Dynamic Length-Restricted Coding

by

Travis Gagie



A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2003 by Travis Gagie

Abstract

Dynamic Length-Restricted Coding

Travis Gagie

Master of Science

Graduate Department of Computer Science

University of Toronto

2003

Suppose that S is a string of length m drawn from an alphabet of n characters, d of which occur in S. Let P be the relative frequency distribution of characters in S.

We present a new algorithm for dynamic coding that uses at most $\lceil \lg n \rceil + 1$ bits to encode each character in S; fewer than $(H(P) + 4.5)m + d \lg n$ bits overall, where H is Shannon's entropy function; and $O\left((H(P) + 1)m + d \log^2 n\right)$ time to encode and decode. This algorithm does not require P to be known before it encodes S.

We extend recent results by Evans and Kirkpatrick for restructuring binary trees. In particular, we present a new algorithm for constructing node-oriented alphabetic minimax trees. We also describe how to efficiently implement an abstract data type that stores a dynamic list of non-negative integers and supports an operation to determine whether there exists a binary tree on nodes of those depths.

Acknowledgements

First and foremost, I'm grateful for the patient guidance I've received from my supervisor, Faith Fich. I'm also grateful to: my second reader, Ken Sevcik, who taught me information theory; the rest of the Department of Computer Science, especially Linda Chow and Shera Karim; Professor Will Evans of UBC; the staff of St. John's Rehabilitation Hospital, especially Maria Kadar; my friends and fellow graduate students, especially Jim Bergey, Roza Christodoulopoulou, Harold Connamacher, Lisa Drewell, Wayne Hayes, Stephanie Horn, Kleoni Ioannidou, Yannis Kassios, Amy Kaufman (née Dickieson), Vlad Kolesnikov, Antonina Kolokolova, Gerry Lennox, Noel Pawlikowski, Mike Tsang, Charlotte Vilarem, Ben Vitale, and, most of all, Nataša Pržulj; and, as always, my family. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Contents

Glossary of Notation **vii**

1 Introduction **1**

1.1 Related Work 5

1.2 Upper and Lower Bounds for Coding 10

2 Huffman Coding **18**

2.1 Static Huffman Coding 18

2.2 Dynamic Huffman Coding 20

2.3 Length-Restriction 23

3 $(\lceil \lg n \rceil + 1)$-Restricted Coding **26**

3.1 Dynamic Minimax Trees 26

3.2 Dynamic Minimax Coding 29

3.3 Analysis 33

3.4 Alphabetic Coding 36

4 Restructuring Trees **42**

4.1 Restructuring Unordered Trees 42

4.2 Restructuring Leaf-Oriented Binary Search Trees 44

4.3 Restructuring Binary Search Trees 45

4.4 Restructuring Optimal Binary Search Trees 46

4.5 Reordering Binary Search Trees 49

5 Alphabetic Minimax Trees **51**

5.1 Leaf-Oriented Alphabetic Minimax Trees 53

5.2 Node-Oriented Alphabetic Minimax Trees 54

5.3 A Reduction 56

5.4 Construction 58

5.5 A Lower Bound for Dynamic Leaf-Oriented Alphabetic Minimax Trees 67

6 Sequence Maxima Trees **68**

6.1 Depth Sequences 68

6.2 Sequence Maxima Trees 69

6.2.1 The Function β 69

6.2.2 Definition of Sequence Maxima Trees 72

6.2.3 Implementation of Internal Nodes 76

6.2.4 Implementation of Queries 77

6.2.5 Implementation of Increase, Insert, and Split 78

6.2.6 Implementation of Decrease, Delete, and Join 83

7 Conclusion **85**

Bibliography **87**

List of Figures

4.1 Rotating u right. 48

4.2 Rotating v's parent left and u right. 48

5.1 $A(T; 1, 2, 3) = 3$ and $A'(1, 2, 3) = A(T'; 1, 2, 3) = 4$. 57

5.2 Inserting y' into T. 65

6.1 For $a \geq 0$, $\beta(-1, -2, -2, -2, 0, -5, -2, -5, -2, 0) = \beta(0, 0, 0, 0, 0; a)$. . . . 73

6.2 The sequence maxima tree for $-1, -2, -2, -2, 0, -5, -2, -5, -2, 0$. 74

6.3 The list of u's children in the sequence maxima tree T is implemented by the balanced binary search tree B_u. 76

6.4 Increasing v's weight to equal u's leaf siblings' weight. 80

6.5 Increasing v's weight to less than u's leaf siblings' weight. 81

Glossary of Notation

S	the string to be encoded
s_i	the ith character of S
m	the length of S
n	the size of the alphabet from which S is drawn
d	the number of distinct characters in S
P	the relative frequencies of characters in S
p_a	the relative frequency of the character a in S
S_i	the prefix of S of length i
$H(P)$	the binary entropy of P
$\#_a(S)$	the number of occurrences of the character a in S
$\lg x$	$\log_2 x$
$n^{O(1)}$	the set of all functions bounded by polynomials in n
$w_{i,a}$	the weight assigned to the leaf labeled a after i characters of S have been processed
W_i	the weight sequence $w_{i,1}, \ldots, w_{i,n}$
$q_{i,j}$	$\left(\sum_{k=1}^{j-1} \frac{\#_k(S_{i-1})}{2(i-1)}\right) + \frac{\#_j(S_{i-1})}{4(i-1)} + \frac{j}{2n}$
$z_{i,j}$	the codeword assigned to the character j after i characters of S have been processed

$\alpha(T;W)$	the leaf-oriented alphabetic minimax cost of the tree T with respect to the sequence W
$\alpha(W)$	the leaf-oriented alphabetic minimax cost of the sequence W
$A(T;W)$	the node-oriented alphabetic minimax cost of the tree T with respect to the sequence W
$A(W)$	the node-oriented alphabetic minimax cost of the sequence W
$\beta(W;a)$	the least integer c such that W can be partitioned into c subsequences, each of leaf-oriented alphabetic minimax cost at most a
T_v	the subtree of the tree T rooted at the node v
B_u	the balanced binary search tree implementing the list of children of the node u in a sequence maxima tree

Chapter 1

Introduction

Suppose that we are given a string $S = s_1 \dots s_m$ drawn from the alphabet $\{1, \dots, n\}$, and asked to find a short binary encoding. We are primarily interested in the length of the encoding, and secondarily, in the time required to encode and decode.

In particular, suppose that we want to encode S character by character, so that the binary substrings encoding each character do not overlap; and to decode character by character, so that we recognize the end of each such substring when we reach it.

These substrings are called *codewords*. To decode in this way, it must be that immediately after we have decoded any character, there cannot be two codewords such that one is a proper prefix of the other and either could legally come next in the encoding. In this case, the codewords are called *self-delimiting*.

There are algorithms for encoding and decoding which do not adhere to these restrictions, for example Lempel-Ziv coding [ZL77, ZL78] and arithmetic coding [WNC87]. However, in this thesis, we consider only those that do. We also do not consider lossy coding, or coding based on more sophisticated statistics than character frequencies, such as Markov models.

The algorithms we consider include the simplest and most common. Furthermore, there are close upper and lower bounds on the number of bits and time required for

encoding. These bounds are simple functions of: m; n; d, the number of distinct character in S; and $P = p_1, \ldots, p_n$, the relative frequencies of characters in S. The relative frequency of a character a is the number of occurrences of that character, $\#_a(S)$, divided by m.

A *static* code permanently assigns a codeword to each of the d distinct characters which occur in S, such that no codeword is a prefix of another. If both the encoder and the decoder know the code chosen, then S can be encoded as the concatenation of the codewords for $s_1, \ldots, s_m$. The length of the encoding is then m multiplied by the average codeword length in the encoding. However, the code chosen is often specific to the string, in which case it may have to be recorded as a preface to the encoding so that the decoder will have access to it.

A *strict* binary tree is a tree in which every internal node has exactly two children. We can make a code tree strictly binary without increasing the length of the codeword assigned to any character. To do this, we *compress* the code tree. This means that we replace each path whose internal nodes all have degree 1 by a single edge.

To record a strict binary code tree on d leaves takes $O(d \log n)$ bits. It takes $O(d)$ bits to specify a strict binary tree on d leaves, by conducting an in-order traversal, recording a 0 for every increase in depth from one node to the next and a 1 for every decrease in depth from one node to the next. It takes $O(d \log n)$ bits to record an assignment of d characters to leaves.

In a static code, no codeword is a prefix of another. Thus, every static code can be represented as a binary trie, called a *code tree*. Each codeword is the sequence of edge labels on a path from the root to a leaf in the code tree; the leaves are labeled with the characters which occur in S. The reverse is also true. Consider any binary trie where each character occurring in S labels one leaf. Then this trie represents a static code. To encode a character s_i, we record the binary string consisting of the edge labels on the path from the root to the leaf labeled s_i to the root. We can find this path by finding its

reverse, that is, the path from the leaf labeled s_i to the root.

Generally, constructing a good static code requires us to know P in advance. This means we must make two passes over S: one to count the occurrences of each character and one to compute the actual encoding. *Dynamic* coding (also called 'adaptive coding') avoids this problem by working online: these algorithms encode the $(i+1)$st character using a code based on the relative frequencies of characters in the prefix $S_i = s_1 \dots s_i$ already encoded. Dynamic codes may also outperform static codes when the relative frequencies of characters are significantly different in different parts of the string.

