Radix Sorting & Searching

Stefan Nilsson
Department of Computer Science
Lund University
Abstract

Efficient sorting and searching are corner-stones in algorithm design. In computer science it has become a deep-rooted habit to use comparison-based methods to solve these problems. In this thesis we argue that radix sorting and searching algorithms are superior in many respects, both practical and theoretical. They are often remarkably simple and many, if not most, sorting and searching problems may be cast in a form suitable for such algorithms.

In particular, we show that string sorting can be reduced to integer sorting in optimal asymptotic time and we show that a unit-cost RAM with a word length of \( w \) bits can sort \( n \) word-sized integers in \( O(n \log \log n) \) time for arbitrary \( w \geq \log n \), a significantly improved upper bound for sorting. The algorithm is simple and runs surprisingly fast also in practice.

We introduce a new practical radix sorting algorithm, *Forward radixsort*, that combines the advantages of traditional LSD and MSD radixsort in a simple way. Adding a preprocessing step to Forward radixsort we obtain an algorithm with interesting theoretical properties. For example, if \( \frac{B}{n} = O(w) \), where \( B \) is the minimum number of bits that must be inspected to distinguish the strings, it is possible to sort \( n \) binary strings in \( O(n \log \frac{n}{n \log n}) \) time. The result can also be expressed in terms of the entropy \( H \) per bit of the input: \( n \) strings from a stationary ergodic process can be sorted in \( O(n \log \frac{n}{H}) \) time. We implement some of the new radix sorting algorithms in the C programming language and achieve sorting routines that run significantly faster than previously presented string sorting routines.

The other theme of the thesis is a new compression technique, *level compression*, for trie structures. A level-compressed trie, or *LC-trie*, is easy to implement and inherits the good properties of standard tries with respect
to neighbor and range searches, while the average depth is significantly decreased. The expected average depth of an element in an LC-trie is $\Theta(\log^* n)$ for uniformly distributed data and $\Theta(\log \log n)$ for data from a Bernoulli-type distribution.

We also study the problem of string searching using suffix trees. Combining the methods of path compression, level compression, and data compression we achieve an efficient, compact, and fast implementation of a suffix tree. Based on extensive simulations, we argue that this new data structure is useful in many practical situations. For example, it can be used to reduce the number of accesses to secondary memory when searching in very large sets of data.
Preface

When I started as a graduate student I thought I knew what was expected of me: I should read all the articles carefully and try to understand the concepts and details as fully as possible. Then I should try to produce similar, but better, results. I found the approach of my advisor to be altogether different. He rarely finished reading an article. Typically, he looked at the abstract and exclaimed that this must be the wrong approach, there must be a different solution or another more relevant problem. The fact that the authors were famous and the articles published in prestigious journals seemed to be of little consequence. Combining this attitude with persistence and knowledge of your subject you are bound to find paths that no one else has walked before. And, if you are lucky, there will be some exciting discoveries waiting for you. That is what I learned in school during the last few years. For this, and for getting me started, and for long (sometimes very long) and fruitful discussions, and for being a great friend I thank my advisor Ame Andersson. Throughout the work he has been a source of inspiration and I owe many of the ideas in this thesis to him.

The algorithm group in Lund has provided a pleasant and stimulating research environment. Special thanks to Jesper Larsson who carefully read the manuscript and suggested many improvements. I also would like to thank Paul Davidsson, Bengt Nilsson, Rolf Karlsson, and Kurt Swanson for their support. Thanks to you all!

Chapter 2 is based on two previously published articles. One is joint work with Ame Andersson, Rajeev Raman, and Torben Hagerup; the other is joint work with Ame Andersson.


Chapter 3 and Chapter 5 are based on three articles written jointly with Ame Andersson,

# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Sorting and Searching</td>
<td>1</td>
</tr>
<tr>
<td>1.2 A Note on Notation</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Model of Computation</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Distinguishing Prefixes and Tries</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Thesis Outline</td>
<td>9</td>
</tr>
<tr>
<td>2 Radix Sorting</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Previous Results</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Traditional Radix Sorting Algorithms</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Adaptive Radixsort</td>
<td>17</td>
</tr>
<tr>
<td>2.4 Forward Radixsort, Basic Version</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Forward Radixsort, Complete Version</td>
<td>22</td>
</tr>
<tr>
<td>2.6 Improved Bounds on Integer Sorting</td>
<td>26</td>
</tr>
<tr>
<td>2.7 Improved Bounds on String Sorting</td>
<td>31</td>
</tr>
<tr>
<td>2.8 Conclusions</td>
<td>34</td>
</tr>
<tr>
<td>3 Radix Searching</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Trie Structures, Background</td>
<td>38</td>
</tr>
<tr>
<td>3.2 LC-Tries</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Implementation</td>
<td>45</td>
</tr>
<tr>
<td>3.4 Analysis</td>
<td>51</td>
</tr>
<tr>
<td>3.5 Conclusions</td>
<td>58</td>
</tr>
</tbody>
</table>
4 Experiments on Sorting 60
4.1 String Sorting 60
4.2 Integer Sorting 67
4.3 Conclusions 69

5 Experiments on Searching 71
5.1 Searching in Main Memory 71
5.2 Searching in Secondary Memory 75
5.3 Conclusions 77

6 Implementation 79
6.1 Forward Radixsort 82
6.2 Adaptive Radixsort 88
6.3 Integer Sorting in Nearly Linear Time 93

Notations 104

References 105
1

Introduction

Sorting and searching are two of the fundamental problems of computer science. To arrange or maintain a sequence of elements according to some predefined linear order is both a practically important and theoretically interesting problem. This type of problem occurs in many different contexts and is an essential ingredient of many algorithms. The ubiquity of sorting problems may perhaps be explained by the fact that sorting is useful not only to simplify searching but also for collecting related objects and finding duplicates.

In this introductory chapter we define the sorting and searching problems, introduce some notation, specify our machine model, and present the basic concepts of strings, distinguishing prefixes, and tries. The final section contains a brief outline of the thesis.

1.1 Sorting and Searching

We first give a formal definition of the general sorting problem. Given a sequence of \( n \) elements \( x_1, x_2, \ldots, x_n \) taken from a set with a linear order, denoted by \( \leq \), find a rearrangement \( x'_1, x'_2, \ldots, x'_n \) of the sequence such that \( x'_1 \leq x'_2 \leq \cdots \leq x'_n \).

Often the elements to be sorted are records consisting of several fields and a linear order is defined on one of the fields, called the key. In this case the sorting problem is to arrange the records so that the values of the key fields form a nondecreasing sequence. It is often required that the relative order of records with equal keys is not disturbed. A sorting algorithm with this property is said to be stable.

Comparison-based sorting algorithms do not make any assumptions about the structure of their elements. The atomic operation of these al-
A well known lower bound states that a comparison-based sorting algorithm must perform \( \Omega(n \log n) \) comparisons to sort a sequence of \( n \) elements in the worst case and there are several well-known algorithms, such as Heap sort and Mergesort, that match this lower bound [50].

Alphabetic sorting refers to the special problem of arranging strings in alphabetical order. Let \( \Sigma = \{0, 1, \ldots, m - 1\} \) be an alphabet and consider finite strings \( x_1, x_2, \ldots, x_n \) over \( \Sigma \). The \( i \)th character of the string \( x \) will be denoted \( x[i] \). An alphabetic ordering is defined in the following way.

**Definition 1.1**: Let \( x \) and \( y \) be two strings of length \( k \) and \( l \), then \( x \) is smaller than \( y \) in the alphabetic ordering if there is an \( i, 0 \leq i \leq k \), such that \( x[j] = y[j] \) for \( 1 \leq j \leq i \), and either \( i = k < l \) or \( i < k, i < l \) and \( x[i + 1] < y[i + 1] \).

We will use the term **radix sorting** to denote sorting algorithms for alphabetic sorting that make use of the fact that the input keys are strings. The name radix sorting comes from the fact that the keys may also be viewed as numbers in a number system with radix \( m \).

Alphabetic sorting may at first seem to be a rather specialized sorting problem. However, many problems can be cast in this form. The basic data abstractions of most computers are **integers**, **floating-point numbers** (typically defined by the IEEE 754 standard[41]), and **character strings**. All of these are well suited for radix sorting.

Searching is a problem closely related to sorting. Given a set of elements \( \{x_1, x_2, \ldots, x_n\} \) with unique keys and a query key \( k \), find out if one of the elements has a key that equals \( k \) and, if so, find the contents of the other fields of this element.

It is natural to make a distinction between comparison-based searching methods that only use comparison between keys, and character-based methods that make use of the fact that the keys are represented as character strings. Among the character-based methods hashing [50] is probably the most commonly used. It is a randomized scheme with constant time expected search cost. However, traditional hashing schemes do not preserve the order of the elements and hence cannot support operations like neighbor search and range queries efficiently.

The **trie** [36] is a simple and natural data structure that overcomes this problem. The trie is so important to our discussion of both sorting and searching that we devote Section 1.4 to a more detailed presentation of this data structure.
1.2 A NOTE ON NOTATION

Let $f$ and $g$ be two non-negative real-valued functions on the set of natural numbers.

- We say that $f = O(g)$ if there is a positive constant $c$, such that $f(n) \leq c \cdot g(n)$ for all but finitely many values of $n$.
- We say that $f = \Omega(g)$ if there is a positive constant $c$, such that $f(n) \geq c \cdot g(n)$ for an infinite number of values of $n$.
- We say that $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.
- We say that $f = o(g)$ if $f = O(g)$ and $f \neq \Omega(g)$.

As is common in computer science we assume that all logarithms are to the base 2, if not otherwise indicated. For convenience we sometimes omit ceilings from numbers that strictly should be integers. For example, we might say “Let a character consist of $v = \log n$ bits” even when $n$ is not a power of 2.

We also make use of one nonstandard function, the iterated logarithm function $\log^*$ defined as follows.

$$\log^* n = \begin{cases} 1 + \log^* \log n, & \text{if } n > 1; \\ 0, & \text{otherwise.} \end{cases}$$

This function has an extremely slow growth rate. In fact, it is the inverse of the perhaps more well-known tower function.

1.3 MODEL OF COMPUTATION

Sequential computers are standardized to a large degree. Both hardware designers and software producers have a common conception about the architecture of a typical sequential machine. This vaguely defined virtual machine is often referred to as a von Neumann style computer. Similarly, in the area of algorithm design, the unit-cost random access machine (RAM) [55], which models an idealized von Neumann computer, is widely accepted and widely used.

To get a realistic model we assume that each machine word of the RAM consists of a finite number of bits $w$, which is referred to as the word length of the machine. Observe that there is a relationship between $w$ and the number of word sized elements $n$ that can be stored in the memory of
the machine, \( n \) cannot be larger than \( 2^w \). This is a very reasonable assumption, since otherwise a pointer is too large to fit in a machine word. The instruction repertoire is intended to reflect the operations found on a typical computer. It includes addition, subtraction, multiplication, comparison, bitwise logical operations, and shift operations. This model gives fairly good predictions for the time and space requirements of algorithms that do not access secondary storage.

One could of course argue that the RAM model is too coarse and does not account for such things as the different costs of basic instructions and hidden costs for memory allocation and indexing. But even so, an appropriately defined RAM model seems to predict the asymptotic behavior of algorithms on real computers with good accuracy, or to use a well known proverb: “If it ain’t broke don’t fix it.”

We study the time and space complexity of algorithms. In the RAM model the time complexity is defined as the number of basic operations performed when executing an algorithm. The space complexity of an algorithm is the amount of storage it uses in addition to the input. The worst-case complexity is the maximum complexity, taken over all inputs of a certain size, and the expected complexity is the “average” complexity over all inputs from some statistical distribution.

1.4 DISTINGUISHING PREFIXES AND TRIES

A principal observation is that to arrange a sequence of strings in alphabetic order it may not be necessary to inspect all characters. We will refer to the characters that must be inspected in order to verify that the strings are sorted as the distinguishing prefixes of the strings.

Definition 1.2: The distinguishing prefixes of a set \( x_1, \ldots, x_n \) of strings are the shortest prefixes of \( x_1, \ldots, x_n \) that are pairwise different. The distinguishing prefix of a string \( x_i \) that is a prefix of one of the other strings is defined to be the entire string \( x_i \).

An example is given in Figure 1-1.

For convenience we introduce the following notation for distinguishing prefixes. Denote the length of the distinguishing prefix of \( x_i \) by \( s_i \) and let \( S = \sum_{i=1}^n s_i \) and \( \bar{s} = S/n \). That is, \( S \) is the total number of characters that must be inspected in order to arrange the strings in alphabetic order and \( \bar{s} \) is the average length of a distinguishing prefix. Denote the length of the longest distinguishing prefix by \( S_{\text{max}} \).
Binary strings is a special case of particular importance and since we often need to simultaneously discuss both the number of distinguishing bits and the number of distinguishing characters in an alphabet of size $2^n$, it is convenient to introduce the special notation $B$ and $\bar{B}$ to denote the total and average number of distinguishing bits, respectively, while $S$ and $\bar{S}$ denote the number of distinguishing $v$-bit characters. Similarly, $B_{\text{max}}$ denotes the length of the longest binary distinguishing prefix.

We will express the time complexity of several of our algorithms in terms of $S$, the total number of distinguishing characters. This quantity has been thoroughly studied in the context of tries and here we give a quick review of some of the results that are relevant to our discussion. In its original form [36], the trie is a data structure where a set of strings from an alphabet containing $m$ symbols is stored in a natural way in an $m$-ary tree where each string corresponds to a unique path. This data structure is used in many different settings, including string matching, approximate string matching, compression schemes, and analysis of genetic sequences.
Definition 1.3: A trie containing \( n \) strings from an alphabet \( \Sigma \) of size \( m \) is a tree with the following properties.

- If \( n = 0 \), the trie is an empty tree.
- If \( n = 1 \), the trie consists of one node representing the element.
- If \( n > 1 \), the trie consists of a node of degree \( m \). For each \( c \in \Sigma \) there is a child, which is a trie, containing all strings starting with \( c \), with the first character removed.

The trie corresponding to the strings in Figure 1-1 is shown in Figure 1-2. Observe that this definition requires the strings to be prefix-free: no string can be a prefix of another string.

There is a direct relationship between distinguishing prefixes and tries. The length \( s_i \) of the distinguishing prefix of a string \( x_i \) equals the depth of \( x_i \) in the \( m \)-ary trie formed by the strings. In particular, \( S \) equals the average depth of a leaf in the trie. Similarly, the length \( S_{\text{max}} \) of the longest distinguishing prefix equals the height of the trie. Next, we give a quick review of some important statistical properties of tries.

Density Function

A frequently used statistical model is to consider independent elements with common distribution. Devroye [30, 31] has shown that for binary fractional expansions of independent random variables with common density \( f \in L^2 \) on \([0,1]\), that is, \( \int f^2(x) \, dx < \infty \), the expected value \( E(\beta) \) of the average length of a distinguishing prefix has the following asymptotic properties.

Theorem 1.4: For a density function \( f \in L^2 \)

\[
\lim_{n \to \infty} \frac{E(\beta)}{\log n} = 1
\]

and

\[
\lim_{n \to \infty} \frac{E(\beta_{\text{max}})}{\log n} = 2,
\]

This means that tries are asymptotically optimal for a density \( f \in L^2 \). On the other hand, Devroye also shows that if \( f \not\in L^2 \) the expected average depth \( E(\beta) = \infty \) for all \( n \geq 2 \).
1.4 Distinguishing Prefixes and Tries

Stationary Ergodic Process

Another frequently used model is to consider independent elements from a stochastic processes \( \{X_i\} \) with values from an alphabet \( \Sigma \). This is a very general model and to be able to obtain interesting results one often considers \textit{stationary ergodic processes}, a model that has a long tradition in information theory.

Informally, the ergodic property means that the process is statistically homogeneous. The stationary property means that it is time independent. The seminal work by Shannon [66] gives an introduction to the basics of information theory and ergodic \textit{Markov processes}. The more general case of stationary ergodic processes is treated by Billingsley [20]. An important quantity that can be defined for any stationary ergodic process is the \textit{entropy per character}.

\textbf{Definition 1.5}: The entropy \( H \) per character is defines as

\[
H = \lim_{n \to \infty} \frac{1}{n} \mathbb{E} \{ \log P[X_1 = x_1 \mid 1 \leq i \leq n, x_i \in \Sigma] \}.
\]

The Kolmogorov-Sinai Theorem [20] guarantees the existence of this limit. Informally, \( H \) can be viewed as a measure of disorder. If the strings have a high degree of disorder and hence high entropy, we can expect to be able to differentiate between two strings by only looking at a small number of characters. On the other hand, if the entropy is low the strings will have a large number of characters in common and hence it will take more computational power to tell them apart. This idea has been put into mathematical form by Pittel [60]. To state the theorem of Pittel we need to introduce a technical mixing condition.

\textbf{Condition 1.6}: Denote \( F^a \) the \( \sigma \)-field generated by \( X_a, \ldots, X_b, 1 \leq a \leq b \); there exist two positive constants \( c_1 \leq c_2 \) and an integer \( b_0 > 0 \) such that for all \( 1 \leq a \leq a + b_0 \leq b \),

\[
c_1 P(A \mid P(B) \leq P(AB) \leq c_2 P(A)P(B)
\]

whenever \( A \in F^a, B \in F^{a+b_0} \).

This is rather weak assumption that holds for a large class of processes.

\textbf{Theorem 1.7}: For independent strings from a stationary ergodic process obeying Condition 1.6

\[
\lim_{n \to \infty} \frac{\mathbb{E}[\mathcal{S}]}{\log n} = \frac{1}{H}.
\]
An interesting special case is a Bernoulli-type process, where each string \( x \) consists of independent identical random variables \( x(i) \) with values from an alphabet \( \Sigma \) such that \( \Pr(x(i) = s) = p(s) > 0, s \in \Sigma \). In this case the entropy per character \( H = -\sum_{s \in \Sigma} p(s) \log p(s) \) and we get the following corollary.

**Corollary 1.8:** For independent binary strings from a Bernoulli-type process with character probabilities \( p(s) > 0, s \in \Sigma \),

\[
\lim_{n \to \infty} \frac{E[S]}{\log n} = \left( \sum_{s} p(s) \log \frac{1}{p(s)} \right)^{-1}.
\]

**Path-Compressed Tries**

In many applications a standard implementation of a trie can be prohibitively large. A standard technique to counter this problem is to use path compression. All internal nodes with only one nonempty child is removed from the trie. The path-compressed version of the trie in Figure 1-2 is shown in Figure 1-3. A path-compressed binary trie is often called a Patricia tree [42].

The size of a path-compressed trie does not depend on the length of the strings, but only on the total number of strings. A path-compressed trie storing \( n \) strings has at most \( 2n - 1 \) nodes. The Patricia tree, a binary path-compressed trie, has exactly \( 2n - 1 \) nodes. The statistical properties of this trie structure are very well understood [33, 63, 68]. For a large class of distributions path compression does not give an asymptotic improvement over a plain trie. Even so, path compression is very important in practice, since it often gives a significant overall size reduction.

![Figure 1-3. A path-compressed trie.](image-url)
1.5 **Thesis Outline**

In this thesis we study two of the fundamental problems of computer science: sorting and searching. In an attempt to find improved algorithms and data structures for these basic problems we make use of the fact that data is often represented as binary strings. One of the prime ideas of the thesis is to try to reduce the time and space complexity by processing several consecutive bits at once.

In Chapter 2 we give a review of some well known radix sorting algorithms and show that it is possible to improve these classic algorithms with simple means. We introduce a new radix sorting method, *Forward radixsort*, that combines the advantages of LSD and MSD radixsort. Adding a simple preprocessing step to this algorithm we show how a string sorting problem can be reduced to an integer sorting problem in optimal asymptotic time. With this reduction as a starting point we achieve several new and improved time bounds for integer sorting and string sorting. In particular, we show that integer sorting can be performed in $\Theta(n \log \log n)$ time on a unit-cost RAM with arbitrary word length. Combining Forward radixsort with different efficient integer sorting algorithms we obtain several new and improved time bounds for the string sorting problem.

In Chapter 3 we introduce a new compression method, *level compression*, for tries. The idea is to use as large a radix as possible in each internal node without introducing any new empty leaves. It turns out that this can be achieved quite easily. A level-compressed trie, or L.C-trie, has nice statistical properties: the average expected depth is $\Theta(\log^* n)$ for uniformly distributed data and $\Theta(\log \log n)$ for data from a Bernoulli-type process.

In Chapter 4 we present experimental results on sorting. Time measurements on different computers show that our new radix sorting algorithms are very efficient. Carefully implemented radix sorting algorithms often run several times faster than the best comparison-based algorithms. We also discuss how different optimization schemes, old and new, affect the performance of the sorting algorithms. Finally, we show that the new $\Theta(n \log \log n)$ worst-case time integer sorting algorithm is surprisingly fast also in practice.

