sum*(i): return the value $\sum_{j=1}^{i} A[j]$.
- *update*(i, δ): set $A[i] \leftarrow A[i] + \delta$, for some integer δ such that $0 \leq A[i] + \delta \leq 2^k - 1$ and $|\delta| \leq \delta_{max}$.

We also consider the following operation:

- *select*(j): find the smallest i such that *sum*$(i) \geq j$.

In what follows, we refer to the partial sums problem with *select* as the *searchable* partial sums problem. We call the operations that don't change the data structure as *queries* and those that change the data structure as *updates*.

Dietz [Die89] has given a structure for the partial sums problem that supports *sum* and *update* in $O(\lg n/\lg\lg n)$ worst-case time using $\Theta(n \lg n)$ bits of extra space, for the case when $k = \Theta(\lg n)$. As the information-theoretic lower bound on space is kn bits, Dietz's data structure uses a constant factor extra space even when $k = \Theta(\lg n)$, and is worse for smaller k. We modify Dietz's structure to obtain a data structure that solves the searchable partial sums problem in $O(\lg n/\lg\lg n)$ worst-case time using $kn+o(kn)$ bits of space. Thus, we improve the space utilization and support the *select* operation as well.

We show the following trade-off between query and update times: for any parameter $b \geq \lg n/\lg\lg n$, we can support *sum* in $O(\log_b n)$ time and *update* in $O(b)$ time. In particular, we can support *sum* in $O(1)$ time and *update* in $O(n^\epsilon)$ time, for any fixed positive constant $\epsilon \leq 1$. The space used is the minimum possible to within a lower-order term, in all these structures.

When $b = \lg n/\lg\lg n$ or n^ϵ, for some fixed positive constant $\epsilon < 1$, our time bounds are optimal in the following sense. Fredman and Saks [FS89] gave lower bounds for this problem in the *cell probe* model with logarithmic word size, a much stronger model than the word RAM model, we consider. For the partial sums problem, they show that an intermixed sequence of n updates and queries requires $\Omega(\lg n/\lg\lg n)$ amortized time per operation. Furthermore, they give a more general trade-off [FS89, Proof of Thm 3′] between the number of memory locations that must be written and read by an intermixed sequence of updates and queries. Our data structure achieves the optimal trade-off between reads and writes, for the above range of parameter values. If we require that queries be performed using read-only access to the data structure—a requirement satisfied by our query algorithms—then the query and update times we achieve are also optimal.

Next, we consider a special case of the searchable partial sums problem that is of particular interest.

Dynamic Bit Vector

Given a bit vector of length n, support the following operations:

- *rank*(i): find the number of 1's occurring before and including the ith bit
- *select*(j): find the position of jth one in the bit vector and
- *flip*(i): flip the bit at position i in the bit vector

for $1 \leq i, j \leq n$.

A bit vector supporting *rank* and *select* is a fundamental building block for succinct static tree and set representations [BM99, MR97]. Given a (static) bit vector, the *rank* and *select* operations can be supported in $O(1)$ time using $o(n)$ bits of extra space [Cla96, Jac89b] (as described in Section 2.2.3).

As the dynamic bit vector problem is simply the searchable partial sums problem with $k = 1$ (with appropriate translations of the operations *sum*, *select* and *update* to *rank*, *select* and *flip* respectively), we immediately obtain a data structure that supports *rank*, *select* and *flip* operations in $O(\lg n / \lg\lg n)$ worst-case time using $o(n)$ bits of extra space. For the bit vector, however, we are able to exhibit the trade-off for all three operations. Namely, for any parameter $b \geq \lg n / \lg\lg n$ we can support *rank* and *select* in $O(\log_b n)$ time and *update* in amortized $O(b)$ time. In particular, we can support *rank* and *select* in constant time if we allow updates to take $O(n^\epsilon)$ amortized time for any fixed positive constant $\epsilon \leq 1$.

If we remove the *select* operation from the dynamic bit vector problem, we obtain the *subset rank* problem considered by Fredman and Saks [FS89]. From their lower bound on the subset rank problem, we conclude that our time bounds are optimal, in the sense described above.

Next, we consider another fundamental problem addressed by Fredman and Saks [FS89].

Dynamic Array

Given an initially empty sequence of records, support the following operations:

- *insert*(x, i): insert a new record x at position i in the sequence
- *delete*(i): delete the record at position i in the sequence and
- *index*(i): return the ith record in the sequence

for $0 \leq i \leq n$, where n is the current number of records in the array.

Dynamic arrays are useful data structures in efficiently implementing the data types such as the Vector class in Java and C++. The dynamic array problem was called the *list representation* problem by Fredman and Saks, who gave a cell probe lower bound of $\Omega(\lg n/\lg\lg n)$ time for this problem, and also showed that $n^{\Omega(1)}$ update time is needed to support constant-time queries. The *tiered vector* data structure of Goodrich and Kloss [GI99] (for the dynamic array problem) supports *insert* and *delete* operations in $O(n^\epsilon)$ amortized time while supporting the *index* operation in constant worst-case time. This structure uses $O(n^{1-\epsilon})$ words of extra space (besides the space required to store the n elements of the array), for any fixed positive constant $\epsilon \leq 1$.

We first observe that the structure of Goodrich and Kloss can be viewed as a version of the well-known implicit data structure, the 'rotated list' [MS80, Fre83]. Using this connection, we observe that the structure of Goodrich and Kloss can be made to take $O(n^\epsilon)$ *worst-case* time for updates while maintaining the same storage ($O(n^{1-\epsilon})$ additional words) and $O(1/\epsilon)$ worst-case time for the *index* operation, for any parameter $0 < \epsilon \leq 1$. Then, using this structure in small blocks, we obtain a dynamic array structure that supports *insert*, *delete* and *index* operations in $O(\lg n/\lg\lg n)$ amortized time using $o(n)$ bits of extra space. Due to the lower bound result of Fredman and Saks, both our results above are on optimal points of the query time/update time trade-off while using optimal (within lower order term) amount of extra space.

We should also point out that the resizable arrays of Brodnik et al. [BCD+99] can be used to support inserting and deleting elements at either ends of an array and accessing the ith element in the array, all in constant time. This data structure uses $O(\sqrt{n})$ words of extra space, where n is the current size of the array. Brodnik et al. do not support insertion into (or deletion from) the middle of the array.

To simplify notation, we ignore integer roundings at places where they do not affect the asymptotic analysis.

In Section 5.2, we describe our space efficient structures for the partial sums problem. In Section 5.3, we look at the special case of the partial sums problem when the given elements are bits, and give the details of a structure that supports full tradeoff between queries (*select* and *rank*) and update (*flip*). Section 5.4 addresses the problem of supporting the dynamic array operations.