Dynamic codes can be represented as a sequence of binary tries, one for each of the m characters of the string S. When the decoder is decoding the ith codeword, it knows the relative frequencies in the prefix already decoded, and is therefore able to construct the same code tree used by the encoder for the ith character. These code trees differ from static codes for the prefixes in that every character must be assigned a codeword, whether it has occurred or not; therefore, they always have n leaves. This guarantees that even characters which have not previously occurred can be encoded. Having the algorithm construct a new code tree from scratch for each character of S takes a lot of time. Instead, normally techniques are used which allow the code tree to be updated as characters are encoded (or decoded).

A code is b-restricted if every codeword is of length at most b. Such codes are called *length-restricted*. Length-restricting a code is equivalent to restricting the height of the code tree. Given any code, we can construct an n-restricted code without increasing the length of the codeword assigned to any character, by compressing the code tree.

Since a code tree with k leaves must have height at least $\lceil \lg k \rceil$, b-length restriction is impossible for static codes when $b < \lceil \lg d \rceil$, or for dynamic codes when $b < \lceil \lg n \rceil$. In this thesis, we consider $\Theta(\log n)$-restricted codes. These codes ensure good worst-case performance.

An *alphabetic* code ensures that the lexicographic order of the codewords is the same

as that of the characters. If $a < b$ and both occur in S, the leaf labeled a must be visited before the leaf labeled b in an in-order traversal of the code tree. Since the encodings produced preserve lexicographic order, they can be sorted without being decoded. Alphabetic code trees are also called 'alphabetic trees'. If an alphabetic code tree is augmented at each internal node v by storing the lexicographically largest character assigned to any leaf in v's left subtree, then the code tree becomes a leaf-oriented binary search tree.

When we refer to the ith node with a certain property in a tree, we mean the ith node with that property visited in an in-order traversal. In particular, the ith leaf of a tree is the ith leaf in order from left to right. Note that the ith leaf in a strict binary tree is a child of either the $(i-1)$st or ith internal node. It is a descendant of both, if they exist. If we write that the nodes or leaves of a tree are of depths $l_1, \ldots, l_n$, we mean that for $1 \leq i \leq n$, the ith node or leaf is of depth l_i. We say a node u is to the *left* of a node v if u precedes v in an in-order traversal, and to the *right* of v if u follows v. If the left-to-right order of the nodes in a tree is important, then we call the tree *ordered*.

Throughout this thesis, our model of computation is the unit-cost word RAM [Hag98], with $\Theta(\log n)$-bit words.

In the remainder of this chapter, we present general background on coding. In particular, we give an overview of codes which are dynamic, length-restricted, alphabetic, or some combination of these. We also describe basic upper and lower bounds of the length of the encoding in terms of $H(P)$ and m, where H is Shannon's entropy function.

In Chapter 2, we describe coding algorithms called static and dynamic Huffman coding. We show that two dynamic Huffman coding algorithms can be easily modified to produce $O(\log n)$-restricted codes.

The main result of this thesis is a new, efficient algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted, non-alphabetic coding, which we present in Chapter 3. This algorithm overall uses fewer than $(H(P) + 4.5)m + d \lg n$ bits and $O\left((H(P)+1)m + d \log^2 n\right)$ time to encode and decode. We also present an algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted,

alphabetic coding.

In Chapter 4, we give new proofs of recent results, by Evans and Kirkpatrick [EK00], for restructuring leaf-oriented binary search trees on n leaves to have height at most $\lceil \lg n \rceil + 1$ without the depth of any leaf increasing by more than 2; and for restructuring node-oriented binary search trees on n nodes to have height at most $\lceil \lg n \rceil$ without the depth of any node increasing by more than $\lg \lg(n+1)$.

In Chapter 5, we define leaf-oriented and node-oriented alphabetic minimax trees. We then present a new algorithm for constructing a node-oriented alphabetic minimax tree for a sequence of n integers, in $O(n)$ time and $O(h)$ space, where h is the height of the resulting tree.

In Chapter 6, we present a new data structure, called a sequence maxima tree. Using this data structure, we can efficiently recompute the leaf-oriented alphabetic minimax cost of a sequence of integers, after the sequence is updated.

In Chapter 7 we summarize our results and discuss directions for future work.

1.1 Related Work

In the previous section, we described three attributes of codes: they can be static or dynamic, length-restricted or length-unrestricted, and alphabetic or non-alphabetic. These give rise to eight combinations. Six of these have been extensively studied, as problems in coding or as problems in data structures. In this section, we survey results for these types of codes. A more detailed survey is given by Moffat and Turpin [MT02]. When we refer to a binary search tree as *optimal*, we mean that it minimizes the expected number of comparisons made in a search. When we refer to a code as optimal, we mean that it minimizes the expected codeword length.

Several results relate to data structures, but can be applied to coding. In general, algorithms for constructing binary search trees, given a set of keys, can be used to

construct code trees.

Given a binary search tree T on $n-1$ nodes of depths $l_1, \ldots, l_{n-1}$, we can *extend* T; that is, we attach left and right children to any node that does not already have them. Let T' be the resulting tree. We can make T' an alphabetic code tree by labeling each edge from a parent to its left child with a 0, each edge from a parent to its right child with a 1, and the leaves with the characters of the alphabet in lexicographic order.

Suppose that the probability of searches for keys less than the first key in T is p_1; for $1 < i < n$, the probability of searches for keys strictly between the $(i-1)$st and ith keys in T is p_i; and the probability of searches for keys larger than the $(n-1)$st key in T is p_n. Then the expected codeword length using T' to encode S is the same as the expected number of comparisons made during an unsuccessful search in T.

Alternatively, given a binary search tree T on n nodes of depths $l_1, \ldots, l_n$, we can construct a leaf-oriented binary search tree T' as follows. For any internal node u that does not have a left child, we attach as u's left child a leaf v with the same key as u. For any internal node u' that has a left child, we insert between u' and u''s left child a node v' with the same key as u'. We attach as v''s right child a leaf with the same key as u'. We can make T' an alphabetic code tree, by labeling each edge from a parent to its left child with a 0, each edge from a parent to its right child with a 1, and each leaf with the character stored at that leaf.

Suppose that, for $1 \leq i \leq n$, the probability of a search for the ith key in T is p_i. Then the expected codeword length using T' to encode S is at most twice the expected number of comparisons made during a successful search in T.

If we do not require an alphabetic code, we can use a third approach. Given a binary search tree T on n nodes of depths $l_1, \ldots, l_n$, we can make T a trie T', by labeling each each edge from a parent to its left child with a 0 and each edge from a parent to its right child with a 1. Note that this will not be a code tree, because the sequence of edge labels on the path from the root to a node v is a prefix of the sequence of edge

labels on the path from the root to any of v's descendants. For each character i, let the codeword for i be the l_i edge labels on the path in T' from the root to the ith node, preceded by a self-delimiting representation of l_i. One such representation is the Elias gamma code [Eli75] for $l_i + 1$. The Elias gamma code for a positive integer b consists of $\lfloor \lg b \rfloor$ 0's, followed by the $(\lfloor \lg b \rfloor + 1)$-bit binary representation of b.

Suppose that, for $1 \leq i \leq n$, the probability a search for the ith key in T is p_i. If the expected number of comparisons made during a successful search in T is c, then the expected codeword length using this code to encode S is at most $c + 2\lfloor \lg c \rfloor + 1$.

Using these ideas, dynamic codes can be obtained from dynamic binary search trees. They can also be obtained from other dynamic data structures. Albers and Westbrook [AW96] survey several uses of dynamic data structures for data compression.

The problem of static, length-unrestricted, non-alphabetic coding was addressed by Shannon [Sha48], whose results we discuss in section 1.2; and Fano [Fan49], who presented a similar algorithm. Later, Huffman [Huf52] presented a greedy algorithm for constructing an optimal code tree in $O(n \log n)$ time. We discuss Huffman's algorithm in Chapter 2. Subsequent work has largely focussed on Huffman's algorithm. For example, van Leeuwen [vL76] showed that the algorithm could be made to run in $O(n)$ time if the relative frequencies were pre-sorted. Schwartz [Sch64] presented a modification to Huffman's algorithm that computes the code tree with minimum height that could result from the original algorithm. Bookstein and Klein [BK93] present an overview of such developments up to 1993.

Gilbert and Moore [GM59] gave a standard $O(n^3)$ time dynamic programming algorithm for constructing an optimal binary search tree. It is interesting to note that this was first presented as an algorithm for constructing an optimal static, length-unrestricted, alphabetic code tree. Knuth [Knu71] improved Gilbert and Moore's algorithm to run in $O(n^2)$ time, using a monotonicity property of the root to reduce the number of subtrees considered. There are several more efficient algorithms for constructing optimal alpha-

betic code trees, but they do not solve the more general problem of constructing optimal binary search trees [HT71, GW77, HKT79, LP98]. Mehlhorn [Meh75, Meh77] presented a $O(n)$-time algorithm for constructing nearly optimal binary search trees, which we discuss in section 1.2. There has been recent work using alphabetic codes for communication protocols in which the receiver can ask questions of the sender [Wat00, WAF01].