Chapter 5 contains simulations of different trie structures. The measurements show that the nice theoretical properties of L.C-tries are reflected also in practice. In particular, we simulate searching in secondary memory and show that it is possible to build an efficient index by combining path compression and level compression. The index is small, only needs to inspect
a small number of external records during a search, and has a guaranteed worst-case behavior.

In Chapter 6 we present the implementation details of the new radix sorting algorithms that were used in the experimental work. We also give a complete implementation of the $\Theta(n \log \log n)$ worst-case time integer sorting algorithm from Chapter 2. The code is intended to illustrate the algorithm and to prove that it is, in fact, possible to surpass the comparison-based $\Omega(n \log n)$ lower bound with a relatively simple program.
RADIX SORTING IS A SIMPLE and very efficient sorting method that outperforms the more well known comparison-based algorithms in many respects. Even so, radix sorting has traditionally received little attention. A common misconception is that radix sorting algorithms either have to inspect all the characters of the input or use an inordinate amount of extra space and time. In this chapter we show that this is not the case and that radix sorting can be very efficient both in theory and practice.

The chapter starts with a short summary of the previously best known results on string sorting and integer sorting. The next section contains a review of some well known radix sorting algorithms: Bucket sort, LSD radixsort, and MSD radixsort. We discuss how these algorithms can be efficiently implemented and also propose some new optimization techniques.

Next, we discuss how distributive partitioning [34] can be applied to radix sorting. This yields a simple extension of MSD radixsort where the size of alphabet adapts to the number of elements remaining to be sorted. This algorithm has good expected time complexity and performs very well in practice. The worst-case complexity, however, is the same as for standard MSD radixsort.

In Sections 2.4 and 2.5 we introduce a new sorting algorithm, Forward radixsort, that combines the advantages of LSD and MSD radixsort. The algorithm is simple, very efficient, and has good worst-case behavior. Adding a preprocessing step to the basic version of Forward radixsort we obtain an algorithm with interesting theoretical properties. In particular, we show that any string sorting problem can be reduced to an integer sorting problem in optimal asymptotic time.

In light of this it is natural to take a closer look at the integer sorting problem. The main new result is achieved by combining Forward radix-
sort with a packed sorting scheme yielding a new integer sorting algorithm that sorts $n$ word-sized integers in $O(n \log \log n)$ worst-case time on a RAM with arbitrary word length, a significantly improved upper bound for sorting.

Finally, by combining Forward radixsort with different efficient integer sorting algorithms we obtain several new and improved time bounds for string sorting. For example, we show that $n$ binary strings from a stationary ergodic process with entropy $H$ per bit can be sorted in $O(n \log \frac{1}{H})$ time if the average length of a distinguishing prefix fits within a constant number of machine words.

2.1 PREVIOUS RESULTS

There is a plethora of radix sorting algorithms. We will discuss some of them in more detail later on. In this section we merely give a survey of the previously best known upper bounds for radix sorting.

In 1984 Kirkpatrick and Reisch [49] presented a radix sorting algorithm that sorts $n$ word-sized integers on unit-cost RAM with word length $w$ in $\Omega(n \log \frac{w}{\log n})$ worst-case time. The algorithm uses a recursive bucketing technique similar to the one used in the van Emde Boas tree [76-78] from 1975. The first integer sorting algorithm to run in $O(n \log n)$ worst-case time on a RAM with arbitrary word length was invented in 1990 by Fredman and Willard [38]. The rather complicated algorithm, which is based on so called fusion trees, runs in $\Omega(n \sqrt{\log n})$ time.

In 1987 Paige and Tarjan [57] presented a new radix sorting algorithm that sorts $n$ binary strings in $\Omega(B/\log n + n)$ time, where $B$ is the total number of distinguishing bits. Chen and Reif [24] presented an algorithm for binary strings from a stationary ergodic process in 1993. Their algorithm sorts $n$ such strings in $\Omega(n \log \frac{\log n}{H})$ time, where $H$ is the entropy per bit of the process. The algorithm assumes that string comparison can be performed in constant time and hence the result holds only if each string fits within a constant number of machine words.

The algorithm by Paige and Tarjan can be related to the algorithm by Chen and Reif through Theorem 1.7. The theorem, which was proved already in 1985 by Pittel [60], exhibits a close relationship between entropy and distinguishing prefixes and implies that any time complexity expressed in terms of $B$ can also be expressed in terms of $H$. For example, the algorithm by Paige and Tarjan runs in $\Omega(n/H)$ time for input from a stationary ergodic process.
2.2 TRADITIONAL RADIX SORTING ALGORITHMS

Bucketsort

The idea to sort numbers by distributing them into buckets can be found already in one of the very first articles on computer sorting [40]. Bucketsort is still the basic building block of most radix sorting schemes and therefore we devote some space to present this simple algorithm.

Consider the special case of sorting \( n \) integers drawn from a small set \( \{0, 1, \ldots, m - 1\} \). Bucketsort maintains one bucket for each possible key value, puts the elements in their respective buckets, and collects the elements during a traversal of the buckets.

**Implementation** Perhaps the most obvious implementation is to use an array of linked lists to represent the \( m \) buckets. By maintaining pointers to both the head and tail of each list we can insert the elements at the end of the list in constant time and in this way we also assure that the sorting algorithm will be stable.

Another possibility is to use an array of counters, where each counter indicates the number of elements in each bucket. Using this information we can easily compute the position of the first element in each bucket. With this information available it is easy to permute the \( n \) elements into sorted order in \( O(n) \) time. One drawback of this scheme is that each character has to be read twice, one time during the counting and one time during the permutation.

The simplest way to permute the elements is to move them into a new array. It is also possible to perform the permutation in place. One way to do this is illustrated in Figure 2-1. If the first element does not equal 0 we move it into its correct position and remove the element that resides in this new position; if this element equals 0 we are done, otherwise we continue to displace elements until we eventually find a 0 element, which can be placed in the first slot. This procedure is repeated for the next element that has not yet found its correct place. To perform this sequence of dis-

```
1 0 2 2 0 1 0

0 0 2 1 1 2 0
```

**Figure 2-1.** Permuting an array in place.
placements we need to keep track of the current point of insertion for each bucket. This can easily be arranged using a pointer array of size $m$. The initial values of this array are computed as the prefix sums of the counter array. Observe that this in place permutation scheme produces a sorting algorithm that is not stable.

**Optimizations** A certain amount of clustering is often present in many kinds of data. One way to take advantage of this fact is to treat consecutive elements with a common value as a single unit when moving them into a bucket. Using a linked-list representation it is possible to move this sublist of identical elements in constant time. This simple optimization improves the performance in many practical situations as will be shown in Chapter 4.

**Analysis** Bucketsort inspects each element a constant number of times and visits each bucket once, hence it runs in $O(n + m)$ time. The linked list implementation uses $O(n + m)$ extra space: $O(n)$ space is used to implement the linked list and $O(m)$ is needed for the buckets. Using a counter array it is possible to implement Bucketsort using only $O(m)$ extra space. However, the in place implementation is unstable.

**LSD Radixsort**

Bucketsort is a feasible alternative only if the elements to be sorted take their values from a restricted domain. For long strings it is natural to sort the keys in several stages, using Bucketsort at each stage. The most commonly used approach is to start with the least significant digit (LSD). This method is old and was used to sort punched cards. The algorithm has been referred to as LSD radixsort [48, 50], Straight radixsort [65], or Bottom-up radixsort [42].

The algorithm, in its basic form, requires all strings to be of equal length. It works as follows.

- Split the strings into groups according to their last character and arrange the groups in ascending order.

- Apply the algorithm recursively on all strings, disregarding the last character of each string.

After the $i$th step of the algorithm, the input strings will be properly sorted according to their last $i$ characters.
The algorithm can be extended to handle strings of different lengths. One such algorithm is presented by Aho, Hopcroft, and Ullman [1]. The idea is to traverse all strings during a preprocessing step and collect information about the length of each string. We do not pursue this approach further, since all LSD radix sorting algorithms suffer from the problem that they have to inspect all characters of the input and we can expect to do better using other techniques.

**Implementation** The sorting subroutine is typically implemented using Bucketsort. Other sorting subroutines may of course also be used, as long as they are stable.

An important observation is that to implement a string sorting algorithm efficiently we should not move the strings themselves but only pointers to them. In this way, each string movement is guaranteed to take constant time. This is not a major restriction. In fact, it is common to store the characters of each string in consecutive memory locations and represent each string by a pointer to the first character of the string. The length of the string can be stored explicitly or the end of the string can be marked by a specially designated end-of-string character.

**Optimizations** When choosing the size of the alphabet we are faced with a fundamental problem: a large alphabet reduces the total number of passes but increases the total number of buckets that must be inspected. In practice one can sometimes use the structure of the data to choose a larger alphabet than would otherwise be feasible. In our implementation of radix sorting algorithms for ASCII-encoded text we use a new simple heuristic. Two consecutive characters are used for bucketing, yielding an alphabet of size 65536. The new idea is to keep track of the characters that occur in the first and second position. For example, assuming that $n_1$ and $n_2$ different characters have been found in the first and second position, respectively, we only need to inspect $n_1n_2$ buckets. This number will typically be much smaller than the total 65536 buckets. Further details can be found in Chapter 6.

**Analysis** The running time of LSD radixsort is a function of the number of strings $n$, the number of buckets $m$, and the length $l$ of the strings. During the bucketing phase each of the $n$ characters is inspected once and during the rebuilding each of the $m$ buckets is visited once. This procedure is repeated $l$ times and hence the running time is $\Theta(l(n+m))$. 
MSD Radixsort

The major weakness of LSD radixsort is that it needs to inspect all characters of the input. Another, perhaps more natural, alternative is to scan the strings starting with the most significant digit (MSD) and only inspect the distinguishing prefixes. This algorithm is known as MSD radix sort [48, 50] or Top-down radix sort [42]. The special case where the alphabet is binary is often referred to as Radix-exchange sort [44, 50]. The algorithm works as follows.

- Split the strings into groups according to their first character and arrange the groups in ascending order.

- Apply the algorithm recursively on each group separately, disregarding the first character of each string. Groups containing only one string need no further processing.

After the ith step of the algorithm, the input strings will be properly sorted according to their first i characters.

Implementation Once again, Bucket sort is typically used to implement the sorting at each step. In this case we can use the in-place version of Bucket sort, but then the resulting algorithm will of course be unstable.

We want to avoid using a new bucket table for each splitting. This can be arranged if we use an explicit stack to keep track of the flow of computation. The last bucket is treated first and the other sublists to be sorted are pushed on the stack. This is illustrated in Figure 2-2. The sublists on the stack that are waiting to be sorted are suggested by the dotted line.

Optimizations One problem to overcome when implementing MSD radix sort is that the groups may become small while the cost of Bucket sort remains proportional to the size of the alphabet. This fragmentation problem can be countered by swapping to another sorting algorithm when only a

![Figure 2-2](image-url)
small number of elements remain in a group. This is classic optimization method [51] that has been widely used.

Another possible drawback is that the stack can be sizable. It is not difficult to construct an example where the stack will contain one entry for each key. We suggest a new simple way to overcome this problem. If both the list on the top of the stack and the list in turn to be pushed onto the stack are sorted there is no need to allocate another stack record. We can simply append one list to the end of the other. This technique is applicable if the algorithm switches to a simple comparison-based algorithm to sort short subsequences. If we choose to switch when at most \( k \) elements remain in a group, each stack entry will contain at least \( k \) elements and hence the total size of the stack will be at most \( n/k \). In practice, the stack will typically be much smaller as will be shown in Chapter 4. Not only does this simple optimization give a considerable reduction of the stack size, it is also likely to improve the running time.

**Analysis** The algorithm inspects each distinguishing character once and each string at least once. Hence the algorithm has time complexity at least \( \Omega(n + S) \), where \( S \) is the total number of characters of the distinguishing prefixes. To compute the number of buckets that are visited we observe that bucketing takes place at each internal node of the corresponding trie. The number of nodes in the trie could be as large as \( \Theta(S) \) and hence the worst-case time complexity is \( \Theta(n + mS) \).

In practice, when using the optimization techniques discussed above, the performance is often much better than could be expected from this worst-case time bound. In particular, the technique to revert to a simpler sorting algorithm when only a small constant number of elements remain is often very effective, even though it does not improve on the asymptotic bound.

### 2.3 ADAPTIVE RADIXSORT

In this section we adapt an old idea, which has been used primarily for sorting real numbers, to radix sorting. The resulting algorithm is extremely simple, has a good expected running time for a large class of distributions, and works very well in practice.

In an article from 1978 Dobosiewicz [34] presents a sorting algorithm based on *distributive partitioning*. The algorithm sorts \( n \) real numbers by distributing them into \( n \) intervals of equal width. The process is repeated
Chapter 2: RADIX SORTING

recursively for each interval that contains more than 1 element; the numbers are distributed into a number of intervals that equals the number of keys. By combining this approach with a standard Quicksort partitioning step Dobosiewicz gets a sorting algorithm with $O(n)$ expected time for uniform data and $O(n \log n)$ worst-case time.

Ehrlich [35] also presents several searching and sorting algorithms based on the idea of distributive partitioning. In particular, he defines a new data structure: the N-tree. At the root the N-tree partitions the key-space into $n$ intervals of equal width, where $n$ is the number of keys; the same technique is applied recursively on the children.

The approach of distributive partitioning can of course also be applied to radix sorting algorithms. A natural extension of MSD radixsort is to choose the size of the alphabet adaptively: the alphabet size is chosen as a function of the number of elements remaining. For example, for binary strings a natural choice is to use characters consisting of $\log k$ bits to distribute the elements of a group of size $k$ into $\Theta(k)$ buckets. In this way the number of buckets that need to be traversed will be proportional to the number of inspected characters.

Observe that the total number of bits read by the algorithm might be larger than for plain MSD radixsort, since we might read a number of bits beyond the distinguishing prefix. However, this happens at most once for each string and hence will account for at most a linear extra cost. In practice this extra cost is typically overshadowed by the gain from not having to inspect as many superfluous empty buckets.

Analysis

Using this scheme the cost of visiting buckets matches the cost of inspecting bit patterns from the strings. However, in a worst-case scenario the elements will be split into many small groups already after a few steps and the algorithm will read only a small constant number of bits at a time. Hence, the worst-case time complexity is $\Theta(n + B)$, where $B$ is the total number of bits of the distinguishing characters.

Observe that if we decide never to inspect less than $\log m$ bits, where $m$ is the size of the underlying alphabet, the time complexity is guaranteed to be at least as good as for standard MSD radix sort.

The expected cost of distributive partitioning methods is well understood. There is a close relationship between N-trees and Adaptive radixsort. The algorithm performs at most $O(n)$ work at each level of the cor-
responding N-tree. Tamminen [70] has shown that the expected average depth of an N-tree is at most 1.8 for uniformly distributed data and hence adaptive radix sort runs in linear expected time. In fact, the expected average depth is less than 4 for a large class of density functions f. In particular, this result holds if f is Riemann-integrable, bounded, and has compact support. A more detailed average case analysis of bucket algorithms is given by Devroye [32] and Tamminen [71].

2.4 FORWARD RADIXSORT, BASIC VERSION

Several authors [26, 48, 55] have pointed out that MSD radix sort has a bad worst-case performance due to fragmentation of the data into many small sublists. In this section we present a new simple MSD radix sort algorithm, Forward radix sort, that overcomes this problem. The algorithm combines the advantages of LSD and MSD radix sort. The main strength of LSD radix sort is that it inspects a complete horizontal strip at a time; the main weakness is that it inspects all characters of the input. MSD radix sort only inspects the distinguishing prefixes of the strings, but it does not make efficient use of the buckets. Forward radix sort starts with the most significant digit, performs bucketing only once for each horizontal strip, and inspects only the significant characters.

The algorithm maintains the invariant that after the ith pass, the strings are sorted according to the first i characters. The sorting is performed by separating the strings into groups. Initially, all strings are contained in the same group, denoted group 1. This group will be split into smaller groups and after the ith pass all strings with the same first i characters will belong to the same group. The groups are kept in sorted order according to the prefixes seen so far. Each group is associated with a number that indicates the rank in the sorted set of the smallest string in the group. We also distinguish between finished and unfinished groups. A group will be finished in the ith pass if it contains only one string or if all the strings in the group are equal and not longer than i. The ith pass of the algorithm is performed in the following way:

- Traverse the groups that are not yet finished in ascending order and insert each string x, tagged by its current group number, into bucket number x(i) (recall that x(i) is the ith character of x).
- Traverse the buckets in ascending order and put the strings back into their respective groups in the order as they occur within the buckets.
Traverse the groups separately. If the kth string in group g differs from its predecessor in the lth character, split the group at this string. The new group is numbered g + k − 1.

Furthermore, the algorithm checks, in each pass, if all inspected characters are equal. If this is the case, no bucketing is performed.

Implementation
The buckets are implemented by an array of m linked lists. To keep track of the groups we also use a linked list, where each entry contains a list of elements with common prefixes. Each group also has a pointer that indicates the start of the next unfinished group. Using this data structure we can split a group in constant time. The unfinished groups can be traversed in time proportional to the total number of strings in these groups. Figures 2-3 and 2-5 give a snapshot of the group list before and after the third pass.

Another important observation is that it is not necessary to use an explicit tag for each element. Instead each bucket contains lists of elements with a common tag stored in the head of the list. Figure 2-4 illustrates this. In combination with the usual strategy of switching to a simple comparison-based sorting algorithm for small groups this gives a considerable space reduction and it also decreases the running time slightly.

Analysis
Forward radixsort runs in O(S + n + m·Smax) time, where Smax is the length of the longest distinguishing prefix. The first two terms come from the fact that the algorithm inspects each distinguishing character once and each string at least once. The last term comes from the fact that the algorithm runs in Smax passes and visits m buckets in each pass.

The worst-case running time is also bounded by O(S + n + m²). To see this, recall that the algorithm does not perform any bucketing in a pass where all the inspected characters are equal. We study two cases. First, when the number of strings remaining in the unfinished groups is larger than m the cost of bucketing is no larger than the cost of reading the characters. Second, when less than m strings remain there can be at most m passes in which any splitting of groups occurs and hence the total number of buckets visited is O(m²) in this case. In summary, we have the following theorem.
Figure 2-3. Snapshot of the group list after two passes.

Figure 2-4. Snapshot of the buckets after the insertion of all remaining strings with prefix “SO.”

Figure 2-5. Snapshot of the group list after three passes.
**Theorem 2.1:** A sequence of \( n \) strings from an alphabet of size \( m \) can be sorted in
\[
O(S + n + m \cdot \min\{m, S_{\text{max}}\})
\]
time, where \( S \) is the total number of distinguishing characters and \( S_{\text{max}} \) is the length of the longest distinguishing prefix.

As an application, consider the following method for sorting \( n \) binary strings. We regard the strings as consisting of characters with \( 1/2 \cdot \log n \) bits. Then the size of the alphabet \( m = \Theta(\sqrt{n}) \). Furthermore, the number of distinguishing characters \( S = \Theta(B/\log n + n) \) and from Theorem 2.1 we immediately get the following corollary.

**Corollary 2.2:** A sequence of \( n \) strings from a binary alphabet can be sorted in \( O(B/\log n + n) \) time, where \( B \) is the total number of distinguishing bits.

In particular, Theorem 1.7 implies that the running time is \( O(n/H) \) for input from a stationary ergodic process with entropy \( H \) per bit.

**Discussion**

The main virtue of Forward radixsort is its simplicity. The time bound of Corollary 2.2 was also derived by Paige and Tarjan [57] using a more complicated algorithm. Their algorithm first constructs an explicit unordered path-compressed trie structure. During this construction the edges of the trie are put into buckets according to the characters associated with the edges. During the second phase of the algorithm these buckets are traversed and the outgoing edges of each internal node of the trie are arranged in ascending order. The sorted list is produced during a pre-order traversal of the trie. This algorithm can be implemented to run in \( O(S + n + m) \) time.

In the next section we present a different algorithm that improves on this time bound. The main differences between the algorithms are that we avoid the construction of an explicit trie and that we separate the sorting of the edges from the rest of the algorithm. In this way we will be free to use a more efficient sorting scheme than bucketing as subroutine.

### 2.5 FORWARD RADIXSORT, COMPLETE VERSION

In the basic version of Forward radixsort we must visit all buckets, even the empty ones, in each pass. This can be avoided by adding a preprocessing
step. During the preprocessing we create a list $P$ of pairs. A pair $(i, c)$ indicates that character $c$ will take part in the splitting of a group in pass $i$.

Using this idea we get an extended algorithm that consists of three steps. First we create $P$, then we sort $P$, and finally we run the basic algorithm using the information in $P$ to avoid looking at empty buckets.

**Step 1: Create the list of pairs** The strings are processed from left to right in passes dividing them into groups. This time, however, the groups will not occur in sorted order. We only maintain a weaker invariant: after the $i$th pass the strings in an unfinished group will have the first $i$ characters in common. In detail, the following actions are performed in the $i$th pass.