5.2 Dynamic Structures for Partial Sums

Here, we consider the partial sums and the searchable partial sums problems and give trade-offs between query and update times. Then for the dynamic bit vector problem we give some more query-update trade-offs. We begin by describing a weight balanced B-tree which is used later.

5.2.1 Weight Balanced B-Trees

We first reproduce the definition of a weight balanced B-tree or a WBB-tree, as described by Dietz [Die89].

Given a list (sequence) of elements a WBB-tree is a rooted, ordered tree having the following properties:

- The elements of the list are at the leaves of the tree, in the order from left to right.
- The leaves of the tree are at the same depth.
- Let b be the branching factor of the tree and let N be a value chosen so that $1/2 < N/w(root) < 2$.

  Define the fullness of a node x to be the quantity $full(x) = w(x)/b^{h(x)}$

  For every internal node x except the root, $1/2 < full(x) < 2$ and,

  $full(root) < 2$.

Here, for a node x, $h(x)$ refers to the height of x and $w(x)$, called the weight of x, is defined as the number of leaves in the subtree rooted at x.

WBB-trees are closely related to ordinary B-trees [Meh84]. The weight balancing condition enables us to support the index operation efficiently.

Note that for a given list of length n, the height of the WBB-tree with branching factor b is $O(\log_b n)$. Dietz [Die89] has used this structure to describe an optimal algorithm for the list indexing problem. In particular, he shows that this structure supports finding the position of a given element in the list (index operation), inserting and deleting an element from the list all in $O(\lg n / \lg\lg n)$ amortized time (by taking $b = \lg n / \lg\lg n$). The space required by this structure is $O(n \lg n)$ bits.

We first observe that this can be easily generalized to get a structure that supports index operation in $O(\log_b n)$ amortized time and insert and delete operations in $O(b \log_b n)$ amortized time, using the same amout of space, for any parameter $4 \leq b \leq n$. We also use another simple generalization where we associate numbers with the leaves and define the weight of a leaf to be the weight associated with it. Thus, given a sequence of n elements, we use this modified structure to support *sum* and *select* operations in $O(\log_b n)$ time and the operations of incrementing and decrementing the weight of a leaf in $O(b \log_b n)$ time. This modified structure also takes $O(n \lg n)$ bits of space.

5.2.2 Searchable Partial Sums

We begin by solving the partial sums problem on a small set of integers in $O(1)$ time, by adapting an idea of Dietz [Die89].

Lemma 5.2.1 *On a RAM with a word size of w bits, we can solve the searchable partial sum problem on a sequence of $n = w^\epsilon$ numbers, for any fixed $0 \leq \epsilon < 1$, with item size $k \leq w$, in $O(1)$ worst-case time using $O(nw)$ bits of space. The data structure requires a precomputed table of size $O(2^{\epsilon' w})$ bits for some fixed $\epsilon' < 1$.*

Proof. Let $A[1], \ldots, A[n]$ denote the sequence of elements for which the partial sums are to be calculated. We store another array $B[1, \ldots, n]$ which contains the partial sums of A, i.e., $B[i] = \sum_{j=1}^{i} A[i]$. As we cannot hope to maintain B under the *update* operation (with $O(1)$ time per operation), we let B get slightly 'out of date'. More precisely, B is not changed after each *update*; instead, after every m *updates* B will be *refreshed*, or brought up to date. Since the cost of refreshing is $O(n)$, the amortized cost is $O(1)$ per *update*.

We also maintain an array $C[1, \ldots, n]$ in addition to A and B. C is set to all zeroes when B is refreshed. Otherwise, when an *update* changes $A[i]$ by δ, i.e., if $A[i]$ is changed to $A[i] + \delta$, we set $C[i] \leftarrow C[i] + \delta$. Since $|C[i]| \leq n\delta_{max} = w^{O(1)}$ always, the entire array C occupies $O(n \lg w)$ bits, which is less than θw bits, for any fixed $0 < \theta < 1$ and sufficiently large w. Now, *sum*(i) is nothing but $B[i] + \sum_{j=1}^{i-1} C[i]$.

We construct a precomputed table, that has one entry for each possible configuration of the array C written as a bit string, and for each possible index i, $1 \leq i \leq n$. The entry stored for the array configuration C and index i is the value $\sum_{j=1}^{i-1} C[i]$. We store these these entries in the lexicographic order of the bit strings C; and in the increasing order of i within the entries with the same C. Thus, the value $\sum_{j=1}^{i-1} C[i]$ can be computed in constant time by indexing into this precomputed table. Since there are at most $2^{\theta w}$ possible configurations of C, the space used by the table is at most $2^{\theta w} n \lg(n^2 \delta_{max})$ bits, which will be $O(2^{\epsilon' w})$ bits for some fixed $\epsilon' < 1$, as $\theta < 1$ and $m = w^\epsilon$.

We now show how to perform *select* in $O(1)$ time. The idea is to first find an approximate index (for *select*) and then find a corrective term to find the exact index. For finding the approximate index, we use the *Q-heap* structure given by Fredman and Willard [FW94], which solves the following dynamic predecessor problem:

Theorem 5.2.2 *[FW94] For any* $0 < M < 2^w$*, given a set of at most* $(\lg M)^{1/4}$ *integers of* $O(w)$ *bits each, one can support the operations insert, delete, predecessor and successor operations in constant time on a RAM with word size* w *bits, where predecessor*(x) *(successor*(x)*) returns the largest (smallest) element* y *in the set such that* $y < x$ $(y \geq x)$*. The data structure requires a precomputed table of size* $O(M)$ *bits.*

By choosing $M = 2^{\epsilon' w}$, we can support predecessor/successor queries on sets of size $n' = (\epsilon' w)^{1/4}$ in $O(1)$ time, using a table of size $O(2^{\epsilon' w})$ bits. Now, given a sequence of length $n = w^\epsilon$, if $\epsilon \geq 1/4$, then we construct a tree with branching factor n' with the given sequence of elements at the leaves. The height of this tree will be $O(1)$. Define the weight of an internal node to be the sum of the weights of its children and that of a leaf to be the value in the given sequence associated with it. Now, at each internal node, we store the partial sums of the weights of its children using the Q-heap structure of Theorem 5.2.2 to support the predecessor and successor operations on its children in $O(1)$ time. Note that the updates to the Q-heap structure will have $O(1)$ amortized cost. This tree structure supports predecessor and successor queries on the array B.

If the array B were up to date, then we can answer a query for *select*(j), for any j, in $O(1)$ time by finding the successor of j in B using the above tree structure.