Static, length-restricted, non-alphabetic coding was first suggested by Gilbert [Gil71]. He considered the problem of constructing a prefix code based on the relative frequencies in a sample of the string to be encoded. Length restriction prevents the length of the encoding being too long even if the sample is unrepresentative. Super-polynomial time algorithms for static, length-restricted coding were presented by Hu and Tucker [HT72] and Van Voorhis [Van74], before a polynomial time dynamic programming algorithm was found by Garey [Gar74]. Since then, faster and more sophisticated algorithms have been found for constructing optimal height-restricted trees [Ita76, LH90, Sch95, TM95, Tur98]. There has also been work on fast construction of nearly optimal height-restricted trees [FK93, Sch95, ML96, ML97, MPL98, MPL99, LM01a, LM01b].

Garey's algorithm can also be used for constructing optimal static, length-restricted, alphabetic codes. Itai [Ita76] and Wessner [Wes76] independently presented an algorithm adapted from Garey's that can be used for the more general problem of constructing optimal height-restricted binary search trees. Other algorithms for static, length-restricted, alphabetic coding have been presented [Lar87, LP94, LMP99, LM02]. In Chapter 4, we discuss recent results by Evans and Kirkpatrick [EK00] which can be used to restrict the height of ordered binary trees. They showed that any leaf-oriented binary search tree on n leaves can be restructured to have height at most $\lceil \lg n \rceil + 1$, without the depth of any leaf increasing by more than 2. This result can be used for length-restricting static codes without increasing any codeword length by more than 2. They also showed that any node-oriented binary search tree can be restructured to have height at most $\lceil \lg n \rceil$, without the depth of any node increasing by more than $\lg \lg(n+1)$.

In Chapter 2, we discuss two algorithms for dynamic, length-unrestricted, non-alphabetic coding. These are based on a dynamic version of Huffman's algorithm, which was discovered independently by Faller [Fal73] and Gallager [Gal78], and refined by Knuth [Knu85]. Vitter [Vit87] partially analyzed this algorithm and presented an improvement. Jones [Jon88] has presented an unrelated algorithm which uses a variant of splay trees for dynamic coding.

An algorithm by Grinberg, Rajagopalan, Venkatesan, and Wei [GRVW95] uses splay trees for dynamic, length-unrestricted, alphabetic coding. However, the number of bits and the amount of time that this algorithm uses is only bounded to within a multiplicative factor of optimality. Belal, Selim, and Arafat [BSA99] have presented an algorithm for dynamic optimal alphabetic code trees; but insertions or deletions are constrained, and require $O(n)$ time in the worst case.

There has been relatively little work on dynamic, length-restricted coding. There are dynamic binary search trees which have restricted height and good amortized access time, such as biased search trees [BST85] and deepsplay trees [She89], but these have not been used to obtain codes. Bentley, Sleator, Tarjan and Wei [BSTW86] have presented an algorithm that uses a move-to-front heuristic to maintain a list of the characters which have occurred. The codeword for a particular symbol which has occurred is a representation of its position in the list. Using an Elias code, the ith position in the list is represented using $O(\log i)$ bits [Eli75]. Several similar algorithms have been presented [AM95, BE98]. Larmore and Hirschberg [LH90] and Milidiú and Laber [ML97] have asked whether it is possible to maintain an optimal height-restricted code tree. Liddell and Moffat [LM01a] have presented an algorithm for the non-alphabetic case, but with only empirical evidence of efficiency.

1.2 Upper and Lower Bounds for Coding

In this section, we present upper and lower bounds for the length of the encoding of S. Most lower bounds are information theoretic, and rely on Shannon's definition of the *entropy function* of a probability distribution $P' = p'_1, \ldots, p'_n$,

$$H(P') = -\sum_{i=1}^{n} p'_i \lg p'_i.$$

Throughout this thesis, 'entropy' will mean 'binary entropy'. By convention, $\lg 0$ is treated as $-\infty$ and $0 \lg 0$ is treated as 0.

The most well-known lower bound is Shannon's Noiseless Coding Theorem [Sha48].

Theorem 1.1 (Shannon, 1948). *The expected number of bits needed to encode the value of a random variable which takes a value according to the probability distribution P' is at least $H(P')$.*

A corollary of the Noiseless Coding Theorem is that, unless we use more sophisticated statistics than character frequencies, the expected number of bits needed to encode S is at least at least $H(P)m$. In the same paper, Shannon gave a construction which implies a nearly matching upper bound of $(H(P) + 1)m + O(d \log n)$ for the number of bits required.

Theorem 1.2 (Shannon, 1948). *Suppose that $P' = p'_1, \ldots, p'_n$ is a probability distribution with $p'_1, \ldots, p'_n > 0$. Then there exists a binary tree on n leaves that, in some order, are of depths $l_1, \ldots, l_n$, where $l_i \leq \lceil -\lg p'_i \rceil$.*

Corollary 1.3. *Using a static, non-alphabetic, length-unrestricted code, S can be encoded using*

$$(H(P) + 1)m + O(d \log n)$$

bits, where $P = p_1, \ldots, p_n$ is the relative frequency distribution of the characters in S.

Proof. By Theorem 1.2, there exists a code tree such that, if $p_i > 0$, then the leaf labeled i is of depth at most $\lceil -\lg p_i \rceil < 1 - \lg p_i$. The average codeword length in an encoding of S using this code is of length at most

$$\sum_{i \in S} p_i \lceil -\lg p_i \rceil < \sum_{i \in S} p_i (1 - \lg p_i) = H(P) + 1,$$

since $p_i = 0$ for all $i \notin S$. Thus the total number of bits used in the m codewords is less than $(H(P) + 1)m$. To record the code tree takes $O(d \log n)$ bits. □

In Chapter 3, we use the following corollary of Theorem 1.2 to obtain length-restricted, non-alphabetic codes.

Corollary 1.4. *Suppose that $P' = p'_1, \ldots, p'_n$ is a probability distribution. Then there exists a binary tree T of height at most $\lceil \lg n \rceil + 1$ with n leaves that, in some order, are of depths $l_1, \ldots, l_n$, where $l_i \leq \lceil -\lg p'_i \rceil + 1$.*

Proof. By Theorem 1.3, there exists a binary tree T' that has a distinct leaf v_i of depth at most $\lceil -\lg p'_i \rceil \leq \lceil \lg n \rceil$ for every $p'_i \geq 1/n$. Let T'' be a binary tree of height at most $\lceil \lg n \rceil$ which has a leaf for every $p'_i < 1/n$.

Attach T' and T'' as the left and right subtrees of a new node, and let T be the resulting binary tree. T is of height at most $\lceil \lg n \rceil + 1$. For every $p'_i \geq 1/n$, v_i is a leaf in the left subtree of T of depth at most $\lceil -\lg p'_i \rceil + 1$. For every $p'_i < 1/n$, there is a distinct leaf in the right subtree of T of depth at most $\lceil \lg n \rceil + 1 \leq \lceil -\lg p'_i \rceil + 1$. □

One way to prove Theorem 1.2 is by the Kraft Inequality [Kra49]. This characterizes the multi-sets of depths such that there exist binary trees with leaves at those depths.

Theorem 1.5 (Kraft, 1949). *If there exists a binary tree on n leaves of depths $l_1, \ldots, l_n$, then*

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1,$$

with equality if and only if the tree is strictly binary. Conversely, if $l'_1 \geq \cdots \geq l'_n \geq 0$ *with*

$$\sum_{i=1}^{n} 2^{-l'_i} \leq 1,$$

then there exists a binary tree on n *leaves of depths* $l'_1, \ldots, l'_n$.

The proofs of Theorems 1.2 and 1.5 are constructive, and can be used to obtain $O(n)$-time algorithms. Another way to prove Corollary 1.4 is to use Theorem 1.5, choosing $l'_1, \ldots, l'_n$ to be a permutation of

$$\left\lceil -\lg\left(\frac{p'_1 + \frac{1}{n}}{2}\right)\right\rceil, \ldots, \left\lceil -\lg\left(\frac{p'_n + \frac{1}{n}}{2}\right)\right\rceil$$

such that $l'_1 \geq \cdots \geq l'_n \geq 0$. Theorem 1.5 also has the following useful corollary. It can be obtained by applying both directions of Theorem 1.5 and observing that the order of summation does not matter.

Corollary 1.6. *Suppose that we are given a binary tree* T *on* n *leaves of depths* $l_1, \ldots, l_n$. *Then there exists a binary tree* T' *on* n *leaves with non-increasing depths from left to right and that, in some order, are of depths* $l_1, \ldots, l_n$.

Consider an optimal static, non-alphabetic code tree T. Notice that, for any two leaves u and v of T, if the character labeling u has greater relative frequency that the character labeling v, then u's depth is less than or equal to v's depth. Otherwise, we could reduce the weighted average distance from the root of T to the leaves by interchanging the characters labeling u and v. Therefore, by Corollary 1.6, there exists an optimal static, non-alphabetic code tree T' such that the characters labeling the leaves of T' are in non-decreasing order by relative frequency.