- Traverse the unfinished groups and insert each remaining string $x$, tagged by its current group number, into bucket number $x|i$. Also, maintain a list of nonempty buckets; when a string is added to an empty bucket, the bucket is added to this list.

- Traverse the list of nonempty buckets and put the strings back into their respective groups in the order as they occur within the buckets.

- Traverse the groups separately. If the $k$th string in group $g$ differs from its predecessor in the $i$th character, split the group at this string.

  The new group is numbered $g + k - 1$. Also, every character $c$ that participates in the splitting of a group is added to $P$ as a pair $(i, c)$.

Observe that we store a pair $(i, c)$ only if the character $c$ is used to split a group in pass $i$. As a result $P$ contains at most $2n - 2$ elements. To see this, observe that the number of pairs can be no larger than the number of edges in the corresponding path-compressed trie and this number is at most $2n - 2$.

**Step 2: Sort the list of pairs** We observe that the pairs are already sorted according to their first coordinate, the pass number, since the elements were collected in this order. To sort the pairs we therefore only need to sort according to the second coordinate and then collect the strings into groups as indicated by the first coordinate.

This simple approach will always work but will be inefficient if the the number of passes is large, since then the first coordinate attains large values. However, it is always possible to collect efficiently in the following way. Instead of using the number $i$ of the pass we assign consecutive numbers $i'$ to the pairs in such a way that pairs with the same first coordinate
are assigned the same number.

\[
\begin{align*}
  i'_j & = i_j; \\
  i'_k & = \begin{cases} 
    i_{k-1}', & \text{if } i_k = i_{k-1}; \\
    i_{k-1}' + 1, & \text{otherwise.}
  \end{cases}
\end{align*}
\]

Using these numbers, collection can be done in linear time and space, since each \( i' \) is less than \( n \).

**Step 3: Run the basic algorithm** In the final step of the algorithm we run the basic version of Forward radixsort. Using the information in \( P \) we do not have to inspect any empty buckets and hence the running time of this step will be \( \Theta(S + n) \). As before the algorithm performs no bucketing if all inspected characters of a group are equal.

**Reducing the Size of the Integer Sorting Subproblem**

Forward radixsort reduces the problem of sorting \( n \) strings to a problem of sorting at most \( 2n - 2 \) integers. The constant factor 2 can be reduced to 1 by a simple modification of the algorithm. This may seem as a minor improvement, but later on we will apply Forward radixsort several times in a recursive fashion and then the size of the constant becomes crucial.

Using an idea by Kirkpatrick and Reisch [49], we modify Step 1 and Step 3 of the algorithm. For each group we find the set of minimum elements. These elements will occupy the first positions in the sorted group and hence we do not need to process them any further. In particular, there is no need to move them into a bucket. In this way the total number of pairs generated in Step 1 is less than \( n \), since the number of children of each internal node of the path-compressed trie is reduced by one. The same modification is performed in Step 3 and hence the information about what buckets to use will be relevant.

**Analysis**

The first step of the algorithm runs in \( O(S + n) \) time, since we do not visit any empty buckets. Note that his time bound does not allow us to initialize the bucket array. This is, however, not necessary. The standard trick [1, Problem 2.12] is to maintain a pointer in each active bucket that points to a back pointer on a stack. Each time a bucket is accessed, we verify that the contents are not random by making sure that the pointer in that bucket points to an active region of the stack and that the back pointer points to the bucket.
The time complexity of Step 2 is $O(n)$ plus the time to sort $n$ integers from the set \{0, 1, \ldots, m-1\}.

In Step 3 we run the basic version of the algorithm with the difference that we do not inspect any empty buckets and hence the time complexity is $O(S + n)$. In summary, we have the following theorem.

**Theorem 2.3:** A sequence of $n$ strings from an alphabet of size $m$, $m \leq 2^w$, can be sorted on a unit-cost RAM with word length $w$ in

$$O(S + n) + T(n, m, w)$$

time, where $S$ is the total number of significant characters and $T(n, m, w)$ is the time to sort $n$ integers from the set \{0, 1, \ldots, m-1\}.

In particular, for binary strings, treating $v$ bits as a character, we have an alphabet of size $2^v$ and the number of significant characters $S = \Theta(B/v)$. Assuming that $v = O(w)$, so that operations can be performed in constant time on bit patterns of length $v$, we get the following corollary.

**Corollary 2.4:** For any positive integer $v \leq w$, the problem of sorting a sequence of $n$ binary strings on a unit-cost RAM with word length $w$ can be reduced in $O(B/v + n)$ time, where $B$ is the total number of distinguishing bits, to a problem of sorting $n$ integers from the set \{0, 1, \ldots, 2^v - 1\}.

The amount of extra space required is $\Theta(n + 2^v)$. This can be reduced to $\Theta(n)$ using universal hashing [23, 26] to reduce the size of the bucket table, yielding a randomized algorithm with the same expected asymptotic time complexity.

There is a trivial lower bound for string sorting. Since the entire distinguishing prefixes must be read at least once and each element must be processed at least once we make the following observation.

**Observation 2.5:** Arranging $n$ binary strings in alphabetic order on a unit-cost RAM requires $\Omega(B/w + n)$ time in the worst case, where $B$ is the total number of distinguishing bits.

Choosing $v = w$ in Corollary 2.4, we see that the time complexity of the reduction equals the lower bound.

**Corollary 2.6:** The problem of sorting $n$ binary strings can be reduced in optimal asymptotic time to a problem of sorting $n$ word-sized integers.

To achieve any further improvements on the time complexity of string sorting we must turn to the integer sorting problem.
Discussion

The idea to reduce the constant by treating the minimum elements of each group separately goes back to Kirkpatrick and Reisch [49]. In fact, if we apply Forward radixsort to b-bit integers and let the characters consist of b/2 bits we get an algorithm that is virtually identical to the range reduction of Kirkpatrick and Reisch. The equivalent data structure, the *van Emde Boas tree* [76–78], is older. It is also recursive and it uses a similar range reduction technique.

2.6 IMPROVED BOUNDS ON INTEGER SORTING

Sorting \( n \) integers, or binary strings, from the set \( \{0, 1, \ldots, 2^w - 1\} \) on a unit-cost RAM is a special case of particular importance. For comparison-based algorithms there is a well known worst-case lower bound of \( \Omega(n \log n) \). For certain values of \( n \) and \( w \) this bound can of course be improved using the radix sorting algorithms discussed above. For example, if we use LSD radixsort with characters consisting of \( \log n \) bits we get the time bound \( \Theta(wn / \log n) \). This is an improvement over comparison-based algorithms if \( w = o(\log^2 n) \).

In the previous section we discussed how Forward radixsort can be used to reduce a string sorting problem to an integer sorting problem. But there is no reason to stop there. Forward radixsort can also be used to reduce the problem of sorting integers in a certain range to that of sorting integers in a smaller range. If we apply Corollary 2.4 to a set of b-bit integers and use an alphabet consisting of \( v = b/k \) bits we get the following lemma.

**Lemma 2.7:** In \( O(\log n) \) time, using \( 2^{b/k} \) extra space, the problem of sorting \( n \) integers from the set \( \{0, 1, \ldots, 2^b - 1\} \) can be reduced to a problem of sorting at most \( n \) integers from the set \( \{0, 1, \ldots, 2^{b/k} - 1\} \).

**Bucketing as Subroutine**

If we apply Lemma 2.7 \( \log(w / \log n) \) times (with \( k = 2 \)) and then sort the reduced numbers, which now consist of \( \log n \) bits, with Bucketsort we get an algorithm that sorts word-sized integers in

\[
O\left(n \log \frac{w}{\log n}\right).
\]
time. This algorithm is virtually identical to the one presented by Kirkpatrick and Reisch [49].

The algorithm uses $2^{w/2}$ extra space for the first reduction. As before, we can perform a reduction in linear space if we implement the bucket table using universal hashing [23, 26]. In fact, it is possible to implement the complete algorithm in linear space. The size of the elements are halved at each reduction and therefore the total extra space for the reduced subproblems is linear. Furthermore, the $O(n)$ pointers that are used to implement the linked list at each level of the recursion are shorter than the reduced numbers, since we switch to Bucketsort when a number consists of less than $\log n$ bits. Hence, the overall space complexity is linear.

Using a smaller alphabet in Forward radixsort we can reduce the space requirements without introducing randomization. From Lemma 2.7 we know that if we divide the numbers into $k$ parts we can perform a reduction in $O(kn)$ time and $O(2^{w/k})$ space. The reduced sorting problem consists of at most $n$ integers from the set $\{0, 1, \ldots, 2^{w/k} - 1\}$. This reduced problem can be solved in $O(n \log (w/k \log n))$ time and $O(2^{w/k})$ space using the algorithm discussed above. In summary, we have the following theorem.

**Theorem 2.8:** For any integer $k \geq 2$, a sequence of $n$ integers, each consisting of $w$ bits, can be sorted in

$$O \left( nk + n \log \frac{w}{k \log n} \right)$$

(time, using $2^{w/k}$ extra space.

In particular, choosing $k = \log (w/\log n)$ we improve the space complexity of the algorithm by Kirkpatrick and Reisch without increasing the asymptotic time complexity.

**Packed Sorting as Subroutine**

The algorithm above runs in $O(n \log \log n)$ time if $w = O(\log^c n)$ for some constant $c, c > 1$. In this section we achieve this time bound independently of $w$. That is, $n$ integers consisting of $w$ bits can be sorted in $O(n \log \log n)$ worst-case time on a unit-cost RAM with word length $w$.

The idea is to apply Lemma 2.7 $\log \log n$ times (with $k = 2$). This takes $O(n \log \log n)$ time and reduces the original sorting problem to a problem of sorting $n$ short integers each consisting of $w/\log n$ bits. This reduced sorting problem can be solved in $O(n \log \log n)$ time using a packed sorting
routine by Albers and Hagerup [2]. The original algorithm runs on a parallel machine; for the sake of completeness we present a simplified version for the RAM.

The main idea of packed sorting is to make use of the fact that even a sequential computer, such as the unit-cost RAM, is capable of parallel computation in so far that it can perform operations on w-bit words in constant time. The algorithm uses a realistic instruction set that includes comparison, addition, subtraction, bitwise AND, OR, and XOR, and left- and right-shift. All of these instruction are in \(AC^0\): they can be implemented by a constant-depth, polynomial-size circuit with unbounded fan-in. In particular, observe that the algorithm does not use multiplication, which is known not to be in \(AC^0\) [16].

The algorithm is a variation of Mergesort, where short subsequences are merged in sublinear time using the hidden parallelism of the RAM. Assuming that we are given an efficient subroutine for merging sequences of fixed length \(k\) we want to merge two sorted sequences \(x_1, x_2, \ldots, x_n\) and \(y_1, y_2, \ldots, y_n, \ n > k\). As a first step we merge the two sequences \(x_1, x_2, \ldots, x_k\) and \(y_1, y_2, \ldots, y_k\) into a sorted sequence \(z_1, z_2, \ldots, z_{2k}\) using the given subroutine. Without loss of generality we assume that \(z_{2k} = x_k\). Then, the sequence \(z_1, z_2, \ldots, z_k\) will consist of the \(k\) smallest elements in sorted order. Furthermore, none of the elements \(z_{k+1}, z_{k+2}, \ldots, z_{2k}\) will be larger than \(x_{k+1}\) and hence we have reduced the original problem to that of merging the sequences \(z_{k+1}, \ldots, z_{2k}, x_{k+1}, \ldots, x_n\) and \(y_{k+1}, \ldots, y_n\).

To use this technique efficiently we first divide the input into \(n/k\) subsequences, each consisting of \(k\) integers, and sort each of these subsequences, using a standard comparison-based sorting algorithm, in \(O(k \log k)\) time. Now, using the technique discussed above, we can sort in time

\[
n + n/k \log k + T(k) \frac{n}{k} \log \frac{n}{k},
\]

where \(T(k)\) is the time to merge two sequences of length \(k\). We refer the reader to Chapter 6 for the implementation of a merging subroutine that runs in \(T(k) = O(k \log k)\) time if \(k\) elements can be fitted into one machine word. For \(k = \log n\) the expression above is \(O(n \log \log n)\) and we have the following lemma.

**Lemma 2.9:** A sequence of \(n\) integers, each consisting of \(w/\log n\) bits, can be sorted in \(O(n \log \log n)\) time on a unit-cost RAM with arbitrary word length \(w\).
We now have all the necessary building blocks to derive the main result of this section. From Corollary 2.4 it follows that the problem of sorting $n$ integers from the set $\{0, 1, \ldots, 2^b - 1\}$ can be reduced in linear time to a problem of sorting $n$ integers from the set $\{0, 1, \ldots, 2^{b/2} - 1\}$. Applying this reduction $\log \log n$ times takes $O(n \log \log n)$ time and leaves us with an integer sorting problem that can also be solved in $O(n \log \log n)$ time according to Lemma 2.9. We can now state the following theorem.

**Theorem 2.10:** A sequence of $n$ integers, each consisting of $w$ bits, can be sorted in $O(n \log \log n)$ time on a unit-cost RAM with word length $w$.

As above, it is possible to reduce the space requirements of the algorithm by dividing the numbers into more than two characters and, if we are prepared to accept a randomized algorithm, we can achieve linear space complexity by implementing the bucket table with universal hashing [23, 26].

Observe that the packed sorting algorithm cannot distinguish between duplicate keys since several keys are stored within a single machine word. An easy way to overcome this problem is to replace each key $x_i$ by the number $nx_i + 1$. In this way the keys become distinct and we can identify the record that the key belongs to after the sorting is finished. These new numbers might occupy two machine words, but using Forward radix sort this sorting problem can be reduced in linear time to a problem of sorting word-sized integers.

A slightly better time bound is achieved if we follow Albers and Hagenup [2] and use packed sorting instead of a comparison-based algorithm to sort the elements of each of the $n/k$ sublists. This can be done in $O(\log^2 k)$ time if $k$ elements fit within a machine word. The idea is to merge ever longer sequences just as in the standard sequential Mergesort algorithm, but the merging is performed in parallel using the packed merging technique presented in Chapter 6. Using this technique we can merge the sublists of size two in constant time. Merging sublists of size four takes twice this time. Repeating this process until the word is sorted therefore takes $O(1 + 2 + \ldots + \log k) = O(\log^2 k)$ time. Using this result the total time for the merge sort algorithm is

$$n + \frac{n}{k} \log^2 k + T(k) \frac{n}{k} \log \frac{n}{k}.$$ 

For $k = \log n \log \log n$ and $T(k) = O(\log k)$ this expression is linear and we get the following lemma.

**Lemma 2.11:** A sequence of $n$ integers, each containing $w/\log n \log \log n$ bits, can be sorted in linear time on a unit-cost RAM with word length $w$. 


We can now construct a new sorting algorithm for short integers. If the numbers to be sorted consist of $b$ bits, where $b \leq w$, $(b/w) \log n \log \log n$ applications of Forward radixsort reduces the size of the numbers so that Lemma 2.11 can be used.

**Theorem 2.12:** A sequence of $n$ integers, each consisting of $b$ bits, where $b \leq w$, can be sorted in

$$O\left( n \log \frac{b \log n \log \log n}{w} \right)$$

time on a unit-cost RAM with arbitrary word length $w$.

**Discussion**

The first algorithm to make use of the hidden parallelism of the unit-cost RAM model is due to Paul and Simon [59]. They showed that if the machine word is large enough to accommodate $n^2$ elements, it is possible to compute the rank of each element in a single operation. The idea is simple. All $n^2$ comparisons between $n$ integers can be performed using a single subtraction operation if the elements are arranged properly. One word contains $n$ copies of the sequence $x_1, x_2, \ldots, x_n$; the other contains $n$ copies of $x_1$ followed by $n$ copies of $x_2$ and so on.

Albers and Hagenup [2] combined this idea by Paul and Simon with Batcher's bitonic sort [15] and arrived at an algorithm that sorts in linear time provided that $\Omega(\log n \log \log n)$ keys can be packed into one word. Lemma 2.9 and Lemma 2.11 depend on this result.

Fredman and Willard [39] were the first to sort integers in $o(n \log n)$ worst-case time on a RAM with arbitrary word length. Their algorithm, which is based on so-called fusion trees, requires only linear extra space and has time complexity $\Theta(n \log n / \log \log n)$. However, it is quite complicated, has a large constant factor, and requires long word lengths to be efficient. Combining the fusion tree algorithm with a van Emde Boas tree [77] the time complexity can be reduced to $\Theta(n \sqrt{\log n})$ at the cost of utilizing extra space.

A more complete treatment of new fast integer sorting algorithms, including several parallel algorithms and a randomized sorting algorithm that runs in linear time if the word length $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, can be found in the article by Andersson, Hagenup, Nilsson, and Raman [8]. Similar results are also presented in an independent article by Han and Shen [43].
Little is known about the lower bound for integer sorting. In the general RAM model only the trivial $\Omega(n)$ lower bound is known. Ben-Amram and Galil [18] give lower bounds in some restricted RAM models. In particular, they show that sorting requires $\Omega(n \log n)$ time in a restricted RAM model with comparison, addition, subtraction, multiplication, and bitwise boolean operations. Using multiplication it is possible to simulate left shifts in their model. However, if we add the right-shift operation the lower bound no longer holds.

The idea to manipulate machine words using bitwise operations has also been used to obtain improved time bounds for searching. Anderson [6] presents a sublogarithmic search algorithm that does not use multiplication. Tonup [74] presents a priority queue with sublogarithmic operations. Brodnik and Munro [21] build data structures with constant time membership queries using a minimal amount of space.

2.7 IMPROVED BOUNDS ON STRING SORTING

We now have assembled all the necessary tools to implement a number of new and improved radix sorting algorithms. Using Forward radixsort we can reduce a string sorting problem to an integer sorting problem of appropriate size. Then we can apply the integer sorting algorithms of the previous section. As a warmup we first study the combination of Forward radixsort and Bucketsort.

**Bucketsort as Subroutine**

If we let the characters of the alphabet occupy $v = \log n$ bits, we can perform the sorting in Step 2 of Forward radixsort in linear time by distributing the numbers among $2^v = \Theta(n)$ buckets. The total time complexity of the algorithm is $\Theta(b / \log n + n)$. This is not an improvement over previous bounds. In fact, it is only a more complicated way to achieve the same time complexity as in the basic version of Forward radixsort.

**The Algorithm of Theorem 2.8 as Subroutine**

We now turn our attention to the first integer sorting algorithm of the previous section, the one that was implemented by applying Forward radixsort recursively and then switching to Bucketsort. Under the assumption that $b \leq v$, this integer sorting algorithm sorts $n$ integers, each consisting
of $b$ bits, in $\Theta(n \log(b/\log n))$ time. If we use this algorithm as a subroutine in Step 2 of Forward radix sort we get a string sorting algorithm with time complexity

$$\Theta\left(\frac{b}{b} + n + n \log \frac{b}{\log n}\right).$$

The minimum $\Theta(n \log(b/\log n))$ of this expression is attained for $b = B \ln 2$. However, it is not a priori clear how to choose $b$ to achieve this minimum, since $B$ cannot be expected to be known in advance. Below we show how to get around this problem.

Start by choosing $b = \log n$ and run the first step of Forward radix sort until either all strings have been separated or the number of $b$-bit characters processed during the step exceeds $2n$. If not all strings have been separated in this first step, start over again but this time choose $b = 2 \log n$. Continue doubling $b$ until the first step terminates after having processed at most $2n$ characters. If this process stops after one iteration, $B = O(n)$ and the algorithm will run in linear time. If more than one iteration is needed, $bn < B < 2bn$ and hence $b = \Theta(B)$. If $w = \Omega(B)$ and each of these iterations will take $O(n)$ time. Thus, the total time for this doubling procedure will be $\Theta(n \log(b/\log n))$. Next, we run the second step of Forward radix sort using the value of $b$ for which the first step was completed. This requires $\Theta(n \log(b/\log n))$ time as well. Hence, the cost of the doubling can be included in the sorting cost without affecting the asymptotic complexity. In total, for a word length $w = \Omega(B)$, we obtain the time complexity $\Theta(n + n \log(b/\log n))$, an improvement over the bucketing algorithm.

The time bound above is valid only for $w = \Omega(B)$. But even if $w = o(B)$ the same approach can be used. We add one more condition to the doubling procedure: the doubling is terminated when $b > w$. We then run the second step of Forward radix sort with $b = w$. We summarize the results in the following theorem,

**Theorem 2.13:** A sequence of $n$ binary strings with distinguishing prefixes of average length $B$ can be sorted in

$$\Theta\left(n + n \log \frac{B}{\log n}\right)$$

time on a unit-cost RAM with word length $w$ if $w = \Omega(B)$ and in

$$\Theta\left(n \frac{B}{w} + n \log \frac{w}{\log n}\right)$$

if $w = o(B)$. 