However, B may not be up-to-date (but not too out-dated either, as it will be up to date after every n updates). Thus, this gives an approximate index for *select*(j). To find the exact value from this, we do the following:

Let $D[1,\ldots,n]$ be an array such that $D[i] = \min\{A[i], n\delta_{max}\}$. As with C, D is also stored in a single word, and is changed in $O(1)$ time (using either table lookup or bitwise operations) whenever A is changed by an *update*. To implement *select*(j), we first consult the Q-heap data structure to determine an index t such that $B[t-1] < j \leq B[t]$. By calculating $sum(t-1)$ and $sum(t)$ in $O(1)$ time we determine whether t is the correct answer. In general, the correct answer could be an index $t' \neq t$; assume for specificity that $t' > t$. Note that t' is the smallest integer such that $A[t+1]+\cdots+A[t'] \geq j - sum(t)$. Since $j \leq B[t]$ and $B[t] - sum(t) \leq n\delta_{max}$, it follows that $j - sum(t) \leq n\delta_{max}$. By the definition of D, it also follows that t' is the smallest integer such that $D[t+1] + \ldots + D[t'] \geq j - sum(t)$.

Now, we store a precomputed table in which each entry is indexed by a triple of the form (D, x, i) where D is any possible configuration of the array D, $1 \leq x \leq n\delta_{max}$, and $1 \leq i \leq n$. The entry corresponding to the triple (D, x, i) stores the smallest index j such that $D[i+1] + \ldots + D[j] \geq x$. These entries are ordered in the lexicographic order of the triples (D, x, i), so that one can index into the table in constant time. Thus, given D, $j - sum(t)$ and t, one can look up this precomputed table to find t' in $O(1)$ time. A similar procedure is followed if $t' < t$. The space used by this precomputed table is at most $2^{\theta w} n \lg n$ which is $O(2^{\epsilon' w})$ bits, for some constant $\epsilon' < 1$.

Finally, the amortization can be eliminated by Dietz's incremental refreshing approach [Die89]. More specifically, instead of updating the entire arrays B and D after every m *update*s, we update one location in each of them after every *update*, in some fixed predetermined order. Thus, each value of the arrays B and D will be updated once in every m *update*s, and hence the above procedures to compute *sum* and *select* still give the correct answers. ∎

For larger inputs (of length n), we choose a parameter $b = (\lg n)^{\epsilon}$, for some positive constant $\epsilon < 1$ and create a complete b-ary tree, the leaves of which correspond to the entries of the input sequence A. We define the *weight* of an internal node as the sum of the weights of its children and the weight of a leaf to be the input value associated with it. At each internal node we store the weights of its children using the data structure of Lemma 5.2.1. The tree has $O(n/b)$ internal nodes, each occupying $O(b)$ words (of $(\lg n)$ bits each) and the height of the tree is $O(\lg n / \lg \lg n)$.

We now briefly outline how each of the operations can be supported. To compute $sum(i)$, we consider the path from root to the i^{th} leaf (from the left) in the tree, by starting from the leaf and going up using the parent pointers. (Note that, since the tree is a complete b-ary tree having all the leaves at the same level, one can find i^{th} leaf using simple arithmetic.) In this path, for each internal node, we find the partial sum of its child (in the path) and add them up to get $sum(i)$.

To compute $select(j)$, we start from the root and compute $select(j)$ in the structure stored at the root. If this returns an index i, then we find $x = sum(i)$ in the partial sum structure for the root and then recursively find $select(j - x)$ at the i^{th} child of the root.

To update the i^{th} element in the sequence, we start from the i^{th} leaf in the tree and update partial sum structures stored at all the nodes in the path from the leaf to the root.

Clearly all these operations take time proportional to the height of the tree as they traverse the path from the root to a leaf, spending $O(1)$ time at each level. Thus, we have

Lemma 5.2.3 *There is a data structure for the searchable partial sums problem on n elements that supports all operations in $O(\lg n / \lg \lg n)$ time, and requires $O(n)$ words of space.*

We now modify this structure, reducing the space complexity of the structure to $kn + o(kn)$ bits. We first show some trade-offs between the query and update times.

5.2.3 Trade-off between *query* and *update* for Partial Sums

Lemma 5.2.4 *For any parameter $b \geq 4$, there is a data structure for the searchable partial sums problem that supports update in $O(\log_b n)$ time and sum and select in $O(b \log_b n)$ time. The space used is $kn + O((k + \lg b) \cdot n/b)$ bits.*

Proof. We construct a complete b-ary tree over the elements of the input sequence A. With each internal node we store its weight (as defined above). Since the weight of a node at height h is in the range $[0, \ldots, 2^k b^h - 1]$, we need $k + h \lg b$ bits to store the weight of a node at level h. Summing up, we need $O(k + \lg b)n/b$ bits to store the weight of all the internal nodes of the tree and kn bits to store the values (weights) at the leaves. To perform an *update*, we start from the leaf and update the weights of all the nodes in the path from that leaf to the root. Since the height of the tree is $O(\log_b n)$, this requires $O(\log_b n)$ time. The operations *sum* or *select* can be implemented by traversing the tree from the root to a leaf, looking at all the children of a node at each level. This requires $O(b \log_b n)$ time to support these operations. (Note that *select* can in fact be supported in $O(\lg n)$ time by performing a binary search for the appropriate value at node in the path from the root to a leaf.) ∎

Now, to support all the operations in $O(\lg n/ \lg\lg n)$ time, we divide input into groups of numbers of size $(\lg n)^2$ each. These groups are represented internally using Lemma 5.2.4, with $b = (\lg n)^{1/2}$. This requires $kn + o(kn)$ bits, and all operations within a group take $O((\lg n)^{1/2})$ time, which is negligible. The $n/(\lg n)^2$ group sums are stored using the data structure of Lemma 5.2.3, which requires $o(n)$ bits now. The precomputed tables (required in Lemma 5.2.1) also require $o(n)$ bits. Thus we have:

Theorem 5.2.5 *There is a data structure for the searchable partial sums problem that supports all operations in $O(\lg n/ \lg\lg n)$ worst-case time and uses $kn + o(kn)$ bits of space.*

Combining Lemma 5.2.4 with Theorem 5.2.5 and noting that given any parameter $b \geq \lg^2 n$, we can reduce the branching factor from b to $b/\lg n$. This reduces the complexity of queries to $O(b)$ without affecting the asymptotic complexity of *update*. This gives us the following:

Theorem 5.2.6 *There is a data structure for the searchable partial sums problem that supports update in $O(\log_b n)$ and sum and select in $O(b)$ time, for any parameter $b \geq \lg n/ \lg\lg n$, using $kn + o(kn)$ bits of space.*

We now show that one can trade off query and update times for the partial sums problem (without *select*). More specifically, we show that for any parameter $2 \leq b \leq n$, we can support *sum* in $O(\log_b n)$ and *update* in $O(b \log_b n)$ time, while still ensuring that the data structure is space efficient. As these bounds are subsumed by Theorem 5.2.5 for $b \leq (\lg n)^2$, we will assume that $b > (\lg n)^2$ in what follows.