A consequence of this observation is that any algorithm for constructing optimal static, alphabetic code trees can be used for constructing optimal static, non-alphabetic code trees [SK64]. This is done by sorting the characters into non-decreasing order by relative frequency, considering the resulting sequence as a new alphabet, and constructing

an optimal alphabetic code for that alphabet. Sometimes, using a non-alphabetic code can produce a shorter encoding than using any alphabetic code. For example, if $n = 3$ and the relative frequencies of characters in S are $1/4, 1/2, 1/4$, then we can use a non-alphabetic code to encode S such that the average codeword length is $3/2$; but if we use an alphabetic code, the average codeword length is at least $7/4$.

A theorem similar to Theorem 1.2 was proven by Mehlhorn [Meh77] for nearly optimal binary search trees. The proof is constructive, and can be used to obtain an $O(n)$-time algorithm for constructing such trees. We follow Knuth's presentation [Knu98b], which is similar to Shannon's proof of Theorem 1.2.

Theorem 1.7 (Mehlhorn, 1977). *Suppose that* $P' = p'_1, p''_1, p'_2, \ldots, p'_{n-1}, p''_{n-1}, p'_n$ *is a probability distribution with* $p'_i > 0$ *for* $1 \leq i \leq n$. *Then there exists a strict binary tree on* n *leaves, such that the* i*th leaf is of depth at most* $\lceil -\lg p'_i \rceil + 1$ *and the* i*th internal node is of depth less than* $\min(-\lg p''_i, -\lg(p'_i + p'_{i+1}) + 1)$.

Proof. For $1 \leq i \leq n$, let

$$q_i = \sum_{j=1}^{i-1} \left(\frac{p'_j}{2} + p''_j + \frac{p'_{j+1}}{2} \right).$$

Let T be the binary trie for the binary representations of the fractions $q_1, \ldots, q_n$. The binary string consisting of the edge labels on the path from the root to the ith leaf of T (which may be infinitely deep), when considered as a binary fraction, is equal to q_i.

For $1 < i \leq n$, suppose that the binary representations of q_{i-1} and q_i agree on their first b bits. Then

$$2^{-b} > q_i - q_{i-1} = \frac{p'_{i-1}}{2} + p''_{i-1} + \frac{p'_i}{2},$$

so

$$b < -\lg \left(\frac{p'_{i-1}}{2} + p''_{i-1} + \frac{p'_i}{2} \right) \leq \min(-\lg(p'_{i-1} + p'_i) + 1, -\lg p''_{i-1}).$$

Therefore the least common ancestor of the $(i-1)$st and ith leaves of T is of depth less than

$$\min(-\lg(p'_{i-1} + p'_i) + 1, -\lg p''_{i-1}).$$

Note that this is the $(i-1)$st node in T that has two children.

Similarly, for $1 \leq i < n$, the least common ancestor of the ith and $(i+1)$st leaves of T is of depth less than

$$\min(-\lg(p'_i + p'_{i+1}) + 1, -\lg p''_i).$$

Note that this is the ith node in T that has two children.

Let T' be the compressed binary trie for the binary representations of the fractions $q_1, \ldots, q_n$. The ith internal node of T' has depth at most as great as the ith node with two children in T. Therefore, the ith internal node of T' is of depth less than $\min(-\lg(p'_i + p'_{i+1}) + 1, -\lg p''_i)$.

The ith leaf of T' is a child of either the $(i-1)$st or ith internal node of T'. Therefore, the ith leaf of T' is of depth less than

$$\max\left(\min(-\lg(p'_{i-1} + p'_i) + 1, -\lg p''_{i-1}), \min(-\lg(p'_i + p'_{i+1}) + 1, -\lg p''_i)\right) + 1 \leq -\lg p'_i + 2.$$

Since the depth of the ith leaf is an integer, it is at most $\lceil -\lg p_i \rceil + 1$. □

In the non-alphabetic case, Theorem 1.2 has Corollaries 1.3 and 1.4. In the alphabetic case, we can prove similar corollaries that follow from Theorem 1.7.

Corollary 1.8. *Using a static, alphabetic, length-unrestricted code, S can be encoded using $(H(P)+2)m + O(d \log n)$ bits.*

Proof. By Theorem 1.7, there exists an alphabetic code tree such that, if the relative frequency p_i in S of character i is greater than 0, then the leaf labeled i is of depth at most $\lceil -\lg p_i \rceil + 1$. The average codeword length in an encoding of S using this code is of length at most

$$-\sum_{i=1}^{n} p_i(\lceil \lg p_i \rceil + 1) < H(P) + 2,$$

so the total number of bits used in the m codewords is less than $(H(P)+2)m$. To record the code tree takes $O(d \log n)$ bits. □

Corollary 1.9. *Suppose that $P' = p'_1, p''_1, p'_2, \ldots, p'_{n-1}, p''_{n-1}, p'_n$ is a probability distribution with $p'_i \geq 1/2n$ for $1 \leq i \leq n$. Then there exists a strict binary tree on n leaves with height at most $\lceil \lg n \rceil + 1$, such that the ith internal node is of depth less than $-\lg p''_i$ and the ith leaf is of depth at most $\lceil -\lg p'_i \rceil + 1$.*

Proof. By Theorem 1.7, there exists a strict binary tree T on n leaves such that the ith internal node is of depth less than

$$\min(-\lg p''_i, -\lg(p'_i + p'_{i+1}) + 1).$$

Since $p'_i + p'_{i+1} \geq 1/n$, we have

$$\min(-\lg p''_i, -\lg(p'_i + p'_{i+1}) + 1) \leq \lg n + 1$$

. Since every leaf of T is the child of an internal node, T is of height less than $\lg n + 2$. Therefore T is of height at most $\lceil \lg n \rceil + 1$.

For $1 \leq i \leq n$, the ith leaf is a child of either the $(i-1)$st internal node or the ith internal node. These nodes (if they exist) are of depth less than $-\lg p'_i + 1$. Therefore the ith leaf is of depth at most $\lceil -\lg p'_i \rceil + 1$. □

Even if there exists a binary tree on n leaves of depths $l_1, \ldots, l_n$ and $l'_1, \ldots, l'_n$ is a permutation of $l_1, \ldots, l_n$, there may not exist a binary tree on n leaves of depths $l'_1, \ldots, l'_n$. For example, there is a binary tree with leaves of depths $1, 2, 2$, but there is no binary tree with leaves of depths $2, 1, 2$. However, Theorem 1.7 has the following corollary.

Corollary 1.10. *If $l_1, \ldots, l_n$ is a sequence of integers such that*

$$\sum_{i=1}^{n} 2^{-l_i} \leq \frac{1}{2},$$

then there exists a binary tree on n leaves of depths at most $l_1, \ldots, l_n$.

Proof. Let $P' = p'_1, 0, p'_2, \ldots, p'_{n-1}, 0, p'_n$, where $p'_i = 2^{-l_i}/k$ and

$$k = \sum_{j=1}^{n} 2^{-l_j} \leq \frac{1}{2}.$$

By Theorem 1.7, there exists a binary tree on n leaves such that the ith leaf is of depth at most

$$\lceil -\lg p_i' \rceil + 1 = l_i + \lceil \lg k \rceil + 1 \leq l_i. \qquad \square$$

The following lemma, proven in [AW87] and [Aig88], is a partial analogue to Theorem 1.5.

Lemma 1.11. *Suppose that T is a binary tree on n nodes of depths $l_1, \ldots, l_n$. Then*

$$\sum_{i=1}^{n} 2^{-l_i} \leq \lg(n+1)$$

with equality if and only if $n+1$ is a power of 2 and T is complete.

Proof. By induction on n. If $n = 1$, then the single node in T is of depth 0 and $2^0 = \lg 2$. Now let $n > 1$ and assume the claim is true of all trees on fewer than n nodes. Let n' and n'' be the sizes of the left and right subtrees of T. Then, in the left subtree, the nodes have depths $l_1 - 1, \ldots, l_{n'} - 1$ and, in the right subtree, the nodes have depths $l_{n'+2} - 1, \ldots, l_n - 1$. Therefore,

$$\sum_{i=1}^{n'} 2^{-(l_i - 1)} \leq \lg(n'+1)$$

and

$$\sum_{i=n'+2}^{n} 2^{-(l_i - 1)} \leq \lg(n''+1)$$

with equality in both cases if and only if $n'+1$ and $n''+1$ are powers of 2 and the left and right subtrees of T are complete. Note that $l_{n'+1} = 0$, so

$$\sum_{i=1}^{n} 2^{-l_i} \leq \frac{\lg(n'+1)}{2} + 1 + \frac{\lg(n''+1)}{2}$$

with equality if and only if $n'+1$ and $n''+1$ are powers of 2 and the left and right subtrees of T are complete. Since $n' + n'' = n - 1$,

$$\frac{\lg(n'+1)}{2} + 1 + \frac{\lg(n''+1)}{2}$$

$$\begin{aligned} &= \frac{\lg\left((n'+1)(n''+1)\right)}{2} + 1 \\ &\leq \frac{\lg\left(\left(\frac{n-1}{2}+1\right)^2\right)}{2} + 1 \\ &= \lg(n+1) \end{aligned}$$

with equality if and only if $n' = n''$. Therefore,

$$\sum_{i=1}^{n} 2^{-l_i} \leq \lg(n+1)$$

with equality if and only if $n'+1 = n''+1$ is a power of 2 and the left and right subtrees of T are complete. This is equivalent to $n+1$ being a power of 2 and T being complete. □

Unlike Theorem 1.5, there is no converse to Lemma 1.11. For example, there exists a binary tree with nodes of depths $1, 0, 1$, but no binary tree with nodes of depths $0, 1, 1$.