In particular, for input from a stationary ergodic process with entropy \( H \) per bit, the running time is \( \Theta(n \log \frac{H}{w}) \) if \( w = \Omega(B) \).

Relating the time bounds of Theorem 2.13 to the trivial lower bound in Observation 2.5 we see that the algorithm is optimal if
\[
B = \Omega\left(w \log \frac{w}{\log n}\right).
\]

Furthermore, the algorithm is faster than the basic version of Forward radixsort and the radix sorting algorithm by Paige and Tarjan [57], both of which have a time complexity of \( \Theta(nB/\log n + n) \). To see this we make the additional assumption that \( \log n = o(w) \), since otherwise all of these algorithms are optimal. Under this assumption the time complexity in Theorem 2.13 is, in fact, better: \( B/w = o(B/\log n) \) since \( \log n = o(w) \), and \( \log(w/\log n) = o(B/\log n) \) since \( w = o(B) \).

The Algorithm of Theorem 2.12 as Subroutine

The integer sorting algorithm used as a subroutine in the previous section runs in \( O(n \log \log n) \) time if \( w = O(\log^c n) \) for some constant \( c, c > 1 \), and hence we can expect to improve on the string sorting routine above only for large word lengths. The algorithm of Theorem 2.12 runs in time
\[
\Theta\left(n \log \frac{b \log n \log \log n}{w}\right).
\]

Using this algorithm as a subroutine the total cost for Forward radixsort is
\[
\Theta\left(\frac{B}{b} + n + n \log \frac{b \log n \log \log n}{w}\right).
\]

The minimum value of this expression is achieved for \( b = B \ln 2/n \). This gives a string sorting algorithm with time complexity
\[
\Theta\left(n + n \log \frac{B \log n \log \log n}{w}\right).
\]

Once more we will use a doubling procedure to find a value of \( b \) that achieves this time bound. This time we start with \( b = w/\log n \log \log n \). As before the doubling continues until the first step of Forward radixsort has finished in linear time. Also in this case it is straightforward to show that the cost of the doubling procedure does not increase the overall asymptotic time complexity.
The time bound is valid only if $B = O(w)$. For larger values of $w$ we get the time complexity

$$\Theta\left(\frac{B}{w} + n \log \log n\right).$$

This is optimal if $B = \Omega(w \log \log n)$. We summarize the results in the following theorem.

**Theorem 2.14:** A sequence of $n$ binary strings with distinguishing prefixes of average length $\bar{B}$ can be sorted in

$$O\left(n + n \log \frac{B \log n \log \log n}{w}\right)$$

time if $w = \Omega(\bar{B})$, and in time

$$O\left(n \frac{B}{w} + n \log \log n\right)$$

if $w = o(\bar{B})$.

**Discussion**

The time complexity of Theorem 2.13 is an improvement over the previously best known bounds by Paige and Tarjan [57] and Chen and Reif [24]. For large word lengths, the time bound of Theorem 2.14 reduces the time complexity even further.

The sorting algorithm by Chen and Reif [24] has time complexity $\Theta(n \log \frac{\log n}{\log \log n})$, to be compared with the $\Theta(n \log \frac{1}{\log n})$ time bound of Theorem 2.13. The two results can be directly compared since they employ the same statistical model. (The mixing condition of Definition 1.6 is implicitly assumed by Chen and Reif since their proof depends on an application of a theorem by Szpankowski [69] that requires this very same condition.)

The assumption that $w = \Omega(\bar{B})$ is needed in the analysis made by Chen and Reif. Although they claim to handle input strings of arbitrary length, their result relies on the assumption that each distinguishing prefix fits into a constant number of machine words. For example, they assume that element comparisons can be made in constant time.

**2.8 CONCLUSIONS**

The sorting problem is of fundamental importance in computer science and many algorithms have sorting as an essential ingredient. Alphabetic
sorting may at first look like a rather specialized case, but virtually all computers use a binary data representation. In fact, the basic data types of most computers are integers, floating-point numbers, and character strings, all of which are well suited for radix sorting.

We bring together the advantages of several well established radix sorting algorithms in one algorithm, Forward radixsort. The new algorithm is simple and combines the features of the range reduction technique of Kirkpatrick and Reisch [49] with the trie-based sorting algorithm of Paige and Tarjan [57]. Also, Forward radixsort cleanly separates the integer sorting subproblem from the string sorting problem. Hence, this algorithm offers a solution to the problem stated by Paige and Tarjan [57]: "An interesting open problem is whether the ideas of Kirkpatrick and Reisch can somehow be applied to the variable-length lexicographic sorting problem we have considered here."

The main new theoretic results are the asymptotically optimal reduction of any string sorting problem to an integer sorting problem and the \(O(n \log \log n)\) worst-case time integer sorting algorithm. The most obvious open problem is of course to try to close the gap between the \(\Omega(n)\) lower bound for integer sorting and the \(O(n \log \log n)\) upper bound. Also, how fast can we sort using deterministic algorithms and linear space?

From a practical point of view the most important results are probably the application of distributive partitioning to MSD radixsort and the basic Forward radixsort algorithm that combines the advantages of LSD and MSD radixsort. In Chapter 4 we show that these algorithms do, in fact, perform very well in practice.
3

Radix Searching

To rapidly recover information from memory is one of the basic operations of many computer applications. In its simplest form the search problem can be defined as follows. Given a set of elements \( \{x_1, x_2, \ldots, x_n\} \) with unique keys and a query key \( k \), find out if one of the elements has a key that equals \( k \) and, if so, find the contents of the other fields of this element.

There are many variations of this problem. In an attempt to bring some order we make a distinction between comparison-based searching methods that use only comparison between keys, and character-based methods that make use of the fact that the keys are represented as character strings. Another natural way to divide the rich collection of searching algorithms is to distinguish between algorithms that use some precomputed data structure to facilitate the searching and algorithms that only use a simple linear data structure such as an array or a linked list.

In the character-based model the only way to find an element in an unsorted array is to perform an exhaustive search. However, if the input is sorted and stored in a vector we can use binary search [50] to find an element using at \( O(\log n) \) comparisons.

The corresponding problem for characters strings was solved only recently. Andersson, Hagenup, Hästad, and Petersson [7, 9] show that given a \( k \)-character query string and an array of \( n \) strings, arranged in alphabetical order, finding a matching string, or reporting that such a string does not exist, requires

\[
\Theta\left(\frac{k \log \log n}{\log \log (4 + \frac{k \log \log n}{\log n})} + k \log n\right)
\]

character comparisons in the worst case.
With a precomputed data structure available it is possible to speed up the searches. For comparison-based methods there is a plethora of data structures, such as skip lists [62] and different versions of balanced and unbalanced search trees [5]. Many of these data structures support searching, insertion, and deletion of elements using $O(\log n)$ key comparisons.

Among the character-based methods hashing [50] is probably the most commonly used. It is a randomized scheme with constant time expected search cost. However, traditional hashing schemes do not preserve the order of the elements and cannot support operations like neighbor search and range queries efficiently.

In this chapter we study character-based searching methods using a precomputed data structure that keeps the elements in sorted order. The most well known data structure of this type is the trie [36]. The name trie comes from the word retrieval. An example of a trie can be seen in Figure 3-1.

An important variation of the searching problem defined above is the problem of locating substrings in a larger text. A path-compressed trie containing all suffixes of the text, a so called suffix tree [53], is a commonly used data structure for this problem. An example can be seen in Figure 3-2. The suffix tree has many applications in addition to string searching, including approximate string matching, compression schemes, and analysis of genetic sequences. The articles by Apostolico [14] and Manber and Myers [52] contain extensive lists of references.
In this chapter we present a new trie data structure, the level-compressed trie, or LC-trie, and analyze its behavior. Based on the analysis we argue that level compression improves the performance of tries considerably in many practical situations. Our analysis shows that the expected depth of an element in an LC-trie is $O(\log^* n)$ for uniformly distributed data and $\Theta(\log \log n)$ for data from a Bernoulli-type distribution.

### 3.1 TRIE STRUCTURES, BACKGROUND

We remind the reader of the definition of a trie. We say that a string $v$ of length $i$ is the $i$-prefix of a string $u$ if there is a string $w$ such that $u = vw$. The string $w$ is called the $i$-suffix of $u$. We define a binary trie in the following way.

**Definition 3.1:** A binary trie containing $n$ elements is a tree with the following properties.

- If $n = 0$, the trie is an empty tree.
- If $n = 1$, the trie consists of one node representing the element.
- If $n > 1$, the trie consists of a node of degree 2. The left child is a binary trie containing the 1-suffixes of all elements starting with 0 and the right child is a binary trie containing the 1-suffixes of all elements starting with 1.

Note that the strings are assumed to be prefix-free: no string can be a prefix of another string. In particular, this implies that there cannot be any duplicate strings. However, if all strings to be stored in the trie are unique, it is easy to ensure that the strings are prefix-free by appending a special end-of-string marker to each string. For example, we can append the string $1000\ldots$ to the end of each string. In fact, any nonperiodic string can be used.

A natural way to decrease the search cost in a binary trie is to use more than one bit for branching. In this way we obtain a multi-digit trie as defined below.

**Definition 3.2:** A multi-digit trie containing $n$ elements is a tree with the following properties.

- If $n = 0$, the trie is an empty tree.
3.2 LC-TRIES

- If \( n = 1 \), the trie consists of one node representing the element.

- If \( n > 1 \), the trie consists of a node of degree \( 2^i \), \( i \geq 1 \). For each possible \( i \)-prefix \( x \) there is a child, which is a multi-digit trie, containing the \( i \)-suffixes of all elements starting with \( x \).

A number of sophisticated variations of multi-digit tries can be found in the literature. The data structure introduced by Tarjan and Yao [72] allows element location in \( O(w/\log n) \) worst-case time, where the element that can be stored in the data structure come from the set \( \{0, 1, \ldots, 2^w - 1\} \). This structure does not support range and neighbor queries efficiently.

The \( p \)-fast trie and the \( q \)-fast trie by Willard [81] support element retrieval and nearest neighbor search in \( O(\sqrt{w}) \) time in the worst case using \( O(n\sqrt{w}2\sqrt{w}) \) and \( O(n) \) space respectively. Range searching is performed in \( O(\sqrt{w} + s) \) time, where \( s \) is the size of the output. Similar results are also obtained by Karlsson [46].

The \( y \)-fast trie by Willard [80] is a trie structure that uses perfect hashing [37]. This data structure requires \( O(n) \) space and supports element retrieval and nearest neighbor search in \( O(\log w) \) time in the worst case and range search in \( O(\log w + s) \) time.

The reason why rather sophisticated methods are required in order to obtain efficient multi-digit tries is that a careless use of large branching factors will introduce a large number of empty leaves. This may cause a multi-digit trie to perform much worse than a binary trie with respect to space requirements and proximity operations.

### 3.2 LC-TRIES

In an attempt to find a simple trie structure that supports fast retrieval of elements and retains efficient operations for neighbor and range searches we focus our interest on multi-digit tries that do not contain any superfluous leaves.

**Definition 3.3:** A multi-digit trie is **dense** if it contains the same number of leaves as the corresponding binary trie.

Dense tries have the same favorable properties as binary tries with respect to space efficiency and proximity operations. Thus, by finding a dense trie with significantly higher branching factors than a binary trie, we improve the cost of search operations without loosing any of these nice properties.
Figure 3-3. Applying level-compression to a binary trie.

We also want the external path length, the sum of the depths of the leaves, of the trie to be small.

**Definition 3.4:** A dense trie is optimal if no other dense trie containing the same elements has smaller external path length.

This can be achieved quite easily. The idea is to replace the $i$ highest complete levels of the binary trie with a single node of degree $2^i$; this replacement is performed recursively on each subtree. This new data structure, the level-compressed trie, or LC-trie, is not only dense but also optimal. An example of a binary trie and the corresponding LC-trie is given in Figure 3-3.

**Definition 3.5:** A level-compressed trie, or LC-trie, is multi-digit trie with the following properties.

- The degree of the root is $2^i$, where $i$ is the smallest number such that at least one of the children becomes a leaf.
- Each child is a level-compressed trie.

It follows immediately from the definition that an LC-trie is dense. In the next theorem we prove that an LC-trie is optimal.

**Theorem 3.6:** A dense trie is optimal if and only if it is an LC-trie.

**Proof:** Let $S$ be a set of elements. The theorem follows from the following three facts.
There is at least one optimal dense trie containing $S$.

There is exactly one LC-trie containing $S$.

An optimal dense trie is an LC-trie.

An optimal trie exists since there are only finitely many dense tries that can contain $S$. The second fact follows directly from Definition 3.5. The proof of the third fact will be by contradiction. Assume that the dense trie $T$ is optimal and not an LC-trie. Since $T$ is not an LC-trie there is an internal node $v$ in $T$ such that each child of $v$ has degree at least 2. We construct a new dense trie $T'$ in the following way.

- Increase the degree of $v$ by a factor 2.
- For each child $w$ of $v$ the following actions are performed.
  - Replace $w$ by two nodes $w_1$ and $w_2$, each with half the degree of $w$.
  - Place the subtrees of $w$ below $w_1$ and $w_2$ in the obvious way.

It is easily seen that $T'$ is dense and that no leaf in $T'$ has a larger depth than the corresponding leaf in $T$. Thus, the external path length of $T'$ is not larger than that of $T$. The procedure above is repeated until we arrive at a dense trie $T''$, where at least one of $v$'s children is a leaf. Let $x$ be such a leaf. In $T''$ the distance between $v$ and $x$ is 1, while this distance is larger in $T$. Thus, $T''$ has smaller external path length than $T$. This contradicts the fact that $T$ is optimal. □

**Example 3.7:** The LC-trie is a simple data structure that combines the good properties of bucketing and ordinary tries. We demonstrate this with an example large enough to exhibit the differences between LC-tries and other similar methods such as bucketing, N-trees, and quadtrees.

Consider the map of seamounts and guyots in an area of the Pacific Ocean [27] shown in Figure 3-4. In order to perform searches and range queries we want to store the information in a suitable data structure. A bucketing scheme with 144 buckets (= the number of volcanos) results in the partition of Figure 3-5. In this case a range search in the sparse lower right hand corner of the map will be inefficient since we will have to examine a lot of buckets to find only a few items. Searching in a small area in the dense lower left hand corner will also be inefficient. Even for an unsuccessful search we have to examine all the volcanos in one bucket.
Figure 3-4. A map of seamounts in the Pacific.

Figure 3-5. Bucketing.
Figure 3-6. Recursive bucketing.

Figure 3-7. Partitioning in a quadtree.
Figure 3-8. a. The quadtree corresponding to Figure 3-7; b. The level-compressed quadtree.
The problem with dense areas can to some extent be neutralized by
applying bucketing recursively, as shown in Figure 3-6. This structure is
known as an N-tree [35, 70]. Empty buckets in sparse areas is still a prob-
lem though.

Using a quadtree [64] the map is partitioned as shown in Figure 3-7. The
resulting trie structure is shown in Figure 3-8a. In this case the partition
adapts to the varying density of the map and during a range search in the
sparse area we only have to examine a few buckets. However, searching
in a small dense area we have to follow a long path down the tree to find a
single seamount.

A level-compressed quadtree gives the same partition as a plain quad-
tree and therefore the nice properties with respect to proximity problems
are preserved. Search operations are faster than in a quadtree since the
depth of the leaves is reduced, as can be seen in Figure 3-8b. In this ex-
ample the average depth of a leaf in the level-compressed quadtree is 3.0
compared to 4.5 for the ordinary quadtree.

3.3 IMPLEMENTATION

The standard implementation of a k-ary trie is to store a set of children
pointers at each internal node. Different data structures have been used for
this purpose. Some of the most common choices are listed below.

- Each internal node consists of an array of pointers to the descendants.
  The length of the array equals the size of the alphabet.

- The non-null outgoing pointers are stored in a linked list.

- The non-null outgoing pointers are stored in a balanced binary search
tree.

All of these implementations have a large space overhead, reducing the
usefulness in many applications. A space efficient alternative is to store
the children of a node in consecutive memory locations. In this way, only
a pointer to the leftmost child is needed.

This technique can also be used to efficiently implement an LC-trie. The
nodes are stored in an array and each node is represented by a record con-
taining two nonnegative integers bits and adr.

- If bits > 0 the record represents an internal node of degree \(2^\text{bits} \). The
  children of this node are stored in positions \( adr + i, 0 \leq i < 2^{\text{bits}} \).
If bits = 0 the record represents a leaf. If adr = 0 the leaf is empty, otherwise adr contains a reference to the element represented by the leaf.

The degree of a node is always a power of 2 and hence it can be represented using a small number of bits. In fact, if the tree has n leaves, log log n bits will suffice. Thus, in a typical application bits will easily fit into one byte. The fact that we use adr both as a reference into the array and as a pointer to an element causes no problem since integers and pointers are typically represented in the same way.

The search algorithm can easily be implemented in such a way that the search time is proportional to the number of nodes visited on the traversed search path. Let s be the string searched for and let Extract(s, k, m) be a function that returns the number given by the m bits starting at position k in s. It should be clear that if m bits fit into one machine word this operation can be performed in constant time in the unit-cost RAM model by simple arithmetic operations. We denote the array representing the tree by T. The root is stored in T[0]. The algorithm uses two temporary variables k and node.

```c
k = 0;
node = 0;
while T[node].bits > 0 {
    node = T[node].adr + Extract(s, k, T[node].bits);
    k = k + T[node].bits;
}
return T[node].adr;
```

It is also possible to combine path compression and level compression. Each internal node of degree two that has an empty subtree is removed, and at each internal node we use an index that indicates the number of bits that have been skipped. The path-compressed LC-trie is also quite easy to implement. In fact, we can use essentially the same implementation as above. We store the nodes in an array and siblings are stored in consecutive positions. Each node is represented by three numbers, two of which indicate the number of bits to be skipped and the position of the leftmost child. The third number indicates the number of children. We illustrate how this can be done with a more detailed example where we build a compact representation of a suffix tree for a DNA sequence.

**Example 3.8:** DNA sequences are strings made up from the four nucleotide bases A, G, T, and C (A = adenine, G = guanine, C = cytosine, T =
Figure 3-9. a. Trie; b. Binary trie; c. Patricia tree; d. LC-trie.
thymine). In this example we consider the first 15 nucleotide bases of the Epstein-Barr virus: AGAATTGCTCTTGCT. In Figure 3-9 we show how the suffixes of this string can be represented by different trie structures. The numbers at the nodes are pointers to the suffixes of the string.

a. A traditional trie. Each internal node has four outgoing edges corresponding to the bases A, G, T, and C, respectively.
b. The corresponding binary trie. We have coded A, G, T, and C, as 00, 01, 10, and 11, respectively. The string 1000... has been appended to the end of the sequence to make sure that all suffixes are unique.
c. The corresponding Patricia tree.
d. The corresponding LC-trie.

Figure 3-10 shows the array representation of the LC-trie, each entry represents a node. The nodes are numbered in breadth-first order starting at

<table>
<thead>
<tr>
<th>branch</th>
<th>skip</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3-10. Array representation of the LC-trie in Figure 3-9d.
the root. The number in the \textit{branch} column indicates the number of bits used for branching at each node. A value \( k \geq 1 \) indicates that the node has \( 2^k \) children. The value \( k = 0 \) indicates that the node is a leaf. The number in the \textit{skip} column is the Patricia skip value, the number of bits that can be skipped during a search operation. The value in the \textit{pointer} column has two different interpretations. For an internal node, it is used as a pointer to the leftmost child; for a leaf it is used as a pointer to the position of the string in the text.

As an example we search for the string \texttt{TGG}. The binary code for this string is 101101. We start at the root, node number 0. We see that \textit{branch} is 3 and \textit{skip} is 0 and therefore we extract the first three bits from the search string. These 3 bits have the value 5 which is added to \textit{pointer}, leading to position 6 in the array. At this node \textit{branch} is 2 and \textit{skip} is 0 and therefore we extract the next two bits. They have the value 2. Adding 2 to \textit{pointer} we arrive at position 15. This node has \textit{bits} equal to 0 which implies that it is a leaf. The pointer value 5 gives the position of the string in the text. Observe that it is necessary to check whether this constitutes a true hit. For example, the search for \texttt{TGG} is exactly mirrored by a search for \texttt{TCA}; the former string is present in the text, whereas the latter is not.

\textbf{Further Compression}

In the experimental work in Chapter 5 we have used a simple strategy to further reduce the size of the I-C-trie. Internal nodes with a branching factor larger than two are required to have a Patricia skip value equal to zero. This implies that the path compression may be incomplete, but on the other hand we can replace the two fields \textit{branch} and \textit{skip} by one integer, one bit of which is used to distinguish between the different types of values. This will reduce the size of the nodes and the fact that we may end up using somewhat fewer skip values turns out to have little effect in practice. Compared to the original I-C-trie, the number of nodes will only increase by a small fraction and hence we get an overall size reduction. In a recent article Clark and Munro [25] reduce the space requirements even further by using a minimum number of bits for pointers and skip values and by representing the tree structure as compactly as possible.