We construct a complete tree with branching factor b, with the given sequence of n elements at the leaves. Clearly this tree has height $h = \log_b n$. At each internal node, we store the *weight* of that node and also store an array containing the partial sums of the weights of all its children. By using the obvious $O(b)$ time algorithm, the partial sum array at an internal node is kept up-to-date after each *update*. Thus, *update* requires $O(b \log_b n)$ time. The *sum* operation can be supported in $O(\log_b n)$ time by traversing the path from a leaf to the root and computing the partial sums at each level (as described before). Unfortunately, the space used to store this 'simple' structure is $O((k + \lg b)n)$ bits, since the partial sum arrays at an internal nodes at height h requires $O(k + h \lg b)$ bits of space and the weights of the leaves take kn bits of space.

To get around this, we use one of two methods, depending on the value of k. If $k \geq (\lg n)^{1/2}$, then we divide the input values into groups of size $\lg n$. Within a group, we do not store the $A[i]$'s explicitly, but store only their partial sums. The sums of elements in each of the $n/\lg n$ groups are stored in the simple structure above with branching factor $b = \lg n$. The space required by this structure is $O((k + \lg\lg n)n/\lg n)$ which is $o(kn)$ bits. The space required by each group is $(k + \lg\lg n)\lg n$ bits; this sums up to $kn + n\lg\lg n = kn + o(kn)$ bits overall. Clearly the asymptotic complexity of *update* and *sum* are not affected by this change.

If $k < (\lg n)^{1/2}$, then we divide the given sequence of elements into groups of size $\lg n/2k$ each. Again, group sums are stored in the simple structure, which requires $O(kn(k + \lg\lg n)/\lg n) = o(kn)$ bits. We store a precomputed table that has an entry corresponding to each possible configuration of an array of length $(\lg n)/2k$, where each entry is a k bit string, and an index i, $1 \leq i \leq (\lg n)/2k$. The entry corresponding to an array configuration D and an index i is the value $\sum_{j=1}^{i} D[j]$. Again, these entries are stored in the lexicographic order of the strings (D, i) to enable constant time time access into this table. The space required to store this table is at most $\sqrt{n}k(\lg\lg n)^2$ which is $o(n)$ bits, for the given range of k. Thus, we can answer *sum* queries within a group by indexing into this table.

Finally, we note that given any parameter $b \geq (\lg n)^2$, we can reduce the branching factor from b to $b/\lg n$ without affecting the asymptotic complexity of *sum*; however, *update* would now take $O(b)$ steps. Combining this with Theorem 5.2.5 we have:

Theorem 5.2.7 *For any parameter* $\lg n/\lg\lg n \leq b \leq n$, *there is a data structure* *for the partial sums problem that supports sum in $O(\log_b n)$ time and update in $O(b)$ time, and uses $kn + o(kn)$ bits of space.*

Note that using the above structure, one can support *select* in $O(\lg n)$ time by performing a binary search (on the partial sums of the weights of the children) at each node in a path from the root to a leaf for the required element.

5.3 Dynamic Bit Vector

The dynamic bit vector problem is a special case of the searchable partial sums problem. The following corollary follows from Theorem 5.2.5.

Corollary 5.3.1 *Given a bit vector of length n, we can support the rank, select and flip operations in $O(\lg n/\lg\lg n)$ time using $o(n)$ bits of space in addition to the bit vector.*

5.3.1 Trade-off between *query* and *update* for Bit Vector

Theorem 5.2.6 immediately gives the following trade-offs between the query and update times for the dynamic bit vector:

Theorem 5.3.2 *Given a bit vector of length n, we can support $flip$ in $O(\log_b n)$ and rank/select in $O(b)$ time, for any parameter $b \geq \lg n/\lg\lg n$, using $o(n)$ bits of extra space.*

Similarly, Theorem 5.2.7 immediately gives the following trade-off result (the only thing to observe is that, out of the two cases in Theorem 5.2.7, we apply the one that stores the input sequence explicitly, i.e., the case when $k < (\lg n)^{1/2}$).

Corollary 5.3.3 *For any parameter $\lg n/\lg\lg n \leq b \leq n$, there is a data structure for the dynamic bit vector problem that supports rank in $O(\log_b n)$ time and flip in $O(b)$ time, using $o(n)$ bits of space in addition to the bit vector.*

Also, note that using this structure we can support *select* in $O(\lg n)$ time, by performing a binary search (on the partial sums) to find the child containing the required leaf at each node. We now show that this can be improved to $O(\log_b n)$ (though we don't know how to do this for the general partial sums problem). We first note the following proposition.

Lemma 5.3.4 *The operations select and flip can be supported in $O(1)$ time on a bit vector of size $N = (\lg n)^{O(1)}$ on a RAM with word size $O(\lg n)$ using a fixed precomputed table of size $o(n)$ bits. The space required is $o(N)$ bits in addition to the precomputed table and the bit vector itself.*

Proof. Store the bit vector at the leaves of a balanced tree with branching factor $\sqrt{\lg n}$ (one bit per leaf). With each internal node, we keep the sequence of weights of its children using searchable partial sum structure of Lemma 5.2.1. To perform a *flip* operation, we flip the bit at the leaf level using bit wise operations and then change the weights of its ancestors accordingly.

To perform *select*, we start from the root and follow the path down by selecting the appropriate node at each level by using the partial sum structure stored, until we reach a node that is one level above the leaf nodes. At these nodes (which are one level above the leaf nodes), we need to support *select* in a bit vector of length $\sqrt{\lg n}$. This can be done using a precomputed table of size $o(n)$ bits in constant time as explained below. The table has an entry corresponding to each possible bit string of length $\sqrt{\lg n}$ and an index i, $1 \leq i \leq \sqrt{\lg n}$, which stored the position of the i^{th} 1 in the bit string. These entries are stored in the lexicographic order (of the concatenation of their bit representations) to enable constant time access to the table.

Since the height of the tree is a constant, *select* and *flip* can be supported in constant time. The number of nodes in the tree excluding the nodes in the last two levels (leaves and the nodes one level above) is $O(N/\lg n)$. With each of these nodes, we have stored a searchable partial sum structure of Lemma 5.2.1. Hence, the total space used to store all these structures put together is at most $N \lg\lg n/\sqrt{\lg n}$, apart from the space required to store the precomputed tables for these structures which is $o(n)$ bits. ∎

We now show how to support *select* in $O(\log_b n)$ time if *flip* takes $O(b)$ time, for any parameter $(\lg n)^4 \leq b \leq n$.