A slightly stronger but more complicated result has been proven by De Prisco and De Santis [DD96].

Chapter 2

Huffman Coding

In this chapter, we define Huffman trees and Huffman codes and describe static and dynamic Huffman coding. We show that two well-known dynamic Huffman coding algorithms, one by Faller [Fal73], Gallager [Gal78] and Knuth [Knu85]; and another by Vitter [Vit87], can be easily modified to produce $O(\log n)$-restricted codes. This modification only negligibly increases the bound on the length of the encoding and time required to encode (and decode).

2.1 Static Huffman Coding

Given a sequence $P' = p'_1, \ldots, p'_n$, Huffman's algorithm assigns each p'_i as a weight to the root of a 1-node tree. It then repeatedly chooses the two trees whose roots have least weight (breaking ties arbitrarily), and makes them the left and right subtrees of a new node, which is assigned the sum of the weights of its children. This continues until there is only one tree remaining.

Applying Huffman's algorithm [Huf52] to a sequence of non-negative numbers results in a tree with minimum weighted average distance from the root to the leaves. It takes $O(n \log n)$ time. A tree with this property is called a *Huffman tree*. Note that, using this definition, not all Huffman trees for a sequence could be produced by Huffman's

algorithm. For example, for the sequence $1, 1, 1+\epsilon, 1+\epsilon$ with $\epsilon < 1$, the complete binary tree tree of height 2, with its leaves assigned weights $1, 1, 1+\epsilon, 1+\epsilon$ in any order, is a Huffman tree. However, Huffman's algorithm will always make the two leaves with weight 1 siblings.

Huffman's algorithm is often used on *relative frequency distributions*. The relative frequency distribution for the string S is

$$\frac{\#_1(S)}{m}, \ldots, \frac{\#_n(S)}{m}.$$

A tree is a Huffman tree for $w_1, \ldots, w_n$ if and only if it is a Huffman tree for $cw_1, \ldots, cw_n$ for any $c > 0$. Therefore, instead of the relative frequency distribution, it is common to use a list of the frequencies of the character as inputs. This is called a *frequency distribution*. Although Huffman's algorithm can be applied to any sequence, we will always consider the inputs to be a frequency or relative frequency distribution.

Consider a Huffman tree constructed from a frequency distribution or relative frequency distribution on n characters. If the leaves of the Huffman tree are labeled with the n characters, with each character as the label of a leaf whose weight is its relative frequency, the result is a code tree. It is called a *Huffman code*.

If we apply Huffman's algorithm to P, we obtain a code such that the average codeword length in an encoding of S is minimized. By Theorem 1.2, the average codeword length in such an encoding is less than $H(P) + 1$. Therefore the m codewords take fewer than $(H(P) + 1)m$ bits overall.

In a Huffman code, a character with relative frequency p' may be assigned codeword of length greater than $\lceil -\lg p' \rceil$. However, the following theorem by Katona and Nemetz [KN76] shows that such a codeword is of length $O(-\log p')$.

Theorem 2.1 (Katona and Nemetz, 1976). *In a Huffman tree for a relative frequency distribution, any node assigned weight at least $1/F_{i+2}$ is of depth at most i, where F_{i+2} is the $(i+2)$nd Fibonacci number.*

The Fibonacci numbers grow exponentially, with $F_i = \Theta(\phi^i)$, where $\phi \approx 1.62$ is the golden ratio. Therefore, a leaf assigned weight p' is of depth $O(-\log_\phi p') = O(-\log p')$. This also gives us the following corollary.

Corollary 2.2. *Suppose that T is a Huffman tree for a relative frequency distribution $P' = p'_1, \ldots, p'_n$. Then for all p'_i in $1/n^{O(1)}$, any leaf assigned weight p'_i is of depth $O(\log n)$.*

2.2 Dynamic Huffman Coding

In this section, we describe two algorithms for dynamic coding based on Huffman's algorithm: one presented independently by Faller [Fal73] and Gallager [Gal78], and refined by Knuth [Knu85]; and another by Vitter [Vit87]. Because of their close relation to Huffman's algorithm, they are called *dynamic Huffman coding*.

For $0 < i < m$, the Faller-Gallager-Knuth (FGK) algorithm and Vitter's algorithm encode s_{i+1} using a Huffman code for the frequencies of the characters in the string S_i, possibly modified to include an extra leaf. The extra leaf is for the case that s_{i+1} does not occur in S_i: the sequence of edge labels on the path from the root to the extra leaf is a special codeword, indicating that the next $\lceil \lg n \rceil$ bits represent s_{i+1}, unencoded. Therefore the extra leaf is treated as the root of an implicit subtree of height $\lceil \lg n \rceil$ on n leaves.

After encoding s_i, instead of computing a Huffman tree from scratch for the frequencies of characters in S_{i+1}, the FGK algorithm and Vitter's algorithm increment the weight of the leaf labeled s_i in the Huffman tree that they already have and then update this tree so that it becomes a Huffman tree again. The key to quickly computing the sequence of code trees for $S_1, S_2, \ldots, S_m$ is to maintain a particular ordering on the nodes of each tree.

Theorem 2.3 (Gallager, 1978). *Suppose that a strict binary tree T has weights assigned to its leaves and each internal node is assigned the sum of the weights of its children. Then T is a Huffman tree for its leaf weights, if all nodes can be placed in order of non-decreasing weight such that siblings are adjacent and children precede parents.*

Let u be the leaf labeled s_i. Once the weight of u has been incremented, it is necessary to increment the weights of its ancestors. The FGK algorithm and Vitter's algorithm begin at u and move upward to the root, incrementing each ancestor as it is visited.

If an ancestor v of u appears later in the ordering than all other nodes that have the same weight as v, then the algorithms increment the weight of v without changing the ordering. This is because the weights assigned to the leaves are all frequencies, so they are integers. Therefore, the weight of the node following v in the ordering must be at least 1 greater than the weight of v.

If an ancestor v of u appears earlier in the ordering than some other node that has the same weight as v, then they change the ordering before incrementing the weight of v. Let v' be the last node in the ordering that has the same weight as v. They transpose v and v' in the ordering. The problem is that v and v' may no longer be adjacent to their siblings in the ordering. Therefore, in the tree, they interchange the subtrees rooted at v and v' as well.

If one wanted to decrement the weight of a leaf, a similar procedure could be used.

Knuth [Knu85] gives an implementation that allows incrementing and decrementing any weight, and updating the tree accordingly. The time required to perform this operation is proportional to the depth of the updated leaf. The ordering is represented as a doubly-linked list, with every node v pointing to its predecessor and successor in the ordering, the first and last nodes in the ordering having the same weight as v, and its parent and children in the Huffman tree.

The extra leaf is treated as having frequency 0. After the first occurrence of character a in S, the extra leaf is normally expanded to an internal node with two leaves as children.

One of the leaves is the new extra leaf, the other is labeled a. The frequency of a is then incremented as usual. In the case that all other characters have occurred in S, the extra leaf is not expanded; it is just labeled a. Notice that the code trees are strictly binary.

Although Knuth did not analyze the overall efficiency of his algorithm, Vitter showed that it uses at most approximately twice as many bits as static Huffman coding. A tighter upper bound was recently proven using amortized analysis [MLP99].

Theorem 2.4 (Milidiú, Laber, and Pessoa, 1999). *The FGK algorithm encodes S using fewer than $2m$ more bits and the same time to within a constant factor as static Huffman coding, when the latter includes the cost of recording the Huffman code.*

Vitter's algorithm uses a similar approach, but with additional restrictions on the Huffman tree at each step. This results in a slightly better upper bound on the length of the encoding, and makes Vitter's algorithm easier to analyze than the FGK algorithm.

Theorem 2.5 (Vitter, 1985). *Vitter's algorithm encodes S using fewer than m more bits and the same time to within a constant factor as encoding S using static Huffman coding, when the latter includes the cost of recording the Huffman code.*

The following lemma shows that when the FGK algorithm or Vitter's algorithm is applied to a string whose length is in $n^{O(1)}$, they produce an encoding in which the length of every codeword is in $O(\log n)$.

Lemma 2.6. *If m is in $n^{O(1)}$ and the FGK algorithm or Vitter's algorithm is applied to S, then every character of S is encoded using $O(\log n)$ bits.*

Proof. Consider any character s_i of S. Let T be the code tree that results after processing S_{i-1}. Every leaf of T is either labeled with a character that occurs in S_{i-1} or is the extra leaf. Any leaf labeled with a character that occurs in S_{i-1} has weight at least $1/(i-1)$, which is in $1/n^{O(1)}$. Therefore, by Corollary 2.2, every such leaf is of depth $O(\log n)$.

If s_i does not occur in S_{i-1}, then it is encoded as the labels on the path to the extra leaf, plus $\lceil \lg n \rceil$ more bits. Since the code tree is strict, the extra leaf is also of depth $O(\log n)$. Therefore s_i is encoded using $O(\log n)$ bits. □

2.3 Length-Restriction

In this section, we use Corollary 2.2 to show that the FGK algorithm and Vitter's algorithm can be easily modified to produce $O(\log n)$-restricted codes. This modification only negligibly increases the bound on the number of bits and time required.