\textbf{Construction}

In practice a simple method can often be used. First, construct an array of pointers to all suffixes of the text and sort this array with an appropriate
radix sorting algorithm. Given this sorted array, it is easy to make a top-
down construction of the LC-trie.

The same method can be used also to build a suffix tree. But in this
case the worst-case time is quadratic. Even so, we have found this simple
approach useful for many types of data and only for one of the texts that
we employed in the experiments in Chapter 5, a DNA sequence containing
long repeated substrings, this procedure was conspicuously slow.

A suffix tree implemented as a level-compressed Patricia tree can be
constructed in $O(n \log m)$ time and $O(n)$ space, where $n$ is the size of
the text and $m$ is the size of the alphabet. A brute force method that
achieves this starts by constructing an ordinary suffix tree, that is, a path-
compressed $m$-ary trie containing all suffixes of the text. This can be done
in $O(n \log m)$ time and $O(n)$ space [53, 75]. Assuming that the text ends
with a special end marker and that the binary encodings of the characters
are of equal length the LC-trie can now be constructed in two steps.

First, each node of the suffix tree is replaced by a binary Patricia tree
and the skip values are updated. This takes $O(n \log m)$ time since each
color character consists of $\log m$ bits and the total number of nodes is $O(n)$ both
in the original and in the modified tree. Then, the level compression is
performed during a top-down traversal of this modified tree. The topmost
level complete levels of the tree are replaced by a single node of degree $2^l$ and
this replacement is performed recursively on each subtree. This step can be
performed in $O(n)$ time, since the total number of nodes in the tree is $O(n)$.

Array Implementations

An alternative to the standard trie implementation has been suggested by
Manber and Myers [52]. They dispense with the trie altogether and store
references to all suffixes in a suffix array or PAT array [42]. This technique
was used in the Oxford English Dictionary project at the university of
Waterloo. The suffix array consists of an array of pointers. Each pointer points
to a suffix in the text and the pointers are sorted according to the suffix that
they point to. String location is performed by a binary search in the array.
One drawback of this simple structure is that the worst-case time for
searching may be high. Also, the text has to be accessed at each step of the
search, which might become still if the text is stored on secondary memory.

In order to decrease the worst-case cost, Manber and Myers also intro-
duced an augmented suffix array, where two numbers were added to each
entry. These numbers indicate how many characters at the beginning of each string that can be skipped during comparisons. In this way, the worst time search cost was reduced. But also in this case the text has to be accessed at each step of the search. Since the augmented suffix array has a large space overhead, another method that combines a suffix array with a bucket array was suggested for practical use. Strings from a k-ary alphabet are treated as k-ary numbers and the universe is split into a number of buckets. Each bucket contains a pointer into the suffix array. Thus, a string location is reduced to a binary search within a small part of the suffix array. Choosing n/4 buckets the total space requirement increases by only one fourth compared to the suffix array alone.

In Chapter 5 we compare the augmented suffix array with several different trie implementations. It turns out that the LC-trie uses slightly more space than the suffix array but offers a faster search routine. In particular we show that the combination of an LC-trie and a suffix array gives an efficient algorithm for searching in large texts on secondary memory.

3.4 Analysis

We will study the expected average depth on an LC-trie for data \( x_1, \ldots, x_n \) from different statistical distributions. Let \( d_{n,i} \) denote the depth of \( x_i \) in a trie. The average depth of the trie is defined as \( A_n = \frac{1}{n} \sum_{i=1}^{n} d_{n,i} \). This parameter is proportional to the average successful search time. We denote the average depth of a trie and a level-compressed trie by \( A_n^{\text{trie}} \) and \( A_n^{\text{level}} \), respectively.

Uniformly Distributed Data

In the following analysis we give a tight asymptotic bound on the expected average depth of an LC-trie. Our sample \( x_1, \ldots, x_n \) consists of the binary fractions of uniformly distributed real numbers on the unit interval. We choose the indices in such a way that \( x_1 < \cdots < x_n \), and write \( x_0 = 0 \) and \( x_{n+1} = 1 \). In our analysis we will use the random variable

\[
M_n = \max_{0 \leq i \leq n} (x_{i+1} - x_i),
\]

the maximal spacing between adjacent elements. The distribution of \( M_n \) is well studied; already 1897 Whitworth [79] found the distribution function. Devroye [29] and Slud [67], among others, have studied tail estimates for \( M_n \). We will use the following version of a lemma by Slud [67].
Lemma 3.9: If \( \delta \) is a positive constant, then
\[
P \left( \left\lvert nM_n / \ln n - 1 \right\rvert > \delta \right) = O(n^{-5}).
\]

There is a close relationship between \( M_n \) and the branching factor at the root of an LC-trie. If \( M_n < 2^{-1} \) for some integer \( i \), the branching factor will be at least \( 2^i \) since every subinterval of size \( 2^{-i} \) will contain at least one element. On the other hand, if \( M_n > 2 \cdot 2^{-(i+2)} \) the branching factor will be at most \( 2^{i+1} \) since at least one subinterval of size \( 2^{-(i+2)} \) will be empty. This is illustrated by Figure 3-11. Hence, the expected branching factor at the root of the tree is \( \Theta(\ln n / n) \) and the expected number of elements in a subtree is \( \Theta(\ln n) \). We have the following theorem.

**Theorem 3.10:** For an independent random sample taken from the uniform distribution the expected average depth of an LC-trie \( \mathbb{E}(A_n^{\text{level}}) = \Theta(\log^* n) \).

**Proof:** First we determine an upper bound. We will show that the degree at the root is \( \Omega(\ln n / \log n) \) with high probability. This also holds at the lower levels of the trie, since the elements in a subtree are a random sample of independently chosen points in an interval of some length \( 2^{-i} \); thus, the analysis applies with due alteration of details.

Let \( x \) be an element and let \( T_x \) denote the number of elements stored in the same subtrie as \( x \). Choose the integer \( i \) such that \( \frac{2\ln n}{n} < 2^{-i} \leq \frac{4\ln n}{n} \) and split the interval \([0, 1]\) into subintervals of size \( 2^{-i} \). Let \( l \) denote the subinterval into which \( x \) falls. Let \( y \) be any of the other \( n - 1 \) elements and let \( p \) be the probability that \( y \) falls into \( l \). Then, \( p = 2^{-i} \leq \frac{4\ln n}{n} \).

In the expected case, the effect of choosing \( i \) like this is that \( M_n \leq 2^{-i} \). Therefore every subinterval of size \( 2^{i-1} \) will contain at least two elements and hence the number of children of the root will be at least \( 2^i \). This implies that the elements that are stored in the same subtrie as \( x \) will be a subset
of the elements in $I$. From this we will deduce that for some constant $C$, 
$\Pr(T_x > C \ln n)$ is small. This is done by studying the random variables $M_n$ 
and $n_1$, where $n_1$ denotes the number of elements (not counting $x$) falling 
into $I$.

Clearly, $n_1$ obeys the binomial distribution $\mathcal{B}(n - 1, p)$. Let $C = 2c$. We 
use a Chernoff bound [56, Expression 4.12] to get a tail estimate for the 
binomial function,

$$P(n_1 > C \ln n) = O(n^{-1}).$$

If $M_n \leq 2^{-i}$ then, as explained above, $T_x \leq n_1$. In particular

$$M_n \leq 2^{-i} \land n_1 \leq C \ln n \Rightarrow T_x \leq C \ln n$$

and hence

$$P(M_n \leq 2^{-i} \land n_1 \leq C \ln n) \leq P(T_x \leq C \ln n).$$

From Lemma 3.9 we have

$$P\left(M_n > \frac{\ln n}{n}(1 + \delta)\right) = O(n^{-\delta}).$$

Hence

$$P(M_n > 2^{-i}) = P\left(M_n > \frac{\ln n}{n}\left(1 + \frac{n2^{-i}}{\ln n} - 1\right)\right) = O(n^{-\epsilon}),$$

where $\epsilon = \frac{n2^{-i}}{\ln n} - 1 \geq 1$. Combining these results, we get

$$P(T_x > C \ln n) \leq P(\neg(M_n \leq 2^{-i} \land n_1 \leq C \ln n)) \leq P(M_n > 2^{-i}) + P(n_1 > C \ln n) = O(n^{-\epsilon}).$$

From Theorem 3.6 it follows that the expected depth of an element in an 
LC-trie is never larger than that of the corresponding binary trie and from 
Theorem 1.4 we know that the expected external path length of a binary trie 
is $O(n \log n)$. Therefore, we can use the estimate $O(\log n)$ for the expected 
depth of an element in an LC-trie when $T_x > C \ln n$.

We inductively assert that the expected depth $E(A_n^{\text{tie}})$ can be bounded 
by

$$E(A_n^{\text{tie}}) \leq D \log^* n$$
for some constant \( D \). This is obviously true for \( n = 1 \). For \( n \geq 2 \) there is a constant \( D' \) such that
\[
E(A_n^{\text{trie}}) \leq 1 + \sum_{m=1}^{C\ln n} P(T_x = m) \cdot A(m) + P(T_x > C\ln n) \cdot O(\log n)
\]
\[
\leq 1 + \sum_{m=1}^{C\ln n} P(T_x = m) \cdot D \log^* (C\ln n) + O(1)
\]
\[
\leq D'(\log^* (\log n)) + D'.
\]
It can be arranged that \( D' \leq D \). Then the last expression can be bounded from above by \( D' \log^* n - 1 \) + \( D' \leq D \log^* n \) and hence \( E(A_n^{\text{trie}}) = O(\log^* n) \).

The lower bound can be established in a similar way. By choosing the integer \( i \) such that \( \frac{\ln n}{2^i} < 2^{-i} \leq \frac{\ln n}{2^{i-1}} \) we can ensure that
\[
P(T_x \leq c \ln n) \cdot E(A_{c\ln n}^{\text{trie}}) = o(1)
\]
for some positive constant \( c \) and the result follows from an argument similar to the one in the upper bound analysis.

**Corollary 3.11:** For an independent random sample taken from a distribution with density \( f(x), 0 < \alpha \leq f(x) \leq \beta \), the expected average depth of an LC-trie \( E(A_n^{\text{level}}) = \Theta(\log^* n) \).

**Proof:** Without loss of generality assume that \( 0 < \alpha < \beta < \infty \). We make the following observations.

- Let \( \Delta \) be an interval of size \( |\Delta| \) and let \( n_\Delta \) be the expected number of elements falling into \( \Delta \). Then \( n_\Delta \in \text{Bin}(n, p) \), where \( p \leq \beta|\Delta| \).

- Let \( M_n^f \) be the maximum spacing for data taken from \( f \) and let \( M_n^u \) be the maximum spacing for data from the uniform distribution over the interval \([0, 1]\), then \( P(M_n^f > x) \leq P(M_n^u > \alpha x) \). This can be shown in the following way. Consider the density \( g(x) = \alpha, 0 \leq x \leq 1/\alpha \). Then, \( P(M_n^g > x) = P(M_n^u > \alpha x) \). The fact now follows since the domain of \( f \) is a subinterval of the domain of \( g \) and \( f(x) \geq g(x) \) when \( x \in [0, 1] \).

Using the first observation we see that the tail estimate for the binomial equation in the proof of Theorem 3.10 holds if \( C \) is chosen large enough. If
we make the interval I larger by a factor of \( \frac{1}{i} \), that is, we choose \( i \) such that
\[
\frac{\ln n}{\alpha n} < 2^{-i} \leq \frac{2 \ln n}{\alpha n},
\]
the second observation gives
\[
P(M_n^f > 2^{-i}) \leq P(M_n^u > \alpha 2^{-i})
= P\left(M_n^u > \frac{\ln n}{n} \left(1 + \frac{\alpha n 2^{-i}}{\ln n} - 1\right)\right)
= O(n^{-i}),
\]
where \( \varepsilon = \frac{\alpha n 2^{-i}}{\ln n} - 1 > 0 \). Hence, if we choose \( C \) large enough (\( C = 2\beta \varepsilon / \alpha \) will do) we can finish the proof in the same way as in the uniform case.

The lower bound can also be established using the same technique. \( \square \)

**Data From a Bernoulli-Type Process**

Since tries are often used for storing text, it is natural to consider input from a Bernoulli-type process. This model may be viewed as a first approximation of text, where we disregard dependencies between characters and only consider the different character probabilities. The behavior of tries under this model has been thoroughly studied [60,61]. The expected average depth of an uncompressed trie is \( \Theta(\log n) \). In this section we show that the expected average depth of an LC-trie is \( \Theta(\log \log n) \), a significant improvement over uncompressed tries.

We consider independent binary strings from a Bernoulli-type process where each string is a sequence of independent identical random variables with values from an alphabet \( \Sigma = \{0, 1, \ldots, m-1\} \).

**Theorem 3.12:** For an independent random sample from a Bernoulli-type process with character probabilities \( p(s) > 0 \), \( s \in \Sigma \), not all equal, the expected average depth of an LC-trie \( E(A_n^{level}) = \Theta(\log \log n) \). If the probabilities are equal \( E(A_n^{level}) = \Theta(\log^* n) \).

**Proof:** If all characters have the same probability the strings will be uniformly distributed and Theorem 3.10 implies that \( E(A_n^{level}) = \Theta(\log^* n) \). Now consider the case \( \alpha = \min_{s \in \Sigma} p(s) < 1/|\Sigma| \). Let \( k = -\log n / \log \alpha \), and let \( q \) be a number such that \( 0 < q < 1 \). Without loss of generality we can assume that \( p(0) = \alpha \). We will give a lower bound for the probability \( P_{all}[n] \) that all possible strings of length \( qk \) will occur at least once as a prefix among the \( n \) strings. It can be shown that \( P_{all}[n] \) is larger than the probability that the string consisting of \( qk \) zeroes occurs at least \( 2^{qk} \) times among the prefixes of the strings. In fact, to prove this we may assume without loss of generality that the probabilities of the prefixes are equal.
Now consider all possible sequences of length \( n \) of prefixes of length \( l \).
We just need to check that the number of different sequences that contain at
least \( 2^n \) prefixes consisting of only zeros is not greater than the number
of sequences that contain at least one copy of each possible prefix. This is a
straightforward exercise in combinatorics; the details are omitted. Let \( A \) be
the number of prefixes consisting of \( l \) zeros among the \( n \) input strings.

\[
P_{all}(n) \geq P(A \geq 2^n) = P(A \geq 2 - q^2 - \frac{q}{\log \alpha}) = P(A \geq n - (1 + q^2 \log \alpha - 1)) n! - q).
\]

A obeys the binomial distribution \( B(n, \alpha^q) = B(n, n^{-q}) \) and, in particular,
\( E(A) = n^{-q} \). Using the Chernoff bound [56, Theorem 4.2]

\[
P(A \geq (1 - \epsilon)E(A)) \geq 1 - e^{-\epsilon^2 E(A) / 2}
\]

we see that \( P_{all}(n) \to 1 \) as \( n \to \infty \) if \( 1 + q^2 \log \alpha - 1 > 0 \), or equivalently
\( q < \frac{\log \alpha}{\log \alpha - 1} \). In particular, \( P_{all}(n) \to 1 \) for all \( \alpha \) such that \( 0 < \alpha < 1 / 2 \).

Clearly \( E(A_{n}^{level}) \leq E(A_{n}^{ave}) \), but \( E(A_{n}^{ave}) = O(\log n) \) [61] and hence
we get the following recursion inequality for \( D(n) = E(A_{n}^{level}) \).

\[
D(n) \leq 1 + P_{all}(n) \cdot D(n/2^n) + (1 - P_{all}(n)) \cdot O(\log n).
\]

It is easy to check that for \( q = 1/4, (1 - P_{all}(n)) \log n \to 0 \) as \( n \to \infty \) and hence

\[
D(n) \leq O(1) + D(n/2^n) = O(1) + D(n^{1 + q / \log \alpha}).
\]

But \( -1/4 < q / \log \alpha < 0 \) and hence \( E(A_{n}^{level}) = \Omega(\log \log n) \).

The lower bound can be established in a similar way. Choose \( \alpha \) as above
and let \( Q \) be a number such that \( 1 < Q < - \log \alpha \). Denote by \( P_{none}(n) \) the
probability that there will be no string with a prefix consisting of \( l \) zeros
among the \( n \) strings. Assuming that \( \min_{s \in \Sigma} p(s) < 1/|E| \) we make the
following two observations, \( P_{none}(n) = (1 - (1/2))^{-n} = (1 - 1/2)^{-n} \to 1 \) as
n \( \to \infty \) and \( n/2^n = n^{1 + Q / \log \alpha} \), where \( -1 < Q / \log \alpha < 0 \). We have the
inequality

\[
D(n) \geq 1 + (1 - P_{none}(n)) \cdot 0 + P_{none}(n) \cdot D\left(\frac{n}{2^n}\right)
\]

and hence \( E(A_{n}^{level}) = \Omega(\log \log n) \). \( \square \)

Observe that the assumption that the probability of each character is
positive is necessary. For example, consider a ternary alphabet \( \{A, B, C\} \)
where \( p(A) = 0 \). No level compression will take place in this case since
every node in the trie will have at most two children.
Compressed strings

String searching in compressed text documents is a new and interesting problem [3,4]. In this section we argue that the LC-trie might be useful for this kind of applications. We show that under certain assumptions the LC-trie actually behaves better for compressed strings.

Most modern model-based compression schemes use either Huffman coding [45] or arithmetic coding [82] to code a message with respect to a statistical model of the text. Arithmetic coding is the more efficient of the two. The only drawback being that it is slightly more difficult to implement.

In arithmetic coding a message is represented by a subinterval of \([0, 1]\). Successive characters of the text reduce the size of the interval according to the character probability generated by the statistical model. Consider the following example.

**Example 3.13:** The text contains three different characters \(A, B, \) and \(C\) with fixed probabilities \(0.2, 0.3, \) and \(0.5, \) respectively. We want to encode the text \(CAB\). To each of the characters we assign an interval whose size is proportional to the probabilities. In this example the intervals \([0, 0.2), [0.2, 0.5), \) and \([0.5, 1)\) are assigned to the characters \(A, B, \) and \(C,\) respectively. When the first character \((C)\) has been processed the interval is reduced to \([0.5, 1)\), the second character \((A)\) reduces the interval to \([0.5, 0.6)\), and after the third character \((B)\) we get \([0.52, 0.55)\).

We immediately get the following theorem.

**Theorem 3.14:** For an independent random sample from a Bernoulli-type process where each string has been compressed by arithmetic coding the expected average depth of an LC-trie \(E[\overline{A}_{\text{level}}] = \Theta(\log^* n)\).

**Proof:** The strings from the Bernoulli-type process will be uniformly distributed after having been arithmetically coded and hence the result follows immediately from Theorem 3.10. In fact, consider any subinterval \(I\) of \([0, 1]\) and an arithmetically coded infinite string \(s\) from a Bernoulli-type process. It follows from the definition of arithmetic coding that \(P\{s \in I\}\) equals the size of \(I\). \(\square\)

Huffman coding also has the effect of smoothing the input, but to a lesser degree.
**Theorem 3.15:** For an independent random sample from a Bernoulli-type process where each string has been compressed by Huffman coding the expected average depth of an LC-trie $E(A_{\text{n[del]}}) = O(\log \log n)$.

**Proof:** In this case the distribution will not be uniform and the bits will not be independent. The probability of a bit being either 0 or 1 depends on the preceding bits. However, we can make the following observation. The variable length characters of the Huffman coded text correspond to paths from the root to a leaf in the Huffman tree and at each internal node $r$ of the Huffman tree there will be a positive probability $P_r(0)$ that the next bit occurring will be 0. In fact, $P_r(0) \geq \min_{s \in S} p(s)$, since $P_r(0)$ is the sum of the weights of the leaves in the left subtree of $r$. Similarly, $P_r(1) \geq \min_{s \in S} p(s)$. The theorem can now be proved using the same technique as in the proof of Theorem 3.12.

As a possible application we mention a large dictionary where each entry, starting with its key, is coded separately by Huffman coding. The dictionary, together with an LC-trie representing the coded keys, requires less space than the uncompressed document and the LC-trie supports fast location of keys. When a key is found, the appropriate entry could be decoded separately and presented to the user.

### 3.5 CONCLUSIONS

The LC-trie is interesting both from a theoretical and practical point of view. For an independent random sample with a density function that is bounded from above and below the expected average depth of an LC-trie is $O(\log^2 n)$, and for data from a Bernoulli-type process with character probabilities not all equal the expected average depth is $O(\log \log n)$. Uncompressed tries and path-compressed tries both have expected average depth $O(\log n)$ for these distributions.