The structure mainly consists of following sub-structures:

1. We divide the bit vector into *superblocks* of size $\lg^4 n$. With each superblock we store the number of ones in it. The sequence of these superblock counts is stored using the data structure of Theorem 5.2.7 with the same value of b. This structure enables us, in $O(\log_b n)$ time, to look up the number of ones to

the left of any given superblock. The space used by this structure is $o(n)$ bits as the tree size is $O(n/\lg^4 n)$.

2. We store each of the superblocks using the structure of Lemma 5.3.4. The total space used to store these structures for the superblocks is again $o(n)$ bits.

3. In addition, we divide the ones in the bit vector into groups of $\Theta(\lg^2 n)$ successive ones each. Call the first element of a group as its 'leader'. A group's size is allowed to vary between $(\lg^2 n)/2$ and $2\lg^2 n$. Consider the sequence containing the number of ones in each group (in the left to right ordering of their occurrence in the bit vector). We store this sequence using a WBB-tree (described in Section 5.2.1, with the weight of a leaf being the number associated with it) with branching factor b. Again this structure takes $o(n)$ bits, as the number of nodes in the tree is $O(n/\lg^2 n)$. Given an integer j, using this structure we can locate the group in which the j^{th} one lies in $O(\log_b n)$ time, and support changes due to *flip*s in $O(b \log_b n)$ amortized time.

4. With each group, we store the index of the superblock in which the group's leader lies. Also, with each superblock, we store all group leaders which lie in that superblock. The total space used here is $O(n/\lg n)$, which is $o(n)$ bits.

5. Let the *span* of a group be the number of superblocks other than the superblock containing its leader in which the group elements lie, i.e., it is the index of the superblock in which the next group's leader lies minus the index of the superblock in which its group leader lies. If the span of a group is at least 2, we call it a *sparse* group. With each sparse group, we store the bit positions of all the 1s in the group in a dynamic array. Since the maximum size of this array is $O(\lg^2 n)$, using either the implementation of Goodrich and Kloss [GI99] or the implementation of Corollary 5.4.2, we get a dynamic array which allows insertions and deletions in $O(\lg n)$ time and accesses in $O(1)$ time. This requires $O(\lg^3 n)$ bits per sparse group, but there can only be $O(n/\lg^4 n)$ sparse groups, so the total space used here is $O(n/\lg n)$, which is $o(n)$ bits.

To compute *select*(j), we first locate the group in which the j^{th} one lies in $O(\log_b n)$ time using the WBB-tree structure. We also find the number of ones up to the group leader of that group. Thus, we are left with the problem of finding an appropriate one bit in a given group. This is done as follows:

If the given group is sparse, then we look up the dynamic array associated with the group, in which we have stored the positions of all the one bits in that group. This takes $O(1)$ time. Otherwise (if the group is not sparse), we find the superblock in which the group leader lies. Now, the position we are looking for must belong to this or the next superblock (as the next group starts in one of these superblocks). This can be answered in $O(1)$ time, as each superblock is stored using the structure of Lemma 5.3.4. Thus, *select* can be supported in $O(\log_b n)$ time.

To perform *flip*(j), we first flip the j^{th} bit in the bit vector. We also update the partial sum structure stored for the superblocks (item 1) and the structure corresponding to the superblock to which the j^{th} bit belongs (item 2). This takes $O(b)$ amortized time. Next, we locate the group to which position j (of the bit vector) would belong, by performing the operation *select*$($*rank*$(j))$ on the WBB-tree storing the group sums (item 3) and then update (increment or decrement depending on whether the j^{th} bit was initially 0 or 1) the WBB-tree. This takes $O(b \log_b n)$ amortized time. Note that group splits and merges would be taken care of by the amortization in the WBB-tree. If the group leader (of the group to which the j^{th} bit belongs to) changes, then we update it in constant time. Also, we can update the indices of group leaders stored for the superblock (to which the j^{th} bit belongs to) in constant amortized time. This takes care of item 4 above.

Finally, we need to update the dynamic array structure stored for the group, if it is sparse. (Note that if the group becomes sparse after a deletion or becomes dense after an insertion, we are allowed to spend time linear, in the length of the dynamic array, amount of space and construct or free the dynamic array, as these changes happen only after $O(\lg^2 n)$ updates.) For this, we first find the value $i =$ *rank*$(j) -$ *rank*(r_0), where r_0 is the group leader of the group to which the j^{th} bit belongs. We then perform the operation *insert*(j) or *delete*(j) on the dynamic array structure stored for the (sparse) group (item 5) depending on whether the j^{th} bit was initially 0 or 1. It takes $O(\log_b n)$ time to compute i and $O(\lg^2 n)$ time to update the array.

Thus, *flip* takes $O(b \log_b n)$ amortized time.

Again as $b \geq \lg^4 n$, we can actually make the branching factor to be $b/\lg n$. For other values of b, using Corollaries 5.3.1 and 5.3.3, we get:

Theorem 5.3.5 *Given a bit vector of length n, we can support the rank and select operations in $O(\log_b n)$ time and flip in $O(b)$ amortized time, for any parameter $b \geq \lg n / \lg\lg n$, using $o(n)$ bits of extra space.*

5.4 Dynamic Arrays

We look at the problem of maintaining an array structure under the operations of insertion, deletion and indexing. Goodrich and Kloss [GI99] have given a structure that supports (arbitrary) *insert* and *delete* operations in $O(n^\epsilon)$ amortized time and *index* operation in $O(1)$ worst-case time using $o(n)$ bits of extra space to store a sequence of n elements. Here, first we describe a structure that essentially achieves the same bounds above (except that we can now support updates in $O(n^\epsilon)$ worst-case time) using a well known implicit data structure called *recursively rotated list* [MS80, Fre83]. Using this as a basic block, we will give a structure that supports all the dynamic array operations in $O(\lg n / \lg\lg n)$ amortized time using $o(n)$ bits of extra space.

We assume a memory model in which the system returns a pointer to the beginning of a block of requested size and hence any element in a block of memory can be accessed in constant time given its index within the block and the block pointer. This is the same model used in the resizable array of Brodnik et al. [BCD$^+$99].

Rotated lists [MS80] were discovered to support dictionary operations implicitly, on a totally ordered set. A (1-level) rotated list is an arbitrary cyclic shift of the sorted order of the given list. We can search for an element in a rotated list on n elements in $O(\lg n)$ time by a modified binary search, though updates (replacing one value with another) can take $O(n)$ time. However, replacing the largest (smallest) element with an element smaller (larger) than the smallest (largest) can be done in $O(1)$ time if we know the position of the smallest element in the list.