Theorem 2.7. *The FGK algorithm and Vitter's algorithm can be modified to encode S using $O(\log n)$ bits for each character; $O(m)$ more bits overall and the same time to within a constant factor as static Huffman coding, when the latter includes the cost of recording the Huffman tree.*

Proof. If m is in $n^{O(1)}$, then we encode m using dynamic Huffman coding, via the FGK algorithm or Vitter's algorithm. By Theorems 2.4 and 2.5, we use the same time to within a constant factor as static Huffman coding, when the latter includes the cost of recording the Huffman tree. By Lemma 2.6, every codeword in the encoding of S is of length $O(\log n)$.

Otherwise, we partition S into substrings of length L or $L-1$, where L is in $\Theta(n \log n)$. We then encode each substring separately using dynamic Huffman coding, via the FGK algorithm or Vitter's algorithm. Finally, we concatenate the encodings of the substrings; the resulting string is an encoding of S. By Lemma 2.6, every codeword in the encoding of S is of length $O(\log n)$. It remains to be shown that we use $O(m)$ more bits overall and the same time to within a constant factor as static Huffman coding, when the latter includes the cost of recording the Huffman tree.

First, we compare the performance of our new algorithm to the performance of applying static Huffman coding to each substring, and recording each code tree. For each

substring, by Theorems 2.4 and 2.5, we use $O(L)$ more bits and the same time to within a constant factor. Therefore, in total, we use $O(m)$ more bits and the same time to within a constant factor.

Next, we compare the performance of applying static Huffman coding to each substring and recording each code tree, to the performance of applying static Huffman coding to the entire string S.

Suppose that Huffman's algorithm produces a code C_1 when applied to the relative frequencies of the characters in a substring S' of S, and a code C_2 when applied to the relative frequencies of characters in S. Using C_1 minimizes the average length of the codewords in the encoding of S', so, ignoring the overhead of recording C_1, encoding S' using C_1 requires no more bits bits than using C_2.

The Huffman tree for each substring can be computed in $O(L + n \log n)$ time and recorded using $O(n \log n)$ bits. Since L is in $\Theta(n \log n)$, using static Huffman coding to encode each of the substrings separately, and recording a Huffman tree for each substring, we would use $O(L)$ more bits and time for each substring than using static Huffman coding to encode the entire string S. This uses $O(m)$ more bits and the same time to within a constant factor as applying static Huffman coding to the entire string S.

Therefore, we use $O(m)$ more bits overall and the same time to within a constant factor as static Huffman coding, when the latter includes the cost of recording the Huffman tree. □

Increasing the length of the substrings decreases the difference between the upper bounds on the number of bits used by the modified algorithm, and the upper bounds on the number of bits used by the FGK algorithm or Vitter's algorithm. However, increasing the length of the substrings increases the upper bound on the maximum codeword length.

An advantage of the modification we have presented is that, if the relative frequencies of the characters are significantly different in different parts of the string, the modified algorithm may outperform both static and dynamic Huffman coding.

Unlike the FGK algorithm and Vitter's algorithm, this modification can guarantee $O(\log n)$-restricted coding. However, it cannot guarantee $(\lg n + O(1))$-restricted coding. Since the Fibonacci numbers grow exponentially with base ϕ, it only guarantees

$$\log_\phi 2 \cdot \lg(n \lg n) + \lceil \lg n \rceil \approx 2.44 \lg n + O(\log \log n)$$

restricted coding. The additional $\lceil \lg n \rceil$ term is included because, to specify a character that has not previously occurred, $\lceil \lg n \rceil$ bits are sent after the special codeword labeling the path from the root to the extra leaf.

In the next chapter, we present algorithms for dynamic, $(\lceil \lg n \rceil + 1)$-restricted coding.

Chapter 3

$(\lceil \lg n \rceil + 1)$-Restricted Coding

In this chapter, we present an algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted, non-alphabetic coding. This algorithm is based on a data structure called a dynamic minimax tree. It uses fewer than $(H(P) + 4.5)m + d \lg n$ bits to encode S and $O\left((H(P) + 1)m + d \log^2 n\right)$ time overall to encode and decode. We also give an algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted, alphabetic coding.

3.1 Dynamic Minimax Trees

Suppose that we alter Huffman's algorithm so that, instead of making the weight of a node the sum of the weights of its children, its weight is the maximum of their weights plus 1.

This algorithm was presented by Golumbic [Gol76, Knu98a], and the resulting tree minimizes the maximum sum of any leaf's weight and its depth. A tree with this property is called a *minimax tree* [KK85] for the input weight sequence. This maximum sum is called the *minimax cost* of the input, and is the weight assigned to the root of the tree by Golumbic's algorithm. Just as not all Huffman trees can be produced by Huffman's algorithm, not all minimax trees can be produced by Golumbic's algorithm.

Note that the minimax cost of a sequence of integers $-l_1, \ldots, -l_n$ is at most 0 if and

only if there exists a binary tree on n leaves that, in some order, are of depths $l_1, \ldots, l_n$. In this case, a minimax tree for $-l_1, \ldots, -l_n$ is such a tree. Therefore, by Theorem 1.5, the minimax cost of $l_1, \ldots, l_n$ is at most 0 if and only if

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1.$$

Since the algorithms for constructing Huffman trees and minimax trees are so similar, it is not surprising that an analogue of Theorem 2.3 holds for minimax trees. The proof is similar.

Theorem 3.1. *Suppose that a strict binary tree T has weights assigned to its leaves and that each internal node of T is assigned the maximum of the weights of its children plus 1. Also suppose that the nodes of T can be placed in order of non-decreasing weight, such that siblings are adjacent. Then T is a minimax tree for its leaf weights.*

Proof. By induction. The theorem is trivially true if T consists of a single leaf. Suppose that it is true for all strict binary trees on at most $k \geq 1$ leaves, and that T has $k + 1$ leaves. Let R be an ordering of the nodes of T in non-decreasing order by weight, such that siblings are adjacent.

Let u and v be the first two nodes in R. Because T is strict, u has a sibling, which must be v, since no other node is adjacent to u in R. Because any child of u or v would have smaller weight and would, therefore, appear earlier in the ordering, u and v are leaves. Let t be the parent of u and v in T.

Let T' be the strict binary tree that results from removing u and v from T. Then t is a leaf in T'. Every node in T' has the same weight that it has in T, so each internal node of T' has weight equal to the maximum of the weights of its children plus 1. Also, by removing u and v from R, we obtain an ordering in which the nodes of T' are in non-decreasing order by weight and siblings are adjacent. Therefore, by the inductive assumption, T' is a minimax tree for its leaf weights.

Suppose that T is not a minimax tree. Apply Golumbic's algorithm to T's leaf weights and obtain a minimax tree M for T's leaf weights. Since u and v have least weight, we can assume that they are siblings in M. Removing u and v from M gives us a tree M' with the same leaf weights as T' but the same weight assigned to the root as M.

T and M have the same leaf weights, T is not a minimax tree for its leaf weights, and M is. Therefore, the weight assigned to the root of T is greater than the weight assigned to the root of M. This implies that the weight assigned to the root of T' is greater than the weight assigned to the root of M'. This contradicts the fact that T' is a minimax tree for its leaf weights. □

Knuth's implementation of dynamic Huffman trees, described in Section 2.2, can be easily adapted for making minimax trees dynamic. This implementation maintains a binary tree T with weights assigned to its leaves, such that each internal node of T is assigned the maximum of the weights of its children plus 1, and the nodes of T can be placed in order of nondecreasing weight. Therefore, by Theorem 3.1, T is always a minimax tree. As in Knuth's implementation of dynamic Huffman trees, this implementation allows the weights of leaves to be incremented and decremented by 1 in time proportional to the depth of the updated leaf. The only significant modification is that if we increment or decrement the weight of a node v, we may not have to increment or decrement the weight of v's parent, since the maximum of v's weight and its sibling's weight may not change.

In the case that weights always lie in a range of size $\lceil \lg n \rceil + 1$, we can use a simpler implementation than Knuth's. We keep the nodes in an array of size $2n-1$. We maintain two arrays, each containing $\lceil \lg n \rceil + 1$ indices. The ith indices in these arrays are to the first and last nodes of weight w, where w is the ith weight in the range.

3.2 Dynamic Minimax Coding

In this section, we describe how minimax trees can be used for static, $(\lceil \lg n \rceil + 1)$-restricted, non-alphabetic coding. Using dynamic minimax trees, we obtain an algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted, non-alphabetic coding. First, we show that Golumbic's algorithm can be used to construct code trees for static, length-restricted codes.

Lemma 3.2. *Suppose that $P' = p'_1, \ldots, p'_n$ is a probability distribution. Then the minimax cost of the weight sequence*

$$-\min(\lceil -\lg p'_1 \rceil + 1, \lceil \lg n \rceil + 1), \ldots, -(\min(\lceil -\lg p'_n \rceil + 1, \lceil \lg n \rceil + 1)$$

is at most 0.