Tries and suffix trees are the basic data structures of many string algorithms. And their cousin, the quadtree, is widely used in computer graphics, image processing, geographic information systems, and robotics. Hence, the results presented in this chapter have a wide range of possible applications. The practical aspects of LC-tries are further discussed in Chapter 5, where we compare the LC-trie with suffix arrays and other standard trie implementations. In particular, we show that an array implementation of an LC-trie combined with binary encoding, data compression,
3.5 CONCLUSIONS

path compression, and level compression can be used to build an efficient, compact, and fast implementation of a suffix tree.

A possible topic for future research is to see whether it is possible to use level compression in dynamic trie structures without wasting too much space. Another interesting project would be to use an LC-trie in a full-scale implementation of a large index.
4

Experiments on Sorting

There is only one way to know which sorting algorithm is the fastest in practice. You have to implement the algorithm and measure the running time. This may seem like a simple task, but there are many pitfalls. The results will be influenced by such factors as compilers, operating systems, and machine architecture. Also, the amount of work spent on implementing a particular algorithm is likely to affect its efficiency. Another problem is to decide what kind of data to use for the experiments.

Even so, real timings do give additional information that cannot be directly deduced from the RAM model. For example, the unit-cost RAM model does not account for the actual cost of basic operations and hidden costs for storage allocation and indexing. The time measurements are merely intended to illustrate the typical behavior of the algorithms on a number of different machines. The precise performance in a particular application will of course vary.

We present time measurements for some of our radix sorting algorithms and compare the results with the Quicksort implementation of Bentley and McIlroy [19] and the Radixsort implementations of McIlroy, Bostic, and McIlroy [54]. We claim that our algorithms are often faster and less sensitive to adverse data. In particular, there is a large difference between the best Quicksort and the best Radixsort implementations. For string sorting Radixsort is often two to three times faster than Quicksort.

4.1 STRING SORTING

The main design decision to be made when implementing a radix sorting algorithm is whether to represent the input by a linked list or an array. The array representation has the advantage that the permutation can be per-
formed in place, but we do not get a stable algorithm. Algorithms based on linked lists have the advantage of being simple to implement. Experiments by McIlroy, Bostic, and McIlroy [54] show that the list-based implementations are faster on some architectures and array-based implementations on others. Since our primary goal is to compare different algorithms and optimization schemes and not to find the fastest possible algorithm for a particular architecture we have chosen to implement all of our algorithms using linked lists.

Method

The code development has been done on a SPARC station ELC using the SUN cc compiler, the GNU gcc compiler, the gdb debugger, and the tcov line-count profiler. For the final measurements we have used a SPARC station 5 with 64 megabytes of internal memory with the cc and gcc compilers and a 66 MHz 486 PC with 16 Mbytes of internal memory with the gcc compiler and the Linux operating system. In all cases we have used optimization level 04. We have measured the total number of seconds of CPU time consumed by the kernel on behalf of the program. The standard deviation was low, but we did encounter occasional outliers. These atypical timings were, of course, always slower than the typical value. Therefore we have chosen to report the minimum value of five trials.

It is common to study the behavior of sorting algorithms for uniformly distributed data. But realistic sorting problems are usually far from random. Also, uniformly distributed keys can be sorted very efficiently using a simple bucket scheme. To get more realistic timings we have chosen to use samples of real text.

Experiments were performed on a collection of standard texts from the Calgary/Canterbury text compression corpus [17]. Since most of these texts are small we have concatenated all text files and added the standard Unix “words” file. In this way we get a file consisting of 2572220 bytes and 87 886 lines. To get ever larger files we concatenate this file with itself to produce six files of different sizes: the smallest has 87 886 lines and the largest has 527 316 lines.

It is much harder to find realistic worst-case data. To inspect the complete data space is of course unfeasible and we can never be completely sure that a particular input really gives rise to the worst possible performance. However, as an illustration we have used badly fragmented data in an attempt to make the radix sorting algorithms behave badly. The idea
is to force the algorithm to immediately split the strings into a large number of groups. These groups contain both small and large characters so that all buckets will have to be examined in each phase of the algorithm. Furthermore, the groups are constructed in such a way that only one element is eliminated during each bucket phase. Also, the groups are larger than the Insertion sort breakpoint. Below we see the first few lines of such a text from an alphabet \{A, B, \ldots, Z\}. In this particular example the Insertion sort breakpoint is 3.

```
AAZ
AAAZ
AAAAAZ
AAAAAAAZ
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
AAAAAAA
```

The following algorithms were examined.

**Forward radixsort:** This is the implementation of the basic Forward radixsort algorithm as presented in Chapter 6. The number of bits used for bucketing is chosen as a function of the number of elements remaining to be sorted. The minimum and maximum character is recorded in each bucket phase and Insertion sort is used to sort small groups. Furthermore, sublists of elements with a common character are moved as a unit.

**Adaptive radixsort:** This is the implementation of Adaptive radixsort as presented in Chapter 6. The algorithm uses only two different character sizes, 8 bits and 16 bits. In the 8 bit case the minimum and maximum character is recorded. In the 16 bit case a more elaborate book-keeping is performed. The algorithm moves sublists of elements with a common character as a unit. An explicit stack is used to keep track of the flow of computation. In this way we only need
to use one bucket array. The algorithm switches to Insertion sort for small groups.

Program C: Four different implementations of MSD radixsort are presented by McIlroy, Bostic, and McIlroy [54]. We have tried all of these, but only present the measurement for the fastest, which is referred to as Program C in the article. This is an array based algorithm that uses a fixed 8-bit alphabet and performs the permutations in place. It keeps track of the minimum and maximum character in each bucket phase. The algorithm switches to Insertion sort for small groups.

Quicksort: Comparison-based sorting algorithms have a long history of implementation and optimization. The Quicksort variant engineered by Bentley and McIlroy [19] seems to be the clear winner. The algorithm is constructed to work for all kinds of input and hence has some overhead. We therefore use a stripped down version that is specifically tailored for character strings.

Discussion

Several researchers have independently found that a carefully coded radix sorting algorithm often outperforms comparison-based algorithms by a large margin [28,54] and we come to the same conclusion. Some typical timing measurements are shown in Figure 4-1. These results were produced on a SPARC station 5 with the gcc compiler. Similar results were found for the other compilers and machines. The main differences were that the cc compiler produced very bad code for Quicksort, with about twice the running time of the code produced by the other compilers. The PC favors array-based algorithms over linked lists and hence Program C was the winner on this architecture.

Figure 4-2 shows the running time for the fragmented data described above. We see that Forward radixsort is the most robust algorithm. It never uses characters of less than 8 bits, but using the technique described in Section 2.4 it still avoids to inspect a large number of empty buckets. Program C is about twice as slow for fragmented data as for typical data. Adaptive radixsort is less vulnerable to fragmentation since it uses a small alphabet when the number of strings that remain in a group is small. However, using a small alphabet for a large portion of the sorting increases the running time.
Figure 4-1. String sorting. The number of seconds of CPU time consumed by the kernel as a function of the number of keys.
Figure 4-2. String sorting. Fragmented data. The number of seconds of CPU time consumed by the kernel as a function of the number of keys.

Observe that Quicksort is much slower than the radix sorting algorithms, even for data specifically manufactured to demonstrate bad performance for these algorithms. Also, the worst-case cost of Quicksort, even though it is extremely unlikely, is quadratic.

One of the advantages of Forward radixsort is that it reduces the amount of bucketing. In a more realistic implementation the cost associated with each bucket operation will be larger than in our idealized example. The alphabetic order of the characters might be different from the numerical order and there may be several character codes that should end up in the same bucket. For example, the characters 'E', 'e', and 'é' are typically treated as equivalent for the purpose of alphabetic sorting. For the recently proposed 16-bit Unicode character set [73] these factors might be of considerable importance and Forward radixsort might well be the most feasible radix sorting algorithm in this case.

There are many design decisions to be made when implementing a radix sorting algorithm. For example, how should we choose the alphabet size? A theoretically attractive scheme is to choose the number of bits proportional to the logarithm of the remaining number of elements. Another approach is to restrict the alphabet to certain fixed sizes. On a 32-bit
machine it is natural to extract a number of bits that is a multiple of 8. In this way we avoid potential problems with word alignment. A priori it is not clear which of these approaches is the best. In fact, for Forward radixsort the free-choice method was best and for Adaptive radixsort 8-bit alignment was slightly better, but the differences were small.

The single most important optimization is to switch to a simple comparison-based algorithm when only a small number of elements remain to be sorted. Bentley and McIlroy [19] have investigated several different sorting subroutines and have found that Insertion sort is best suited. The exact value of the breaking point does not seem to be crucial. Values in the range 10 to 30 often give satisfactory results. For Forward radixsort the running time is reduced by about 40% and for Adaptive radixsort the reduction is almost 50%.

Since a large part of the total running time is spent inside the Insertion sort routine it is important to implement this routine carefully. We have taken advantage of the fact that in MSD radix sorting algorithms the strings to be compared are known to have a common prefix and hence there is no need to compare the strings starting at the first position. For highly repetitious data this optimization reduces the overall running time by as much as 10%.

One of the biggest problems to overcome in the implementation of Forward radixsort was the large space overhead. However, we found a simple and very efficient solution to this problem. The idea, which is presented in some detail in Section 2.4, is to use only one tag per group, instead of one tag per element. This idea alone reduces the total number of tags to less than 10% of the number of elements. Also, the running time decreases by roughly 10%.

But we can do even better if we are more restrictive with the splitting and do not split consecutive groups that are finished. This simple optimization gives a drastic reduction in space complexity. In fact, the total number of group records that are allocated is typically only 6% of the total number of elements. This optimization also reduces the running time with another 4%. The same technique has been used to minimize the explicit run-time stack used by Adaptive radixsort.

With these two optimizations the space complexity of Forward radixsort is no longer a problem. In our implementation a group record consists of 6 words and a bucket record consists of 5 words. In addition to the space needed to represent the pointers of the linked list Forward radixsort uses less than 0.8 extra words per element for typical data.
4.2 INTEGER SORTING

The \(O(n \log \log n)\) worst-case time integer sorting algorithm from Chapter 2 turns out to be surprisingly fast in practice. We have implemented the algorithm using the range reduction scheme of Kirkpatrick and Reisch [49]. The packed sorting is implemented as described in Chapter 2. The straightforward (and unoptimized) code is presented in Chapter 6.

Method

To give an indication of the performance of the new algorithm we have compared with Adaptive radixsort and some well known comparison-based algorithms. Heapsort and Mergesort are both guaranteed to run in \(O(n \log n)\) time, while Quicksort and Adaptive radixsort are known to be fast in practice.

**Heapsort:** An efficient implementation of Heapsort by Carlsson [22]. The original Pascal code has been translated into C and all procedures have been inlined.

**Mergesort:** A slightly modified version of the Mergesort by Gonnet and Baeza-Yates [42]. The algorithm sorts short sublists with Insertion sort. This optimization improves the running time of the original algorithm by about 15%.

**Quicksort:** The Quicksort algorithm by Bentley and McIlroy [19] discussed in the previous section, now adapted for integers.

**Adaptive radixsort:** The algorithm uses \(\log n - 2\) buckets to sort \(n\) numbers and switches to Insertion sort when less than 25 elements remain.

The measurements of Figure 4-3 were produced on a SPARC station 5 with the gcc compiler. The data consists of uniformly distributed integers from the set \([0, 1, \ldots, 2^{31} - 1]\) generated by the “minimum standard” 32-bit random number generator described by Park and Miller [58]. We have also made experiments on integers consisting of 4 consecutive 8-bit ASCII characters from the text used in the string sorting experiments. The results were virtually identical to those for random numbers.
Figure 4-3. Integer sorting. The number of seconds of CPU time consumed by the kernel as a function of the number of keys. Worst-case efficient algorithms are indicated by solid lines and average-case efficient algorithms by dashed lines.
4.3 CONCLUSIONS

We have investigated the performance of a number of string sorting algorithms. It is evident that radix sorting algorithms are much faster than the more frequently used comparison-based algorithms. On the average, Adaptive radix sort is the fastest algorithm in our experiments. However, Forward radix sort is only slightly slower and has a guaranteed worst-case
behavior. Furthermore, Forward radixsort is better suited for large alphabets such as the recently proposed Unicode 16-bit character set [73].

Out of curiosity we implemented the new sorting algorithm with worst-case time complexity $O(n \log \log n)$. To our surprise it turned out that this algorithm is very fast also in practice. Already on a 32-bit machine (SPARC station 5) it outperforms some of the most widely used comparison-based algorithms. Indeed, in our experiments it was the fastest among the worst-case efficient algorithms.

It is nice to see that theory works in practice.
5

Experiments on Searching

To study the real world behavior of the various trie structures discussed in Chapter 3, we have carried out a number of experiments. We have chosen not to make experiments on a specific architecture using a specific programming language. Instead, we have performed simulations, measuring critical operations.

In Section 5.1 we measure the size and depth of different trie structures and argue that level compression reduces both the overall size and the depth of a trie considerably. In Section 5.2 we show how to use an LC-trie to implement a suffix tree index into a large text in secondary memory. Using the LC-trie as an internal data structure we are able to reduce the number of external memory accesses while at the same time reducing the size of the internal structure.

5.1 Searching in Main Memory

Method

To estimate space requirements, we let characters and branching values occupy 1 byte each; small integers, such as skip values, occupy 2 bytes each; and pointers occupy 4 bytes. In order to measure search costs, we have counted the average number of steps required for a successful search. As steps we count the number of traversed nodes in a trie and the number of string comparisons made in a suffix array. Experiments were performed on the following data.

Random text: A text consisting of zeroes and ones, independently chosen with equal probability.
**DNA:** The complete DNA sequence of the Epstein-Barr virus.

**FAQ, ASCII:** An English text in ASCII format, the Frequently Asked Questions list for the newsgroup comp.windows.x (2 Sep 91).

**FAQ, Huffman:** The same text as above, but Huffman coded [45].

Each suffix of the texts were used as a key. In each case, we performed experiments on 1%, 10%, and 100% of the input. The following data structures were examined. A more detailed description can be found in Section 3.3.

**Augmented suffix array:** The worst-case efficient data structure presented by Manber and Myers [52]. Each entry in the array contains one address (4 bytes) and two skip values (2 bytes each). Hence, an augmented suffix array of length $n$ occupies $8n$ bytes.

**Trie, linked list implementation:** An alphabetic trie where the outgoing pointers are stored in a linked list at each internal node. Path compression is used. An internal node of degree $d$ is represented by $d$ list nodes. Each such node contains one character, one skip value, and two pointers, a total of 11 bytes. External nodes are marked by a specially designated skip value and contain one pointer into the document, requiring 6 bytes per node. For $n$ leaves and $I$ internal nodes the total space is $11I + 6n$.

**Trie, array implementation:** An alphabetic trie where the pointers are implemented with an array of length $m$ equal to the size of the alphabet at each internal node. Path compression is used. Each internal node uses $4m$ bytes for pointers and 2 bytes for the skip value. External nodes are represented as above. For $n$ leaves and $I$ internal nodes the total space is $(4m + 2)I + 6n$.

**Patricia tree:** A binary trie implemented as a Patricia tree. The trie is stored in an array and each node is represented by a skip value and an address, requiring 6 bytes. The total number of nodes in a Patricia tree is $2n - 1$ and hence the space required is $6(2n - 1)$.

**LC-trie:** We implement the trie in the space efficient way described in Section 3.3, where each node is represented by two integers, one short integer that either holds a skip or a branch value and one long integer that holds a pointer, a total of 6 bytes per node. An LC-trie with $n$ leaves and $I$ internal nodes requires $6(n + 1)$ bytes.
Not all combinations of input texts and data structures make sense; in an
alphabetical trie it makes no difference how the characters have been en-
coded; for binary strings the alphabetical trie and the binary trie coincide.

We also performed experiments on a trie where each internal node was
represented as a binary search tree. This structure achieved roughly the
same average depth as a Patricia tree, but at the cost of using considerably
more space. These results are not included here.

Discussion

The simulation results are presented in Table 5-1. Starting with the space
requirements, we observe that among the various trie structures the LC-trie
requires the smallest amount of space for all kinds of input. The only data
structure that uses less space than the LC-trie is the suffix array.

The DNA sequence has a very even character distribution and hence the
results for this input are very similar to the results for random data. In both
cases, the average depth in an LC-trie is considerably smaller than in any of
the other trie structures. As expected, it is also smaller than the number of
binary search steps required in a suffix array. Recall that the average depth
of an LC-trie is $O(\log n)$ for independent random data.

For English text the LC-trie has a markedly better performance than an
ordinary trie. The array implementation achieves a smaller average depth
at the price of a prohibitively large space overhead.

Texts are often represented using ASCII-code, where each character con-
sists of an 8 bit integer. In many applications only a few of the 256 possible
characters are actually used. For example, English text typically uses less
than a hundred characters. The superfluous bits in the ASCII-code will in-
crease the search time in the binary trie. To counter this potential problem
we have used Huffman coding [45] to compress the text. Huffman coding
has the advantage of generating more evenly distributed binary strings
and we can expect a well balanced LC-trie with high branching factors.
In fact, as shown in Theorem 3.15, the expected average depth of an LC-
trie with Huffman coded data from a Bemoulli-type process is $O(\log \log n)$. Observe that it is possible to use the Huffman coded version of the charac-
ters within the trie even if the text is stored in a different format.

We see that the search cost in an LC-trie is significantly reduced for Huff-
aman coded text. Using Huffman coding on a binary trie also gives a slight
improvement. However, it is the combination of Huffman coding and level
compression that makes the difference. Huffman coding smooths the text
<table>
<thead>
<tr>
<th>File</th>
<th>Suffix Array</th>
<th>Trie (list)</th>
<th>Trie (array)</th>
<th>Patricia Tree</th>
<th>LC-trie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Steps</td>
<td>Size</td>
<td>Depth</td>
<td>Size</td>
</tr>
<tr>
<td>Random text</td>
<td>2</td>
<td>10.0</td>
<td>16</td>
<td>12.4</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>13.3</td>
<td>160</td>
<td>15.6</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>16.6</td>
<td>1600</td>
<td>18.9</td>
<td>2400</td>
</tr>
<tr>
<td>DNA</td>
<td>1.7</td>
<td>9.8</td>
<td>14</td>
<td>12.4</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>13</td>
<td>138</td>
<td>15.9</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>172</td>
<td>16.4</td>
<td>1378</td>
<td>20.3</td>
<td>2067</td>
</tr>
<tr>
<td>FAQ, ASCII</td>
<td>1.9</td>
<td>9.9</td>
<td>15</td>
<td>15.5</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>13.2</td>
<td>155</td>
<td>21.6</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td>193</td>
<td>16.6</td>
<td>1545</td>
<td>20.2</td>
<td>2318</td>
</tr>
<tr>
<td>FAQ, Huffman</td>
<td>1.9</td>
<td>9.9</td>
<td>15</td>
<td>7.2</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>13.2</td>
<td>155</td>
<td>9.9</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>193</td>
<td>16.6</td>
<td>1545</td>
<td>13.1</td>
<td>2196</td>
</tr>
</tbody>
</table>

Table 5-1: Empirical results for suffix array and suffix trees. Sizes are measured in kbytes.
so that level compression becomes more effective.

One might argue that the branching operation in an LC-trie is more expensive than in a Patricia tree: instead of using just one bit for branching at each node we use a varying number of bits. However, the following argument shows that the total search cost in an LC-trie is, in fact, smaller. First, we note that during a search in an LC-trie we examine exactly the same bits in the query string as during a search in a Patricia tree. The difference is that in the LC-trie consecutive bits are sometimes examined in groups, while they are examined one at a time in a Patricia tree. For example, if a node in an LC-trie uses 4 bits for branching, we extract these 4 bits from the query string and treat them as a 4-digit binary number. Depending on the value of this number we determine in which subtree the search is to be continued. In the corresponding Patricia tree we extract the same four bits one at a time, traversing 4 nodes in the tree.

In summary, with respect to space and average depth, the LC-trie is a clear winner among the various trie structures. The only real advantage of a suffix array is the fact that it uses slightly less memory than the trie structure. However, a suffix array, as opposed to a trie, requires access to the document itself at each step of the binary search.

5.2 SEARCHING IN SECONDARY MEMORY

Many applications need to handle large texts that do not fit in main memory. To reduce the number of time-consuming accesses to secondary memory a partial data structure that fits in main memory could be employed. We suggest a partial LC-trie stored in main memory used as an index into a suffix array stored in secondary memory. The partial LC-trie is constructed in the following way. As soon as we reach a node that covers less than k strings, where k is a constant, we stop the tree building algorithm and add a pointer into the suffix array. The number k is called the cutoff value.

We have compared this data structure with a simpler alternative suggested by Manber and Myers [52], where the internal data structure merely consists of a bucket array.

Method

We have studied the following two data structures.

**Suffix array/bucket array:** In all experiments, we use approximately \( n/4 \) buckets, as suggested by Manber and Myers [52]. We have cho-
sen the number of buckets in such a way that the proper bucket can be computed by truncating a number of bits. Thus, in most cases (the exception being the k-digit coding) we have chosen the number of buckets to be a power of 2. Each bucket is represented by a pointer. The total space for b buckets is 4b.