A 2-level rotated list consists of elements stored in an array divided into blocks where the i-th block is a rotated list of i elements. It is easy to see that such a structure containing n elements has $r = O(\sqrt{n})$ blocks. In addition, all the elements of block i are less than every element of block $i+1$, for $1 \leq i < r$. This structure supports searches (for an element) in $O(\lg n)$ time and updates (*insert* and *delete*) in $O(\sqrt{n})$ time, if we also explicitly store the position of the smallest element in each block (otherwise the updates take $O(\sqrt{n} \lg n)$ time). This is easily generalized to an

l-level rotated list where searches take $O(2^l \lg n)$ time and updates take $O(2^l n^{1/l})$ time [Fre83]. The structure uses $O(n^{1-1/l})$ pointers to store the positions of smallest elements in each block.

To use this structure to implement a dynamic array, we do the following. We simply store the elements of the array in a rotated list based on their order of insertions. We also keep the position of the first element in each recursive block. Since we know the size of each block, $index(i)$ operation just takes $O(l)$ time in an l-level rotated list implementation of a dynamic array. Similarly, inserting/deleting at a given position i can be performed as done in a rotated list, taking $O(2^l n^{1/l})$ time. Thus we have,

Theorem 5.4.1 *A dynamic array having n elements can be implemented using an l-level rotated list such that queries can be supported in $O(l)$ worst-case time and updates in $O(2^l n^{1/l})$ worst-case time using an extra space of $O(n^{1-1/l})$ pointers, for any integer parameter $1 \le l \le \lg n$.*

Choosing $l = 1/\epsilon$, we get

Corollary 5.4.2 *A dynamic array containing n elements can be implemented to support queries in $O(1)$ worst-case time, and updates in $O(n^\epsilon)$ worst-case time using $O(n^{1-\epsilon})$ pointers, where ϵ is any fixed positive constant.*

Using this structure, we now describe a structure that supports all the dynamic array operations in $O(\lg n / \lg\lg n)$ amortized time using $o(n)$ bits of extra space.

We divide the given list of length n into sub-lists of length $\Theta(\lg^4 n)$. In particular, each sub-list will be of length between $\frac{1}{2}\lg^4 n$ and $2\lg^4 n$. We implement each of these sub-lists using the dynamic array structure of Corollary 5.4.2, with $\epsilon = 1/8$. These arrays support updates in $O(\sqrt{\lg n})$ time and access in constant time. The total extra space required to store all these dynamic arrays is $O(n/\sqrt{\lg n})$ bits.

Next, consider the sequence obtained by storing the sizes of these sub-lists in the order in which they appear in the array. We construct a WBB-tree on this sequence (described in Section 5.2.1, with the weight of a leaf being the number associated with it) with branching factor $\sqrt{\lg n}$. The space required to store this tree is $o(n)$ bits, as the number of nodes in the tree is $O(n/(\lg^4 n))$.

Now, to access an element, we first use the WBB-tree to find the sub-list to which the element belongs and then use the dynamic array corresponding to that block to access the element. Thus queries take $O(\lg n / \lg\lg n)$ time.

To update the array, we first update the WBB-tree using $O(\lg n/\lg\lg n)$ amortized time. Then we find the sub-list in which the update has to be performed (using the WBB-tree) and update the dynamic array corresponding to that sub-list. Thus, it takes $O(\lg n/\lg\lg n)$ amortized time to perform an update. Thus we have

Theorem 5.4.3 *A dynamic array can be implemented using $o(n)$ bits of extra space besides the space used to store the n records, in which all the operations can be supported in $O(\lg n/\lg\lg n)$ amortized time, where n is the current size of the array.*

Chapter 6

Conclusions

We have considered succinct representations of *suffix trees*, *suffix arrays*, *static dictionaries* supporting membership and rank, *cardinal trees*, a list of numbers to support *partial sums* queries, *dynamic bit vectors* and *dynamic arrays* in the extended RAM model with appropriate word sizes arising naturally from the problem instances. We have also considered the problem of static dictionary in the bitprobe model. Here we summarize our main results, describe further work and mention some open problems.

6.1 Summary, Related Work and Open Problems

6.1.1 Succinct Structures for Indexing

We gave a suffix tree representation for a text of length n over an alphabet Σ, that can be stored using $n \lg n + 4n + o(n)$ bits of space. Given a pattern of length m, this structure supports finding an occurrence of the pattern in the text (search query) in $O(m \lg |\Sigma|)$ time. The main idea here is to use the succinct representation of binary trees to represent the tree structure of a suffix tree. Counting the number of occurrences of the pattern in the text (counting query) can also be supported in the same amount of time with no extra space. Finding all the occurrences of the pattern in the text (enumerative query) can be supported with an additional $O(occ)$ time, where occ is the number of occurrences of the pattern in the text.

For binary texts, we obtained two space efficient structures: one structure uses $\frac{n}{2}\lg n + O(n)$ bits of space and supports search queries in $O(m)$ time. The other structure requires $O(n \lg n / \lg\lg n)$ bits of space and supports existential queries (finding whether a pattern occurs in the text) in $O(m)$ time. This was the first structure to break the $O(n \lg n)$ bottleneck on space though it supports only existential queries. Each of these structures uses a different technique, either in the storage scheme or in the search algorithm, to reduce the space requirement. The first structure constructs a sparse suffix tree for all the suffixes that start with the bit that occurs most number of times in the given binary text. The second structure uses an iterative algorithm to search for the pattern. Since the first appearance [MRR98] of these results, several improvements have appeared and we briefly describe some of them here.

Building on our suffix tree representation and using other techniques, Grossi and Vitter [GV00] have developed an $O(n)$ bit index structure for a given binary text that answers search queries in $o(m)$ time — $O(m/\lg n + \lg^{\epsilon} n)$ when m is $\Omega(\lg n)$ and $O(1)$ otherwise. Enumerative queries, using this structure, can be supported in $O(m/\lg n + occ \lg^{\epsilon} n)$ time, where *occ* is the number of occurrences of the pattern in the text and ϵ is any fixed positive constant less than 1. Their representation is based on an $O(n)$-bit representation of a compressed suffix array.

Extending the ideas of Grossi and Vitter, we gave a compressed suffix array representation for a given binary text of length n that can be stored using $O(nt(\lg n)^{1/t})$ bits of space, which answers *lookup* queries in $O(t)$ time, for any parameter $1 \leq t \leq \lg\lg n$. In particular, this gives an indexing structure for a binary text of length n that uses $O(n \lg^{\epsilon} n)$ bits of space and answers indexing queries in optimal $O(1+m/\lg n)$ time. Finding all the occurrences of the pattern requires an additional $O(occ)$ time. This gives the first $o(n \lg n)$ bit structure that supports enumerative queries in optimal time.