Proof. Corollary 1.4 implies that there exists a binary tree of height at most $\lceil \lg n \rceil + 1$ on n leaves that, in some order, are of depths $l_1, \ldots, l_n$, where $l_i \leq \lceil -\lg p'_i \rceil + 1$. We can assign the ith weight in the sequence, $-\min(\lceil -\lg p'_i \rceil + 1, \lceil \lg n \rceil + 1)$, to the leaf of depth l_i. Then the sum of this leaf's depth and its weight is at most

$$l_i - \min(\lceil -\lg p'_i \rceil + 1, \lceil \lg n \rceil + 1) \leq 0. \qquad \square$$

Consider a minimax tree for the weight sequence

$$-\min(\lceil -\lg p_1 \rceil + 1, \lceil \lg n \rceil + 1), \ldots, -\min(\lceil -\lg p_n \rceil + 1, \lceil \lg n \rceil + 1).$$

For $1 \leq i \leq m$, suppose that the leaf assigned the ith weight is of depth l_i. Label this leaf with the character i. Then this tree is a code tree for a $(\lceil \lg n \rceil + 1)$-restricted code, since $l_1, \ldots, l_n \leq \lceil \lg n \rceil + 1$. The average codeword length in an encoding of S using this code is

$$\sum_{i=1}^{n} p_i l_i \leq \sum_{i=1}^{n} p_i(\lceil -\lg p_i \rceil + 1) < -\left(\sum_{i=1}^{n} p_i \lg p_i\right) + 2 = H(P) + 2.$$

In Chapter 2, we were able to use dynamic Huffman trees to obtain an algorithm for dynamic coding. Similarly, we can use dynamic minimax trees to obtain an algorithm for dynamic, length-restricted coding. After encoding a prefix S_i of S, we want any character with high relative frequency in S_i to be the label of a leaf of small depth. To ensure this, we maintain two invariants. First, immediately after we encode S_i, if a leaf is labeled with a character a, then it has weight at least

$$-\min\left(\left\lceil -\lg\left(\frac{\#_a(S_i)}{i}\right)\right\rceil + 2, \lceil \lg n \rceil + 1\right).$$

This guarantees that, if a leaf is labeled with a character that has a high relative frequency in S_i, then it has weight close to 0. Second, the minimax cost of the sequence of weights assigned to the leaves is always at most 0. This guarantees that, if a leaf has weight close to 0, then it is of small depth. We will describe a simple approach that maintains the first of these invariants but not the second. We will then describe how it can be modified so that both invariants are maintained.

As we encode (or decode), we maintain a dynamic minimax tree for a sequence of n weights, using either of the two implementations described in Section 3.1. Each weight is initially $-(\lceil \lg n \rceil + 1)$. Each character of the alphabet labels one leaf of this tree. For $1 \leq i \leq m$, after encoding (or decoding) the ith character s_i of S, we update the weight of the leaf labeled with s_i to be

$$-\min\left(\left\lceil -\lg\left(\frac{\#_{s_i}(S_i)}{i}\right)\right\rceil + 2, \lceil \lg n \rceil + 1\right),$$

where S_i is the prefix of S of length i. If this changes the weight assigned to that leaf, we start at that leaf and move towards the root, updating the weights of internal nodes and possibly interchanging subtrees. However, unlike the FGK algorithm and Vitter's algorithm, we may be able to stop before reaching the root. Suppose that we reach a node t that has weight equal to the maximum of its children's weights plus 1. Then t's weight does not need to be updated and we can stop.

With this approach, the minimax cost of the weight sequence may become greater than 0. The problem is that we only update the weight of a leaf immediately after encoding an occurrence of the character with which it is labeled. If a character has high relative frequency in a prefix of S, then it is assigned weight close to 0. If it does not occur again, then its relative frequency drops, but we do not decrease its weight. If enough leaf weights become large, the minimax cost of the sequence will exceed 0.

For example, consider encoding the string $12^2 3^4 4^8 5^{16} 6^{32} 7^{64} 8^{128}$, drawn from the alphabetic $\{1, \ldots, 8\}$. In the code tree that results after encoding this string, the leaf labeled 1 has weight -2 and the leaves labeled 2 through 8 have weight -3. Applying Golumbic's algorithm shows that the minimax cost of $-2, -3, -3, -3, -3, -3, -3, -3$ is 1.

We modify this algorithm so that it maintains the invariant that the minimax cost of the weights is always at most 0. As we encode (or decode) S, in the background, we cycle through the characters that have already appeared and, if necessary, decrease the weights of the corresponding leaves and update the tree. Specifically, we maintain a queue of the characters that have occurred earlier in S. After encoding a character s_i of S, we update the weight of the leaf labeled s_i and, if s_i does not occur in S_{i-1}, then we enqueue s_i. We then dequeue a character, say a; update the weight of the leaf labeled a; and re-enqueue a. Finally, we update the tree. We will refer to this algorithm as *dynamic minimax coding*.

For $0 \leq i \leq m$, let $W_i = w_{i,1}, \ldots, w_{i,n}$ be the weight sequence created by dynamic minimax coding after encoding or decoding i characters of S. Then, after encoding or decoding i characters of S and updating the tree, we have a minimax tree for W_i. Note that $w_{0,1} = \cdots = w_{0,n} = -(\lceil \lg n \rceil + 1)$.

The value of $w_{i,a}$ depends on n, i, $\#_a(S_i)$, the position j of the last occurrence of a in S_i (if $\#_a(S_i) \geq 1$), and the position k of the character in S_i that was encoded immediately before a was last dequeued (if this has happened). Note that, if $\#_a(S_i) \geq 1$, then the weight of the leaf labeled a was last updated after the algorithm encoded s_j or s_k.

Lemma 3.3. *For $1 \leq i \leq m$, the weight sequence $W_i = w_{i,1}, \ldots, w_{i,n}$ that is maintained by dynamic minimax coding has minimax cost at most 0.*

Proof. By Lemma 3.2, the weight sequence

$$-\min\left(\left\lceil -\lg\left(\frac{\#_1(S_i)}{i}\right)\right\rceil + 1, \lceil \lg n \rceil + 1\right), \ldots,$$
$$-\min\left(\left\lceil -\lg\left(\frac{\#_n(S_i)}{i}\right)\right\rceil + 1, \lceil \lg n \rceil + 1\right)$$

has minimax cost at most 0. Compare this to W_i.

If $\#_a(S_i) = 0$, then the weight of the leaf labeled a has not been changed since it was initially set, so $w_{i,a} = -(\lceil \lg n \rceil + 1)$. If $\#_a(S_i) \geq 1$, then let j be the largest number less than or equal to i such that either s_j is an occurrence of a, or a was dequeued immediately after s_j was encoded. Then the weight of the leaf labeled a was last updated immediately after s_j was encoded.

Since a does not occur after s_j in S_i, we know that $\#_a(S_j) = \#_a(S_i)$. Immediately after s_j was encoded, there could be at most j characters in the queue, so a was re-enqueued in position at most j. One character was dequeued after each of s_{j+1} through s_i. Therefore, since a has not been dequeued since then, $i < 2j$. This implies that

$$\begin{aligned} w_{i,a} &= -\min\left(\left\lceil -\lg\left(\frac{\#_a(S_j)}{j}\right)\right\rceil + 2, \lceil \lg n \rceil + 1\right) \\ &\leq -\min\left(\left\lceil -\lg\left(\frac{\#_a(S_i)}{i}\right)\right\rceil + 1, \lceil \lg n \rceil + 1\right). \end{aligned}$$

Therefore, the minimax cost of W_i is at most 0. □

From Lemma 3.3, immediately after we encode s_i, the leaf labeled a is of depth at most $-w_{i,a}$. Since $w_{i,1}, \ldots, w_{i,n} \geq -(\lceil \lg n \rceil + 1)$, the height of the dynamic minimax tree maintained by dynamic minimax coding is at most $\lceil \lg n \rceil + 1$. This means that dynamic minimax coding is ($\lceil \lg n \rceil + 1$)-restricted. Note that s_i is encoded using at most

$$\left\lceil -\lg\left(\frac{\#_{s_i}(S_{i-1})}{i-1}\right)\right\rceil + 2$$

bits.

3.3 Analysis

In Section 3.1, we proved that dynamic minimax coding has certain properties. In this section, we prove that any algorithm with these properties encodes S using fewer than $(H(P) + 4.5)m + d \lg n$ bits. We then prove that dynamic minimax coding uses $O\left((H(P) + 1)m + d \log^2 n\right)$ time overall to encode and decode.

Theorem 3.4. *Any algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted coding that encodes each character s_i using at most*

$$\left\lceil -\lg\left(\frac{\#_{s_i}(S_{i-1})}{i-1}\right)\right\rceil + 2$$

bits, for $1 < i \leq m$, encodes S using fewer than $(H(P) + 4.5)m + d \lg n$ bits.

Proof. If s_i does not occur in S_{i-1}, then

$$\left\lceil -\lg\left(\frac{\#_{s_i}(S_{i-1})}{i-1}\right)\right\rceil + 2$$

is infinite, so s_i is encoded using at most $\lceil \lg n \rceil + 1$ bits.