**Suffix array/LC-trie:** The cutoff value (the maximal number of elements stored in a leaf in the LC-trie) has been chosen in such a way that both the time and space measurements of the two data structures can be easily compared, that is, both measurements will be better for one structure than for the other.

We have ignored the time spent in the internal structure and only counted the number of accesses to secondary memory required during the binary search, since these are typically much more expensive. Also, we have only measured space requirements for the internal structures.

Another natural way to try to improve the behavior of the bucketing method is to use a better encoding. For the sake of completeness we have tried both the 96-digit coding suggested by Manber and Myers and a combination of bucketing and Huffman coding. Both methods enhance the performance slightly but do not come close to the performance of the LC-trie.

In addition to the texts used in the previous section we have used some standard texts from the Calgary/Canterbury text compression corpus [17]. The following additional texts were used.

- **bib:** Bibliographic files ("ref" format).
- **trans:** Transcript of a terminal session.

**Discussion**

Looking at Table 5-2 we observe that for almost all texts the LC-trie has both a lower average number of accesses to secondary memory and a much smaller size. The only exception is the random string and the DNA sequence, where the bucket array is slightly better. This is to be expected since bucketing is very efficient for uniformly distributed data.

Our experimental results accentuate some of the problems of using a plain bucketing scheme. The major drawback is that for nonuniform distributions many elements may end up in a few buckets while many buckets remain empty. This will not only be space inefficient, but also the accumulation of elements will increase the number of accesses to secondary
Table 5.2. Empirical results for partial data structures. Sizes are measured in kbytes.

memory. The LC-trie/suffix array implementation has the advantage that the number of elements represented by a leaf is bounded. In our experiments no cutoff value is greater than 100. However, when using the bucket array/suffix array on the text "FAQ" the largest bucket contained no less than 9198 elements and hence the maximum search cost is much higher for the bucket array.

5.3  CONCLUSIONS

Our experiments indicate that the theoretical advantages of LC-tries are reflected also for real data and we see that level compression reduces both the overall size and the depth of a trie considerably. A suffix array uses slightly less memory but, as opposed to a trie, requires access to the document itself at each step of the binary search.

As a possible application we have used an LC-trie to implement a suffix tree index into a large text in secondary memory. The combination of path compression, level compression, and data compression gives an efficient
data structure. Both the size of the structure and the average and worst-case number of accesses to secondary memory is smaller than for a plain bucketing scheme.

It is nice to see that theory works in practice.
6

Implementation

CAREFUL CODING is important if we want to achieve highly efficient programs. In this chapter we present C-code [47] for Adaptive radixsort and the basic version of Forward radixsort. We also give a complete implementation of the $O(n \log \log n)$ worst-case time integer sorting algorithm.

The strings to be sorted are represented as pointers to bit strings. Each string starts on a word boundary and the length of a string is stored explicitly.

```c
typedef unsigned int word;
typedef word *bitstring;
typedef int character;

typedef struct listrec *list;
struct listrec {
    bitstring str;
    int length;
    list next;
};
```

One of the basic operations of a radix sorting algorithm is to extract a number of bits to be used as a character. The efficiency of this operation has a large impact on overall performance. We therefore choose to implement it using macros. To gain efficiency we implement the arithmetics using bitwise operators instead of the standard division and modulus operators. There are two factors to consider: the requested bit pattern might cross a word boundary and it might overflow the end of the string. The top level macro GETCHAR checks if the requested pattern lies outside of the string and pads the answer with an appropriate number of zeroes. The BITS macro extracts the actual bits; it distinguishes between the case were the pattern
can be found within only one word and the case were the pattern resides in two consecutive words.

```c
#define W 32  /* word length */
#define DIVW(x) ((x) >> 5)  /* x div W */
#define MODW(x) ((x) & 31)  /* x mod W */
#define MINW(x) ((W - x)

#define BITS(str, word, left, right) \  
    (str[word] << left) >> right | \  
        (left > right ? str[word+1] >> (W + right - left) : 0))
#define GET(str, pos, bits) \  
    (BITS(str, DIVW(pos), MODW(pos), MINW(bits)))
#define GETCHAR(str, len, pos, bits) \  
    (pos + bits <= len ? GET(str, pos, bits) : \  
        GET(str, pos, len - pos) << (pos + bits - len))
```

Memory allocation is another crucial operation and a large number of calls to the system memory allocation routines might be costly. We therefore use the following simple memory manager. The function call `initmem(m, elemsize, blocksize)` initializes a memory allocator that can allocate at most `blocksize * MAXBLOCKS` slots of memory each consisting of `elemsize` bytes. The memory manager will request space from the operating system (using `malloc`) only after `blocksize` successive calls to `allocmem`. The function `deallocmem` frees the space obtained by the previous call to `allocmem`. Space is freed in last-in first-out order. All space obtained so far can be freed by a call to `resetmem`. Finally, a call to `freemem` does the necessary clean-up and returns the memory to the operating system (using `free`). Only the declarations are given here.

```c
#define MAXBLOCKS 100
typeof struct {
    void *block[MAXBLOCKS];
    int allocnr;
    int nr;
    int blocksize;
    void *current, *first, *last;
} memory;

void initmem(memory *m, int elemsize, int blocksize);
void allocmem(memory *m, int elemsize);
void deallocmem(memory *m, int elemsize);
void resetmem(memory *m);
void freemem(memory *m);
```
Finally, we define a boolean data type and two utility routines for computing the maximum and minimum of two elements.

```c
typedef int boolean;
#define TRUE 1
#define FALSE 0
#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

**Insertion Sort**

Both string sorting algorithms presented in this chapter use the strategy of switching to a simple comparison based algorithm when only a small number of strings remain in a group and therefore we need a routine for comparing bit strings.

The routine takes advantage of the fact that the strings to be compared will have a common prefix when the radix sorting algorithms call the comparison based sorting algorithm. Hence, the comparison routine compares the strings starting at a specified position p.

```c
int bitcmp(list a, list b, int p)
{
    const bitlen = MIN(a->length, b->length);
    const firstword = DIVW(p); /* p / W */
    const lastword = DIVW(bitlen); /* bitlen / W */
    bitsstring astr, bstr;
    int i;

    astr = a->str + firstword;
    bstr = b->str + firstword;

    i = firstword;
    while (i < lastword && *astr == *bstr)
    { i++, astr++, bstr++ }

    if (*astr < *bstr)
        return -1;
    else if (*astr > *bstr)
        return 1;
    else
        return a->length - b->length;
}
```

The Insertion sort algorithm is straightforward. String comparison is performed starting at position p of each string. The algorithm returns the head and the tail of the sorted list.
list Insertsort(list r, list *tail, int p)  
{  
list fi, la, t;  
  
for (fi = la = r, r = r->next; r = la->next)  
  
if (strncmp(r, la, p) >= 0) /* add to tail */  
    la = r;  
else if (strncmp(r, fi, p) <= 0) { /* add to head */  
    r->next = r->next;  
    fi = r;  
  } else { /* insert into middle */  
    for (t = fi; strncmp(r, t->next, p) >= 0; )  
      t = t->next;  
    r->next = r->next;  
    t->next = fi;  
  }  
  *tail = la;  
  return fi;  
}

6.1 FORWARD RADIXSORT

This is an implementation of the basic Forward radixsort algorithm from Section 2.4. A more detailed description of the group and bucket data structures can be found there.

The group structure member insp is used to make splitting of groups possible during the phase where elements are moved from buckets back into their previous groups. The group structure member finis indicates if the elements in a group are sorted; this information makes it easy to skip finished groups during a traversal of the group data structure.

#define INSERT BREAK 20  
#define MAXBITS 16  
#define MAXBuckets 20*MAXBITS

typedef struct grouprec *group;  
typedef struct bucketrec *bucket;

struct grouprec {  
    list head, tail; /* a list of elements */  
    group next; /* the next group */  
    group nextund; /* the next unfinished group */  
    group insp; /* insertion point */  
    boolean finis; /* is the group finished? */  
};
struct bucketrec {
  list head, tail;  /* a list of elements */
  int size;        /* list length */
  group tag;       /* group tag */
  bucket next;     /* next bucket item */
};

static memory groupmem[1];
static memory bucketmem[1];

When inserting a list of elements into a bucket we distinguish between two cases. If the first bucket item has the same tag as the list to be inserted the list is just appended, otherwise a new bucket item is created.

static void intobucket(bucket *b, list head, list tail,
         int size, group g)
{
  bucket btemp = *b, newb;

  if (!btemp || btemp->tag != g) { /* create new tag */
    newb = (bucket)
      allocmem(bucketmem, sizeof(struct bucketrec));
    newb->next = btemp;
    newb->head = head;
    newb->size = size;
    newb->tag = g;
    *b = btemp = newb;
  } else { /* append */
    btemp->tail->next = head;
    btemp->size += size;
  }

  tail->next = NULL;
  btemp->tail = tail;
}

The next function traverses the groups and puts the elements into buckets. The parameter pos indicates the current position in the strings and bits is the number of bits used for bucketing. The minimum and maximum character encountered is recorded in chmin and chmax, respectively. To be able to skip groups that are already finished the previous group is kept track of. Also, the previously read character is recorded. In this way it is possible to move the elements in blocks consisting of strings that have a common character in position pos. Furthermore, a group that is not split during this phase is left behind and not put into a bucket.
static int intobuckets(group g, bucket b[], int pos, int bits,
    character *chmin, character *chmax)
{
    group prevg;
    character ch, prevch;
    const character endofstr = 1<<bits;
    *chmin = endofstr - 1;
    *chmax = 0;
    boolean split;
    list tail, tailn;
    int size, count = 0;

    resetmem(bucketmem);
    for (prevg = g; g = g->nextunf; g = g->nextunf) {
        if (g->finis) /* skip group g */
            {prevg->nextunf = g->nextunf; continue;}
        tail = g->head;
        split = FALSE;
        prevch = tail->length > pos ?
            GETCHAR(tail->str, tail->length, pos, bits) :
            endofstr;
        count++;
        size = 1;

        /* traverse an element list */
        for ( ; tailn = tail->next; tail = tailn) {
            ch = tailn->length > pos ?
                GETCHAR(tailn->str, tailn->length, pos, bits) :
                endofstr;
            count++;
            size++;
            if (ch == prevch) continue;
            intobucket(b+prevch, g->head, tail, size-1, g);
            if (prevch != endofstr) {
                if (prevch < *chmin) *chmin = prevch;
                if (prevch > *chmax) *chmax = prevch;
            }
            g->head = tailn; split = TRUE;
            prevch = ch; size = 1;
        }

        if (split) { /* no bucketing if characters equal */
            intobucket(b+prevch, g->head, tail, size, g);
            if (prevch != endofstr) {
                if (prevch < *chmin) *chmin = prevch;
                if (prevch > *chmax) *chmax = prevch;
            }
            g->head = NULL;
            prevg = g;
        }
6.1 FORWARD RADIXSORT

```c
} else if (prevch == endofstr) /* skip group */
    prevg->nextunf = g->nextunf;
else
    prevg = g;
}
return count;
```

The next function puts a list of elements back into the group structure and splits the old groups. If two consecutive groups are both finished, there is no need to perform any splitting.

```c
static void intogroup(group g, list head, list tail,
                      boolean finis)
{
    group newg;

    if (!g->head) { /* back into old group */
        g->head = head;
        g->tail = tail;
        g->finis = finis;
        g->insp = g;
    } else if (finis && g->insp->finis) {
        g->insp->tail->next = head; /* don't split if both */
        g->insp->tail = tail;       /* groups are finished */
    } else { /* split */
        newg = (group)
            allocmem(groupmem, sizeof(struct grouprec));
        newg->head = head;
        newg->tail = tail;
        newg->next = g->insp->next;
        newg->nextunf = g->insp->nextunf;
        newg->finis = finis;
        g->insp = g->insp->nextunf = g->insp->next = newg;
    }
}
```

This function traverses the bucket array and puts the elements back into their groups. The groups are split and finished groups are marked.

```c
static void intogroups(bucket b[], int pos, int bits,
                       character chmin, character chmax)
{
    const character endofstr = 1<<bits;
    bucket s, *bp;
    boolean finis;
```
Chapter 6: IMPLEMENTATION

if (b[endofstr]) {
    for (s = b[endofstr]; s; s = s->next)
        intgroup(s->tag, s->head, s->tail, TRUE);
    b[endofstr] = NULL;
}

for (bp = b + chmin; bp <= b + chmax; bp++) {
    if (!(*bp)) continue;
    for (s = *bp; s; s = s->next) {
        finis = FALSE;
        if (s->size < INSERT_BREAK) {
            if (s->size > 1)
                s->head = insertSort(s->head, &s->tail, pos);
            finis = TRUE;
        }
        intgroup(s->tag, s->head, s->tail, finis);
    }
    *bp = NULL;
}

This is a straightforward function that traverses the finished groups and returns the elements in sorted order:

static list collect(group g)
{
    list head, tail;
    g = g->next;
    head = g->head;
    tail = g->tail;
    for (g = g->next; g; g = g->next) {
        tail->next = g->head;
        tail = g->tail;
    }
    return head;
}

This function implements a simple heuristic to choose an appropriate number of bits for bucketing when n elements remain in a group.

static int choosebits(int n)
{
    if (n < 500) return 8;
    else if (n < 1000) return 10;
    else if (n < 2000) return 11;
    else if (n < 4000) return 12;
    else if (n < 8000) return 13;
else if (n < 16000) return 14;
else if (n < 32000) return 15;
else return 16;
}

The main function is straightforward. We use a dummy group g as the header of the group data structure. It does not contain any elements, but only a pointer to the first unfinished group. No bucketing is performed in a pass where all characters are equal. This implies that the worst-case time is O(S + n), where S is the total number of distinguishing 8-bit characters, as explained in the analysis of Forward radixsort in Chapter 2.

```c
list FRs(list t, int n)
{
    static bucket b[MAXBUCKETS+1];  
    // buckets */
    group g, g2;                   
    // groups */
    int pos = 0;                   
    // pos in string */
    int count;
    int bits;
    character chmin, chmax;
    if (n<2) return t;

    initmem(groupmem, sizeof(struct grouprec), n/15);
    initmem(bucketmem, sizeof(struct bucketrec), n/5);

    g = (group) allocmem(groupmem, sizeof(struct grouprec));
    g2 = (group) allocmem(groupmem, sizeof(struct grouprec));
    g->next = g->nextunf = g2;
    g2->head = t;
    g2->next = g2->nextunf = NULL;
    g2->finis = FALSE;

    bits = choosebits(n);
    count = intobuckets(g, b, pos, bits, &chmin, &chmax);
    while (g->nextunf) {
        pos += bits;
        intgroups(b, pos, bits, chmin, chmax);
        bits = choosebits(count);
        count = intobuckets(g, b, pos, bits, &chmin, &chmax);
    }
    t = collect(g);

    freemem(bucketmem);
    freemem(groupmem);

    return t;
}
```
6.2 ADAPTIVE RADIXSORT

This is an implementation of Adaptive radixsort as presented in Section 2.3. For this algorithm we have used another string representation. The alphabet consists of 8-bit characters and each string ends with the null character '\0'.

```c
typedef int character;
typedef unsigned char char_t;
typedef char_t *string;

#define CHARSMINUS 256
#define CHAR(s, p) s[p]

typedef struct listrec *list;
struct listrec {
    string str;
    list next;
};
```

Since we use a new string representation we need to make a slight modification of the Insertion sort routine. each call to `bcrypt` is replaced by a call to the standard string comparison routine `strcmp`.

In this implementation we use two characters (16 bits) for bucketing when more than 1500 strings remain, otherwise we use only one character. The macro `SHORT` extracts two consecutive characters.

```c
#define INSERT_BREAK 12
#define BYTE_BREAK 1500

#define BUCKET_INDEX CHARSMINUS
#define CHAR(s, p) s[p]
#define SHORT(s, p) s[p] << 8 | (s[p] ? s[p+1] : 0)

#define IS_ENDMARK(ch) ((ch & 255) == 0)
#define HIGH(ch) ch >> 8
#define LOW(ch) ch & 255
```

```c
typedef struct bucketrec {
    list head, tail;
    int size; /* size of list, 0 if the list is sorted */
} bucket;

typedef struct stackrec {
    list head, tail;
    int size; /* size of list, 0 if the list is sorted */
    int pos; /* current position in string */
} stack;
```
static memory stackmem[1];
static stack *stackp;

static void push(list head, list tail, int size, int pos)
{
    stackp = (stack *)
    allocmem(stackmem, sizeof(struct stackrec));
    stackp->head = head;
    stackp->tail = tail;
    stackp->size = size;
    stackp->pos = pos;
}

static stack *pop()
{
    stack *temp;
    temp = stackp;
    stackp = (stack *)
    deallocmem(stackmem, sizeof(struct stackrec));
    return temp;
}

static stack *top() { return stackp; }
static boolean stackempty() { return (!stackp); }

The next function puts a list of elements into a bucket array of size 256. The
routine keeps track of the minimum and maximum nonzero characters.

static void intobucket(bucket *b,
    list h, list t, int size,
    character ch,
    character *chmin, character *chmax)
{
    if (!b->head) {
        b->head = h;
        b->tail = t;
        b->size = size;
        if (ch != '0'){
            if (ch < *chmin) *chmin = ch;
            if (ch > *chmax) *chmax = ch;
        }
    } else {
        b->tail->next = h;
        b->tail = t;
        b->size += size;
    }
}
For 2-byte bucketing the bookkeeping is more elaborate. We use two integer arrays, used1 and used2, to keep track of what characters occur in the first and second position in each pass.

```
static void intobucket2(bucket *b, list h, list t, int size,
    character ch, int *used1, int *used2)
{
    if (!b->head) {
        b->head = h;
        b->tail = t;
        b->size = size;
        used1[HIGH(ch)] = used2[LOW(ch)] = TRUE;
    } else {
        b->tail->next = h;
        b->tail = t;
        b->size += size;
    }
}
```

If the number of elements in a list is small we sort the list using Insertsort before pushing it onto the stack. Observe that this function call refers to the modified version of the algorithm, the one that uses `strcmp` instead of `bitcmp`.

If the top of the stack contains a list of elements that is already sorted and the elements to be put on the stack are also sorted there is no need to create a new stack record, we just append the elements to the list on the top of the stack.

```
static void ontostack(bucket *b, int pos)
{
    b->tail->next = NULL;
    if (b->size <= INSERT_BREAK) {
        if (b->size > 1)
            b->head = Insertsort(b->head, &b->tail, pos);
        b->size = 0;   /* finished */
    }
    else if (!b->size && !stackempty() && !top()->size) {
        top()->tail->next = b->head;
        top()->tail = b->tail;
    } else {
        push(b->head, b->tail, b->size, pos);
        b->size = 0;
    }
    b->head = NULL;
}
```
The following function traverses a list and puts the elements into buckets according to the character in position pos. The buckets are traversed in ascending order and the lists are pushed onto the stack.

```c
static void bucketing1(list a, int pos)
{
    static bucket b[CHARS];
    bucket *bp;
    character ch, prevch;
    character chmin = CHARS-1, chmax = 0;
    list t, tn;
    int size = 1;

    prevch = CHAR(a->str, pos); /* into buckets */
    for (t = a; tn = t->next; t = tn) {
        ch = CHAR(tn->str, pos); size++;
        if (ch == prevch) continue;
        intobucket1(b+prevch, a, t, size-1,
                    prevch, &chmin, &chmax);
        a = tn;
        prevch = ch; size = 1;
    }
    intobucket1(b+prevch, a, t, size,
                prevch, &chmin, &chmax);

    if (b->head) /* put onto stack */
        b->size = 0; /* finished */
        onestack(b, pos);
    }
    for (bp = b + chmin; bp <= b + chmax; bp++)
        if (bp->head) onestack(bp, pos+1);
}
```

The code for bucketing on two bytes is very similar to the 1-byte version presented above. The main difference is that we use the information in used1 and used2 to reduce the number of empty buckets that are inspected.

```c
static void bucketing2(list a, int pos)
{
    static bucket b[BUCKETS]; /* buckets */
    character ch, prevch;
    list t, tn;
    int size = 1;
    int used1[CHARS]; /* what buckets are used? */
    int used2[CHARS];
    int buckets1 = 0, buckets2 = 0;
    character ch1, ch2, high;
```
Chapter 6: IMPLEMENTATION

```c
for (ch = 0; ch < CHAR_SIZE; ch++)
    used1[ch] = used2[ch] = FALSE;

prevch = SHORT(a->str, pos); /* into buckets */
for (t = a; t->next; t = t->next) {
    ch = SHORT(t->str, pos); size++;
    if (ch == prevch) continue;
    int bucket2(b+prevch, a, t, size-1,
                 prevch, used1, used2);
    a = t;
    prevch = ch; size = 1;
} int bucket2(b+prevch, a, t, size,
             prevch, used1, used2);

for (ch = 0; ch < CHAR_SIZE; ch++) {
    if (used1[ch]) used1[buckets1++] = ch;
    if (used2[ch]) used2[buckets2++] = ch;
}

for (ch1 = 0; ch1 < buckets1; ch1++) { /* put onto stack */
    high = used1[ch1] << 8;
    for (ch2 = 0; ch2 < buckets2; ch2++) {
        ch = high | used2[ch2];
        if (b[ch].head) {
            if (is_endmark(ch) b[ch].size = 0; /* finished */
                ontostack(b[ch], pos+2);
        }
    }
}
```

The main function takes the number of elements in the input as its second parameter. For typical data the algorithm puts only a small number of records on the stack. Hence the blocksize of the memory allocator is kept pretty small (n/50). The main function also chooses the appropriate version of the bucketing function.