More recently, Mäkinen [Mäk00, Mäk01] has given a structure, called *compact suffix array*, that takes less space than a suffix array (though no explicit bound was shown) and supports search and counting queries in $O(m + \lg n)$ time. He used the idea of representing regions of a suffix array from previous regions by incrementing the corresponding entries using a pointer from the later regions to the previous regions (similar to Ziv-Lempel data compression [ZL77]), to reduce the size of the suffix array.

Ferragina and Manzini [FM00, FM01] have given an *opportunistic* data structure for the indexing problem and have shown its experimental performance. They describe the space occupancy of their structure to be optimal in an information-content sense, as the given text is stored in space linear in the k-th order *entropy* of text, for any fixed k. Search and counting queries, using this structure, can be supported in $O(m)$ time and enumerative queries in $O(m + occ \lg^{\epsilon} n)$ time, for any fixed $0 < \epsilon < 1$.

Sadakane [Sad00] has proposed a *compressed text database* using the compressed suffix array of Grossi and Vitter. The space occupancy of this compressed database is linearly proportional to the 0-th order entropy of the text. This structure supports search and counting queries in $O(m \lg n)$ time and enumerative queries in $O(m \lg n + occ \lg^{\epsilon} n)$ time. This also supports the *decompress* operation, which returns a substring of length l in the compressed database in $O(l + \lg^{\epsilon} n)$ time and the *inverse* operation which returns the inverse of the suffix array.

Some interesting open problems in this area are:

- Is there a compressed suffix array representation that uses $O(n)$ bits and supports *lookup* queries in constant time (or can one prove that no such structure exists)? Due to a lower bound result of Demaine and López-Ortiz [DL01], $\Omega(n)$ bits are required by any structure that answers indexing queries in $O(m)$ time.

- One obvious way to construct our suffix tree representations is to construct the usual suffix tree first and then construct the parenthesis representation of the tree from it. However, this method uses more space during the construction phase than is required by the final structure. Can one avoid this problem?

6.1.2 Static Dictionary Supporting Rank

We gave a static dictionary representation that supports membership and rank (for the elements present) queries in constant time (called a rank dictionary) for a subset of size n from a universe of size m that uses $n \lceil \lg m \rceil + O(\lg \lg m)$ bits of space. Using this as a substructure and using the techniques of *universe reduction* and *sharing primes*, we got a structure that stores a set of dictionaries with total cardinality n over a universe of size m, using $n \lceil \lg m \rceil + O(\lg \lg m)$ bits of space. This structure supports rank and membership queries on individual dictionaries in constant time. We then used this structure to get a representation of a k-ary cardinal tree on n nodes that uses $2n + n\lceil \lg k \rceil + o(n) + O(\lg\lg k)$ bits of space and supports 'parent', 'i^{th} child', 'child labeled j', 'degree' and 'subtree size' operations in constant time. This cardinal tree representation can be used to get a suffix tree structure that takes $n(\lg n + \lg|\Sigma|) + O(n)$ bits of space, for a given text of length n and supports search queries in $O(m)$ time.

Recently Raman et al. [RRR02] have obtained a static dictionary representation that uses $\mathcal{B}(n,m) + o(n) + O(\lg\lg m)$ bits of space and supports both *rank* and *select* operations on the given set in constant time, where $\mathcal{B}(n,m) = \lg\binom{m}{n} \approx n\lg(m/n) + n\lg e + O(\lg n)$ is the information theoretic lower bound on the amount of space required to store a subset of size n from a universe of size m. Using this, they have also obtained a cardinal tree representation that uses $\mathcal{C}(n,k) + o(n + \lg k)$ bits of space and supports 'parent', 'i^{th} child', 'child labeled j' and 'degree' operations in constant time, where $\mathcal{C}(n,k) = \left\lceil \lg\left(\frac{1}{kn+1}\binom{kn+1}{n}\right)\right\rceil \approx n(\lg k + \lg e)$ is the information theoretic lower bound on the space required to store a cardinal tree of degree k with n nodes. But this structure does not support the subtree size operation (in constant time). It is an interesting open problem to extend this or find a representation that uses comparable amount of space and also supports the subtree size operation.

It is also interesting to see if one can get a static dictionary structure (perhaps also supporting rank and select) that uses only $\mathcal{B}(n,m) + o(n)$ bits of space. (This bound has been achieved in the cell probe model [RRR02].) The lower bound due to Mehlhorn [Meh82] shows that in the word RAM model one cannot get such a structure using minimal perfect hashing based methods. One major open problem is to make the succinct dictionary structures dynamic, while keeping them succinct.

6.1.3 Static Dictionary in the Bitprobe Model

We have shown the construction of a scheme to store two element subsets of the universe $[m]$ using $3m^{2/3}$ bits of space which can be used to answer membership queries using two adaptive bit probes. This improves the $3m^{3/4}$ bit existential scheme given by Buhrman et al. [BMRV00]. We then generalized this to a scheme resulting in an adaptive scheme that answers queries using $\lg\lg n + 2$ probes using $o(m)$ bits of space, for a large range of n.

More recently, Radhakrishnan et al. [RSV00] have studied the quantum complexity of the static dictionary problem and have given several upper and lower bounds.

Pagh [Pag01b] has given a structure that requires $O(kn + m/2^k)$ bits of space and answers queries using $O(k)$ bit probes to the structure. For $k = \lg(m/n)$, it gives a scheme that takes $O(\mathcal{B}(n, m))$ bits of space and supports membership queries using optimal $O(\lg(m/n))$ bit probes to the structure. This matches the above lower bound for t when $s = O(\mathcal{B}(n, m))$. Note that, for $k = \lg \lg n$, this gives a scheme using $O(m/\lg n)$ bits of space that answers queries using $\lg \lg n$ probes. Though this scheme does not improve the scheme of Corollary 4.2.16, this scheme is better for sufficiently large values of n.

Using the ideas of Theorem 4.2.8, we can get, for example, a scheme with $s = O(t^2 m^{1/(t-1)})$, for any $t \geq 3$, for two element subsets of the universe (i.e., $n = 2$) and a scheme with $s = O(t^2 m^{1/(t-2)})$, for any $t \geq 5$, for three element sets of the universe. The real open problem here is to come up with an explicit scheme that uses $o(m)$ bits of space and answers queries using constant number of probes. Using probabilistic methods, Buhrman et al. [BMRV00] have shown the existence of such a scheme. We conjecture that, for example, there is no scheme using $o(m)$ bits that answers queries using three probes.

6.1.4 Dynamic Data Structures

We mainly focused on the partial sums problem and the dynamic array problem. For the partial sums problem, we gave a succinct structure that supports *sum* in $O(\log_b n)$ time and *update* in $O(b)$ time, for any parameter $b \geq \lg n/\lg \lg n$. We also gave a succinct structure that supports both these operations and the *select* operation, all in $O(\lg n/\lg \lg n)$ worst case time. As a special case, we considered the dynamic bit vector problem where, using $o(n)$ bits of extra space, we can also support the *select* operation in $O(\log_b n)$ time while *flip* can be supported in $O(b)$ amortized time, for any parameter $b \geq \lg n/\lg \lg n$.