```c
static list MSD(list a, int n)
{
    list res = NULL;
    stack *s;

    if (n < 2) return a;

    initmem(stackmem, sizeof(struct stackrec), n/50);
    push(a, NULL, n, 0);
```
while (!stackempty()) {
    s = pop();
    if (!s->size) { /* finished */
        s->tail->next = res;
        res = s->head;
        continue;
    }
    if (s->size <= BYTE_BREAK)
        bucketing1(s->head, s->pos);
    else
        bucketing2(s->head, s->pos);
}
freeem(stackmem);
return res;

6.3 INTEGER SORTING IN NEARLY LINEAR TIME

Here we present a complete implementation of the $O(n \log \log n)$ worst-case time integer sorting algorithm from Chapter 2. The code is intended to illustrate the algorithm and to prove that it is possible to surpass the $\Omega(n \log n)$ lower bound for comparison-based sorting with a relatively simple program. No attempts have been made to optimize the code.

Packed Merging

The subroutine for merging short sequences that we implement in this section is due to Albers and Hagerup [2]. It is based on an idea by Paul and Simon [59] and a lemma by Batcher [15]. To state the lemma we need to introduce the concept of a bitonic sequence. A sequence is bitonic if it is the concatenation of a nondecreasing and a nonincreasing sequence, or if it can be obtained as the cyclic shift of such a sequence.

Lemma 6.1: Consider a bitonic sequence $x_1, x_2, \ldots, x_{2k}$ and the two sub-sequences

$$L = \min(x_1, x_{k+1}), \min(x_2, x_{k+2}), \ldots, \min(x_k, x_{2k})$$

and

$$R = \max(x_1, x_{k+1}), \max(x_2, x_{k+2}), \ldots, \max(x_k, x_{2k}).$$

The sequences $L$ and $R$ are bitonic, and each element of $L$ is smaller than or equal to each element of $R$. 
The lemma suggests a parallel algorithm for sorting a bitonic sequence of length \( k \), where \( k \) is a power of 2, and, in particular, an algorithm for merging two sorted sequences.

- If \( k = 1 \) halt.
- If \( k > 1 \) compare the corresponding elements of the left and right half in parallel and generate the sequences \( L \) and \( R \), then sort these bitonic sequences in parallel.

This algorithm clearly runs in \( \mathcal{O}(\log k) \) time.

Assume that the word length \( w \geq 2k(b + 1) \), where \( b \) is the length of an integer and \( k \) is the number of elements of each sequence to be merged. Also, for simplicity, assume that \( k \) is a power of 2. In this way we will be able to fit \( 2k \) bit fields of size \( b + 1 \) into a machine word. The rightmost \( k \) bits will contain a \( b \)-bit element and the leftmost bit will be used as a test bit. We illustrate the algorithm by a simple example were we show to merge the sequences \( X = 7, 6, 3, 2 \) and \( Y = 5, 3, 2, 0 \). In our example \( k = 4 \), \( b = 3 \), and \( w = 32 \). The sequence \( X \) is represented by the word

\[
0000 0000 0000 0000 0111 0110 0011 0010.
\]

Note that the rightmost bit fields are used and that none of the test bits are set.

First we define some constants and utility routines.

```c
#define W 32   /* machine dependent */
typedef unsigned int word;   /* W-bit word */
word Masks[5];   /* 5 = Log(W) */

typedef int boolean;
#define TRUE 1
#define FALSE 0
#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define AND &
#define OR |
#define XOR -

int Log(int n)
{
    int x = -1;

    while (n > 0) {n >>= 1; x++;
    return x;
}
```
The function above computes the position of the leftmost nonzero bit in $O(\log k)$ time. This next function is a bit more exotic.

```c
word CopyTestBit(word A, int b)
{
    return A - (A>>b);
}
```

It translates the word 0000 0000 0000 1000 1000 0000 1000 0000 into the word 0000 0000 0000 0111 0111 0000 0111 0000. That is, it assumes that only the test bits are set and copies the test bits into the empty positions of each bit field and at the same time clears the test bits.

We will also make repeated use of a number of bit mask constants. In our example we use the following masks.

```c
M[0] = 1000 1000 1000 1000 1000 1000 1000 1000
M[1] = 0000 0000 0000 0000 1000 1000 1000 1000
M[2] = 0000 0000 1000 1000 0000 0000 1000 1000
M[3] = 0000 1000 0000 1000 0000 1000 0000 1000
```

These numbers can be computed in $O(\log k)$ time as follows.

```c
void MakeMasks(word M[], int k, int b)
{
    int s, t;
    const int log_k = Log(k);

    M[0] = 1<<b;
    s = b+1;
    for (t = 0; t <= log_k; t++, s <<= 1)
        M[0] = M[0] OR M[0]<<s;

    s = (b+1)<<log_k;                /* s = k(b+1) */
    for (t = 0; t <= log_k; t++, s >>= 1)
        M[t+1] = M[t] XOR M[t]<<s;
}
```

We are now ready to implement the merging algorithm. Our first task is to reverse the sequence $X = 7, 6, 3, 2$ and move it $k$ positions to the left. That is, we want to reverse the order of the $8$ numbers in the word

```
0000 0000 0000 0000 0111 0110 0011 0010.
```

First the mask $M[1]$ is used to swap the first and second half of the word. Then $M[2]$ is used to perform similar swapping actions in parallel on each half of the word. This procedure is repeated until the word has been reversed. The reversal is accomplished in $O(\log k)$ time, since each phase is
performed in constant time. Combining the reversed version of X with Y we produce the word \( Z = 2 \ 3 \ 6 \ 7 \ 5 \ 3 \ 2 \ 0 \) (given in hexadecimal notation) representing a bitonic sequence. For the precise details we refer to the code at the end of this section.

We can now apply Lemma 6.1. In the first pass we use \( M[1] \) to divide \( Z \) into two words \( A \) and \( B \) containing the elements to be compared. Note that the elements are moved into corresponding positions in the two words. From these words we produce a bit mask \( N \) with test bits indicating which elements of \( A \) that are smaller than or equal to the corresponding elements of \( B \).

\[
\begin{align*}
A &= 5 \ 3 \ 2 \ 0 \ 0 \ 0 \ 0 \ 0 \\
B &= 2 \ 3 \ 6 \ 7 \ 0 \ 0 \ 0 \ 0 \\
N &= 0 \ 0 \ 8 \ 8 \ 8 \ 8 \ 8 \ 8 
\end{align*}
\]

Using this bit mask we can produce the sequences \( L \) and \( R \) of Lemma 6.1 from the following computations.

\[
\begin{align*}
B \text{ AND } N &= 0 \ 3 \ 6 \ 7 \ 0 \ 0 \ 0 \ 0 \\
A - (A \text{ AND } N) &= 5 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
(A \text{ AND } N) \gg s &= 0 \ 0 \ 0 \ 0 \ 0 \ 3 \ 2 \ 0 \\
(B - (B \text{ AND } N)) \gg s &= 0 \ 0 \ 0 \ 0 \ 2 \ 0 \ 0 \ 0 
\end{align*}
\]

Combining these words with the OR operator we have performed the first pass of the merging subroutine in constant time and produced the word

\[
Z = 5 \ 3 \ 6 \ 7 \ 2 \ 3 \ 2 \ 0.
\]

We now repeat this procedure in parallel on the two halves of \( Z \). The code is essentially the same, the only difference being that we now use the bit mask \( M[2] \) instead.

\[
\begin{align*}
A &= 6 \ 7 \ 0 \ 0 \ 2 \ 0 \ 0 \ 0 \\
B &= 5 \ 3 \ 0 \ 0 \ 2 \ 3 \ 0 \ 0 \\
N &= 0 \ 0 \ 8 \ 8 \ 8 \ 8 \ 8 \ 8 
\end{align*}
\]

In this case the computations yield the following result.

\[
\begin{align*}
B \text{ AND } N &= 0 \ 0 \ 0 \ 0 \ 2 \ 3 \ 0 \ 0 \\
A - (A \text{ AND } N) &= 6 \ 7 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
(A \text{ AND } N) \gg s &= 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 2 \ 0 \\
(B - (B \text{ AND } N)) \gg s &= 0 \ 0 \ 5 \ 3 \ 0 \ 0 \ 0 \ 0 
\end{align*}
\]

Combining these numbers we have finished the second pass of the merging subroutine and produced the word
\[ Z = 6 \ 7 \ 5 \ 3 \ 2 \ 3 \ 2 \ 0. \]

Repeating this process the words will eventually be merged. Each iteration is performed in constant time and the number of passes is \( O(\log k) \). The complete code is as follows.

```c
word Merge(word X, word Y, int k, int b, word Masks[])
{
    word Z, A, B, M, N;
    int s, t;
    const int log_k = Log(k);

    // s = k(b+1) * /
    for (t = 0; t <= log_k; t++, s >>= 1) {
        M = CopyTestBit(Masks[t+1], b);
        X = (X AND M) << s OR (X - (X AND M)) >> s;
    }
    Z = X OR Y;

    // s = k(b+1) * /
    for (t = 0; t <= log_k; t++, s >>= 1) {
        M = CopyTestBit(Masks[t+1], b);
        A = (Z AND M) << s;
        B = Z - (Z AND M);
        N = ((B OR Masks[0]) - A) AND Masks[0];
        M = CopyTestBit(M, b);
        Z = (B AND N) OR
            (A - (A AND N)) OR
            (A AND N) >> s OR
            (B - (B AND N)) >> s;
    }
    return Z;
}
```

**Merging**

It is straightforward to incorporate the packed merging routine from the previous section into a mergesort algorithm using the technique described in Section 2.6. The algorithm is based on linked lists and it is useful to have a utility routine to add a record to the tail of a list.

```c
typedef struct listrec *list;
struct listrec {
    word k;
    list next;
};
```
The mergesort algorithm below sorts the first n elements of the list X. The sorted sublist is returned. After the call the variable X will point to the remainder of the list. The parameter k indicates how many integers that can be packed into a word and b is the number of bits of each integer.

```c
void AddToTail(list r, list *head, list *tail)
{
    if (!*head)
        *head = r;
    else
        (*tail)->next = r;
    *tail = r;
}
```

The mergesort routine above sorts a list of words, where each word contains k integers. The following routine produces such a list. Before packing the numbers into a word they are sorted in O(klogk) time using Mergesort. In fact, we use the mergesort routine above with k equal to 1. The routine also returns the number of keys and pads the last word with b-bit integers of maximal size.

```c
list Pack(list x, int k, int b, int *n)
{
    list head, tail, temp, top;
    const int s = (b+1)<<Log(k); /* s = k(b+1) */
    const int maxkey = CopyTestBit(1<<b, b);
    int pos;
    *n = 0;
```
MakeMasks(Masks, 1, b);
head = NULL;
while (x) {
    /* sort the first k elements */
    top = Mergesort(kx, K, 1, b);
    /* create a new list record */
    temp = (list) malloc(sizeof(struct listrec));
    temp->k = 0;
    temp->next = NULL;
    AddToTail(temp, &head, &tail);
    /* pack the k elements into one word */
    pos = 0;
    for (; top; top = top->next) {
        (am)++; 
        tail->K = tail->K OR top->K<<pos;
        pos += b+1;
    }
    while (pos < s) {  /* pad the last word */
        tail->K = tail->K OR maxkey<<pos;
        pos += b+1;
    }
    return head;
}

The unpacking of a packed list is straightforward.

list Unpack(list x, int n, int k, int b) {
    list head, tail, temp;
    const int maxkey = CopyTestBit(1<<b, b);
    int i;

    head = NULL;
    for (i = 0; i < n; i++) {
        if (i != 0 && i % k == 0)
            x = x->Next;
        temp = (list) malloc(sizeof(struct listrec));
        temp->next = NULL;
        AddToTail(temp, &head, &tail);
        tail->K = x->K AND maxkey;
        x->K >>= b+1;
    }
    return head;
}

The next routine computes the number of keys k that can be packed into one word, packs the list, calls the Mergesort routine, and unpacks the now sorted list.
list PackSort(list x, int b)
{
    list xpaked;
    int k;  /* Length of a sequence */
    int i;  /* k = 2^-i, for some integer i */
    /* 2k(b+1) <= W */
    int n;
    for (i = 1; (b+1)<<i+1) <= W; i++)
    {
        k = 1<<i-1;
        xpaked = Pack(x, k, b, &n);
        MakeMasks(Masks, k, b);
        xpaked = Mergesort(xpaked, (n-k-1)/k, k, b);
        return Unpack(xpaked, n, k, b);
    }
}

Range Reduction
The range reduction phase of the algorithm is performed using the technique of Kirkpatrick and Reisch [49]. The following utility routines return the b/2 most significant digits and the (b+1)/2 least significant digits of a b-bit number, respectively.

    word High(word x, int b)
    {
        return x >> (b/2);
    }

    word Low(word x, int b)
    {
        return x << (W - b/2) >> (W - b/2);
    }

During the bucketing we keep track of each bucket that is used. The first time a bucket is used we add it to the list Batch.

    void IntoBatch(word x, list *Batch)
    {
        list temp;
        temp = (list) malloc(sizeof(struct listRec));
        temp->K = x;
        temp->next = *Batch;
        *Batch = temp;
The bucketing is slightly involved. To avoid having to initialize the bucket array we use a standard technique [1, Problem 2.12]. Each bucket contains a pointer check that points to an array Active of active buckets. Each time a bucket is accessed, we verify that the contents are not random by making sure that the pointer in that bucket points to an active region of the stack and that the back pointer in the array points to the entry.

The following routine adds the list record \( x \) to the bucket array \( B \). If \( hi \) is true the high order bits of the \( b \)-bit word in \( x \) is used to decide where to put the record, otherwise the low order bits are used. If this is the first use of the bucket, it is added to Batch:

```java
void IntoBucket(list x, bucket B[], int b, boolean hi, word Active[], word *firstfree, list *Batch)
{
    word a, p;
    a = hi ? High(x->K, b) : Low(x->K, b);
    /* Check if bucket is nonactive */
    /* If so, add to Active */
    p = B[a].check;
    if (p >= *firstfree || Active[p] != a) {
        B[a].head = B[a].tail = NULL;
        B[a].check = *firstfree;
        Active[(*firstfree++)] = a;
        IntoBatch(a, Batch);
    }
    /* into bucket */
    AddToTail(x, &(B[a].head), &(B[a].tail));
}
```

The next routine finds the minimum element in bucket \( B[i] \) and moves it to the front of the list. This element is left behind in the bucket and the rest of the list is returned.

```java
list Tail(bucket B[], int i)
{
    word min = "0";
    list t;

    for (t = B[i].head; t; t = t->next)
        min = MIN(min, t->K);
```
for (t = B[i].head; t->K != min; t = t->next) 
  t->K = B[i].head->K;
B[i].head->K = min;
B[i].tail = B[i].head;
t = B[i].head->next;
B[i].head->next = NULL;
return t;

The main routine of the new integer sorting algorithm is presented below. The elements are distributed into bucket array \( H \) according to their high order bits. The active buckets of \( H \) are traversed and for each bucket we leave the minimum element behind and move the rest of the elements into the bucket array \( L \) according to their low order bits.

During the bucketing all active buckets are put in the \textbf{Batch} list. This list will contain at most \( n \) integers, each consisting of at most \( \lfloor b + 1 \rfloor / 2 \) bits. This list is sorted either by a recursive call to \textbf{Sort} or by a call to the packed mergesort routine \textbf{PackSort}. The range reduction routine \textbf{Sort} is called at most \( 1 + \log \log n \) times.

Using the sorted \textbf{Batch} list it is easy to collect the elements in ascending order. We traverse the buckets of the low order bucket array \( L \) in ascending order and put the elements back into the high order bucket array \( H \). Finally, the active buckets of \( H \) are traversed in ascending order and the sorted output is assembled.

```c
list Sort(list x, int n, int b) {
  bucket *H, *L;  /* bucket array, high and low order bits */
  word *Hactive, *Lactive;  /* array of active buckets */
  word Hfirstfree = 0;  /* number of occupied buckets */
  word Lfirstfree = 0;
  list Batch = NULL;  /* The reduced sorting problem */
const boolean HIGH = TRUE, LOW = FALSE;
  Word i, p;
  list next, s, t, head, tail;

  H = (bucket *) malloc((1<<(b+1))/2) *
  sizeof(struct bucketrec));
  Hactive = (Word *) malloc(n * sizeof(Word));
  L = (bucket *) malloc((1<<b)/2) *
  sizeof(struct bucketrec));
  Lactive = (Word *) malloc(n * sizeof(Word));
```
/* bucketing on high order bits */
for ( ; x; x = next) {
    next = x->next;
    x->next = NULL;
    Int0Bucket(x, H, b, HIGH, Hactive, &Hfirstfree, &Batch);
}
/* bucketing on low order bits */
for (i = 0; i < Hfirstfree; i++) {
    s = Tail(H, Hactive[i]);
    for ( ; s; s = next) {
        next = s->next;
        s->next = NULL;
        Int0Bucket(s, L, b, LOW, Lactive, &lfirstfree, &Batch);
    }
}
if (((b+1)/2) < Log(Log(n+4)) >= W && b >= 4)
    Batch = Sort(Batch, n, (b+1)/2);
else
    Batch = PackSort(Batch, (b+1)/2);

/* move the elements back to the high order bucket table */
for (s = Batch; s; s = s->next) {
    p = L[s->K].check;
    if (p < lfirstfree && Hactive[p] == s->K) {
        for (t = L[s->K].head; t; t = next) {
            next = t->next;
            t->next = NULL;
            H[High(t->K, b)].tail->next = t;
            H[High(t->K, b)].tail = t;
        }
        L[s->K].head = NULL;
    }
}
/* traverse the buckets and put the output together */
head = tail = NULL;
for (s = Batch; s; s = s->next) {
    p = H[s->K].check;
    if (p < Hfirstfree && Hactive[p] == s->K) {
        if (!head)
            head = H[s->K].head;
        else
            tail->next = H[s->K].head;
        tail = H[s->K].tail;
        H[s->K].head = NULL;
    }
}
return head;
## Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log n )</td>
<td>binary logarithm; ( \log_2 n )</td>
<td>3</td>
</tr>
<tr>
<td>( \log^* n )</td>
<td>iterated logarithm function</td>
<td>3</td>
</tr>
<tr>
<td>( w )</td>
<td>word length of a RAM</td>
<td>3</td>
</tr>
<tr>
<td>( \Sigma )</td>
<td>alphabet</td>
<td>2</td>
</tr>
<tr>
<td>( m )</td>
<td>size of alphabet</td>
<td>2</td>
</tr>
<tr>
<td>( x[i] )</td>
<td>ith character of a string ( x )</td>
<td>2</td>
</tr>
<tr>
<td>( S )</td>
<td>number of distinguishing characters</td>
<td>4</td>
</tr>
<tr>
<td>( S )</td>
<td>average length of a distinguishing prefix</td>
<td>4</td>
</tr>
<tr>
<td>( S_{\text{max}} )</td>
<td>maximum length of a distinguishing prefix</td>
<td>4</td>
</tr>
<tr>
<td>( B )</td>
<td>number of distinguishing bits</td>
<td>5</td>
</tr>
<tr>
<td>( \bar{B} )</td>
<td>average length of a binary distinguishing prefix</td>
<td>5</td>
</tr>
<tr>
<td>( B_{\text{max}} )</td>
<td>maximum length of a binary distinguishing prefix</td>
<td>5</td>
</tr>
<tr>
<td>( L^2 )</td>
<td>( f \in L^2 ) if ( \int f^2(x) , dx &lt; \infty )</td>
<td>6</td>
</tr>
<tr>
<td>( H )</td>
<td>entropy per character</td>
<td>7</td>
</tr>
<tr>
<td>( A_{\text{tie}}^n )</td>
<td>average depth of a trie</td>
<td>51</td>
</tr>
<tr>
<td>( A_{\text{level}}^n )</td>
<td>average depth of an LC-trie</td>
<td>51</td>
</tr>
<tr>
<td>( M_n )</td>
<td>maximal spacing</td>
<td>51</td>
</tr>
</tbody>
</table>
References


REFERENCES 107