For the dynamic array problem, we first gave a structure that uses $o(n)$ words of extra space and supports updates in $O(n^\epsilon)$ worst case time and query in $O(1)$ worst case time, for any fixed positive constant $\epsilon \leq 1$. Then, using this structure, we obtained a dynamic array structure that supports both query and update operations in optimal $O(\lg n/\lg \lg n)$ amortized time using $o(n)$ bits of extra space.

The following related problems remain open:

- In the searchable partial sums problem, we were able to support select in $O(\log_b n)$ time and update in $O(b)$ time using $kn + o(kn)$ bits of space, when either $k = 1$ (for any parameter $b \geq \lg n / \lg\lg n$) or $b = \lg n / \lg\lg n$. When is this trade-off achievable in general?

- For the partial sums problem, we have shown tradeoffs between the query and update times. Can we show similar tradeoffs between query and update operations for the dynamic array problem (both upper and lower bounds)? Also, in particular, is there a succinct structure where updates can be supported in $O(1)$ time and accesses in $O(n^{\epsilon})$ time?

- Another closely related problem looked at by Dietz [Die89], and Fredman and Saks [FS89] is the *list indexing* problem which is like the dynamic array problem, but adds the operation *position*(x), which gives the position of a given item x in the sequence, and also modifies *insert* to insert a new element after an existing one. Dietz has given a structure for this problem that takes $\Theta(n)$ extra words and supports all the operations in the optimal $O(\lg n / \lg\lg n)$ time. It is not clear how one can reduce the space requirement to just $o(n)$ extra words, and still support the operations in optimal time.

Publications

[MRR01] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[Rao02] S. Srinivasa Rao. Time space tradeoffs for compressed suffix arrays. *Information Processing Letters*, to appear.

[RR99] Venkatesh Raman and S. Srinivasa Rao. Static dictionaries supporting rank. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, volume 1741 of *Lecture Notes in Computer Science*, pages 18–26. Springer, December 1999.

[RRR01a] Jaikumar Radhakrishnan, Venkatesh Raman, and S. Srinivasa Rao. Explicit deterministic constructions for membership in the bitprobe model. In *Proceedings of the European Symposium on Algorithms (ESA)*, volume 2161 of *Lecture Notes in Computer Science*, pages 290–299. Springer, August 2001.

[DMRR01] Erik D. Demaine, J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Beating bitvectors with oblivious bitprobes. Technical report IMSc/2001/12/56, Institute of Mathematical Sciences, Chennai, India, 2001.

[RRR01b] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS)*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer, August 2001.

Bibliography

[Ajt88] Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988.

[AP85] Alberto Apostolico and Franco P. Preparata. Structural properties of the string statistics problem. *Journal of Computer and System Sciences*, 31(3):394–411, December 1985.

[BCD+99] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 1999.

[BDMR99] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 1999.

[Ben98] David Benoit. Compact tree representations. Master's thesis, University of Waterloo, 1998.

[BF99] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing*, pages 295–304, 1999.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[BMRV00] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the ACM Symposium on Theory of Computing*, pages 449–458, 2000.

[Car75] Alfonso F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[CC96] Livio Colussi and Alessia De Col. A time and space efficient data structure for string searching on large texts. *Information Processing Letters*, 58(5):217–222, June 1996.

[CHM$^+$86] B. Clift, David Haussler, Ross M. McConnell, Tom D. Schneider, and Gary D. Stormo. Sequence landscapes. *Nucleic Acids Research*, 4(1):141–158, 1986.

[Cla96] David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[CM96] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[Die89] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer, 1989.

[DL01] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 289–294, 2001.

[FG96] Paolo Ferragina and Roberto Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001.

[FNSS92] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *Journal of the ACM*, 39(4):764–782, April 1992.

[Fre83] Greg N. Fredrickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30(1):80–94, January 1983.

[FS89] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994.

[FWM84] Christopher W. Fraser, Alan L. Wendt, and Eugene W. Myers. Analysing and compressing assembly code. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 117–121, 1984.

[GBYS92] Gaston H. Gonnet, Ricardo Baeza-Yates, and T. Snider. New indicies for text: PAT trees and PAT arrays. In *Information Retrieval: Algorithms and Data Structures*, pages 66–82. Prentice Hall, 1992.

[GI99] Michel T. Goodrich and John G. Kloss II. Tiered vectors: Efficient dynamic array for JDSL. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1999.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

[Irv95] Rob Irving. Suffix binary search trees. Technical Report TR-1995-7, Department of Computing Science, University of Glasgow, April 1995.

[Jac89a] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[Jac89b] Guy Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, January 1989.

[Kär95] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proceedings of the Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204. Springer, July 1995.

[Kär99] Juha Kärkkäinen. *Repetition-Based Text Indexes.* PhD thesis, University of Helsinki, Finland, November 1999.

[KU96] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proceedings of the Second International Computing and Combinatorics Conference*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer, December 1996.

[LV89] Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, June 1989.

[Mäk00] Veli Mäkinen. Compact suffix array. In *Proceedings of Symposium on Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2000.

[Mäk01] Veli Mäkinen. Trade off between compression and search times in compact suffix array. In *Proceedings of Workshop on Algorithm Engineering and Experiments*, volume 2153 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2001.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[Meh82] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 170–175, 1982.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science, 1984.

[Mil94] Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 625–634, May 1994.

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[MNSW98] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, August 1998.

[Mor68] Donald R. Morrison. PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[MP69] Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. MIT Press, Cambridge, Massachusetts, 1969.

[MR97] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997; to appear in *SIAM Journal on Computing*.

[MRR98] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. In *Proceedings of the 18th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 186–196. Springer, 1998.

[MRS01] J. Ian Munro, Venkatesh Raman, and Adam Storm. Representing dynamic binary trees succinctly. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 529–536, 2001.

[MS80] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, October 1980.

[Mun96] J. Ian Munro. Tables. In *Proceedings of the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

[Mut97] S. Muthu Muthukrishnan. Randomization in stringology. In *Proceedings of the FST & TCS Pre-conference Workshop on Randomization*, pages 23–27, 1997.

[Pag01a] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[Pag01b] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 425–432, 2001.

[RPE81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, to appear, 2002.

[RSV00] Jaikumar Radhakrishnan, Pranab Sen, and S. Venkatesh. The quantum complexity of set membership. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 554–562, 2000.

[Sad00] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 1969 of *Lecture Notes in Computer Science*, pages 410–421. Springer, December 2000.

[SS90] Jeanette P. Schmidt and Alan Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.

[Wei73] Peter Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.