Now suppose that s_i does occur occur in S_{i-1}. Then the algorithm encodes s_i using at most

$$\left\lceil -\lg\left(\frac{\#_{s_i}(S_{i-1})}{i-1}\right)\right\rceil + 2 < -\lg(\#_{s_i}(S_{i-1})) + \lg(i-1) + 3$$

bits. Since $i - 1 < m$, we know that $\lg(i-1) < \lg m$, so

$$\begin{aligned}
&-\lg(\#_{s_i}(S_{i-1})) + \lg(i-1) + 3 \\
&< -\lg(\#_{s_i}(S_{i-1})) + \lg m + 3 \\
&= -\lg\left(\frac{\#_{s_i}(S)}{m}\right) + \lg\left(\frac{\#_{s_i}(S)}{\#_{s_i}(S_{i-1})}\right) + 3 \\
&= -\lg p_{s_i} + \lg\left(\frac{\#_{s_i}(S)}{\#_{s_i}(S_{i-1})}\right) + 3.
\end{aligned}$$

If s_i is the jth occurrence of the letter a, with $j \geq 2$, then it is encoded using fewer than

$$-\lg p_a + \lg\left(\frac{\#_a(S)}{\#_a(S_{i-1})}\right) + 3 = -\lg p_a + \lg\left(\frac{\#_a(S)}{j-1}\right) + 3$$

bits. Therefore, the total number of bits used to encode all occurrences of a, except the first, is less than

$$\begin{aligned}
&\sum_{j=2}^{\#_a(S)} \left(-\lg p_a + \lg \left(\frac{\#_a(S)}{j-1} \right) + 3 \right) \\
&= (\#_a(S) - 1)\left(-\lg p_a + \lg(\#_a(S)) + 3\right) - \lg\left((\#_a(S) - 1)!\right).
\end{aligned}$$

Note that, for $x \geq 1$, $x! > (x/e)^x$, where $e \approx 2.71$ is the base of the natural logarithm [Rob55]. Therefore,

$$\lg\left((\#_a(S) - 1)!\right) > (\#_a(S) - 1)\left(\lg(\#_a(S) - 1) - \lg e\right),$$

so the total number of bits used to encode all occurrences of a, except the first, is less than

$$(\#_a(S) - 1)\left(-\lg p_a + 3 + \lg e + \lg\left(\frac{\#_a(S)}{\#_a(S) - 1} \right) \right).$$

Since, for $x > 0$,

$$x \lg\left(\frac{x+1}{x} \right) = \lg\left(1 + \frac{1}{x} \right)^x < \lg e,$$

the total number of bits used to encode all occurrences of a, except the first, is less than

$$(\#_a(S) - 1)(-\lg p_a + 3 + \lg e) + \lg e.$$

In total, S is encoded using fewer than

$$\begin{aligned}
&\sum_{a \in S} (\lceil \lg n \rceil + 1 + \lg e + (\#_a(S) - 1)(-\lg p_a + 3 + \lg e)) \\
&= d(\lceil \lg n \rceil + 1 + \lg e) - \left(\sum_{a \in S} \#_a(S) \lg p_a \right) + \left(\sum_{a \in S} \lg p_a \right) + (m - d)(3 + \lg e) \\
&< (H(P) + 4.5)m + d \lg n
\end{aligned}$$

bits. □

We now consider the time used by dynamic minimax coding.

Theorem 3.5. *Dynamic minimax coding uses* $O\left((H(P) + 1)m + d \log^2 n\right)$ *time to encode and decode* S.

Proof. After encoding each character s_i of S, we update the weight of the leaf labeled s_i, from w_{i-1,s_i} to w_{i,s_i}. While we do this, we ensure that the tree remains a minimax tree. If $w_{i-1,s_i} < w_{i,s_i}$, then we increase the weight by repeatedly incrementing it, as described in Section 3.1. Incrementing a leaf's weight from w to $w+1$ takes time proportional to its depth, which is bounded above by $-w$. Thus, incrementing the weight of the leaf labeled s_i from w_{i-1,s_i} to w_{i,s_i} takes $O(t_i)$ time, where

$$t_i = -\sum_{j=w_{i-1,s_i}}^{w_{i,s_i}-1} j.$$

If $w_{i-1,s_i} = w_{i,s_i}$, then we do not change the weight. In this case, let $t_i = 0$. If $w_{i-1,s_i} > w_{i,s_i}$, then we decrease the weight by decrementing it, as described in Section 3.1.

Suppose that s_i does not occur in S_{i-1}. Then $w_{i-1,s_i} = -(\lceil \lg n \rceil + 1)$ and $w_{i,s_i} = -\min(\lceil \lg i \rceil + 2, \lceil \lg n \rceil + 1)$. Therefore, $t_i < (\lceil \lg n \rceil + 1) \lg n$. Since there are d distinct characters in S,

$$\sum_{s_i \notin S_{i-1}} t_i < d(\lceil \lg n \rceil + 1) \lg n.$$

Now suppose that s_i occurs in S_{i-1}. Then

$$\#_{s_i}(S_i) = \#_{s_i}(S_{i-1}) + 1 \leq 2 \cdot \#_{s_i}(S_{i-1}),$$

so the relative frequency of this character in S_i is less than twice its relative frequency in S_{i-1}. Therefore, w_{i,s_i} is at most $w_{i-1,s_i} + 1$, so $t_i \leq w_{i-1,s_i}$.

Suppose that an algorithm for dynamic coding encodes each character s_i using exactly $-w_{i-1,s_i}$ bits. If $\#_{s_i}(S_{i-1}) = 0$, then $w_{i-1,s_i} = -(\lceil \lg n \rceil + 1)$. Otherwise, by the definition of the weight sequence W_{i-1},

$$w_{i-1,s_i} = -\min\left(\left\lceil -\lg\left(\frac{\#_{s_i}(S_j)}{j}\right)\right\rceil + 2, \lceil \lg n \rceil + 1\right),$$

where $j \leq i-1$ is the position of the last character in S_{i-1} after which the weight of the leaf labeled s_i is updated by dynamic minimax coding. Therefore,

$$w_{i-1,s_i} \geq -\min\left(\left\lceil -\lg\left(\frac{\#_{s_i}(S_{i-1})}{i-1}\right)\right\rceil + 2, \lceil \lg n \rceil + 1\right).$$

By Theorem 3.4, this algorithm uses fewer than $(H(P) + 4.5)m + d \lg n$ bits to encode S, so

$$-\sum_{i=1}^{m} w_{i-1,s_i} < (H(P) + 4.5)m + d \lg n.$$

Therefore, in total, we spend

$$O\left(\sum_{i=1}^{m} t_i\right) \subseteq O\left(d \log^2 n + \sum_{i=1}^{m} w_{i-1,s_i}\right) \subseteq O\left((H(P) + 1)m + d \log^2 n\right)$$

time incrementing leaves' weights.

Decrementing a leaf's weight from $w + 1$ to w takes time proportional to its depth, which is bounded above by $-w$. Since every leaf's weight is initially $-(\lceil \lg n \rceil + 1)$ and is never less than this, for each decrement of a leaf's weight from $w + 1$ to w, there must have been a corresponding preceding increment from w to $w + 1$. Therefore, in total, we spend

$$O\left(\sum_{i=1}^{m} t_i\right) \subseteq O\left((H(P) + 1)m + d \log^2 n\right)$$

time decrementing leaves' weights. □

More careful analysis shows that the bound in Theorem 3.5 can be improved to

$$O\left((H(P) + 1)m + \min(d \log^2 n, n \log n)\right).$$

3.4 Alphabetic Coding

In this section, we give an algorithm for dynamic, ($\lceil \lg n \rceil + 1$)-restricted, alphabetic coding. This algorithm uses $(H(P) + 4.5)m + d \lg n$ bits and $O(m \log n)$ time overall. Thus, the bound on the number of bits used by this algorithm is the same as on the number of bits used by dynamic minimax coding, but the bound on the time used by this algorithm is worse. We also suggest modifications that might improve our algorithm's time complexity.

There is an ordered analogue of minimax trees, leaf-oriented alphabetic minimax trees [KK85, CKP86], which we discuss in Chapter 5. However, in that chapter, we also

prove that there is no equivalent of Theorem 3.1 that applies to leaf-oriented alphabetic minimax trees. Therefore, we use a different approach for alphabetic coding. Specifically, we use an implicit representation of a dynamic code tree. The following construction is similar to that in the proofs of Theorem 1.7 and Corollary 1.9.

First, we specify the string that will be used to encode S. For $1 < i \leq m$ and $1 \leq j \leq n$, let

$$q_{i,j} = \left(\sum_{k=1}^{j-1} \frac{\#_k(S_{i-1})}{2(i-1)} \right) + \frac{\#_j(S_{i-1})}{4(i-1)} + \frac{j}{2n}.$$

For $1 < i \leq m$ and $1 < j \leq n$, suppose that the binary representations of $q_{i,j-1}$ and $q_{i,j}$ agree on their first b bits. Then

$$2^{-b} > q_{i,j} - q_{i,j-1} = \frac{\#_{j-1}(S_{i-1}) + \#_j(S_{i-1})}{4(i-1)} + \frac{1}{2n},$$

so

$$\begin{aligned} b &< -\lg \left(\frac{\#_{j-1}(S_{i-1}) + \#_j(S_{i-1})}{4(i-1)} + \frac{1}{2n} \right) \\ &\leq \min \left(-\lg \left(\frac{\#_{j-1}(S_{i-1})}{i-1} \right) + 2, -\lg \left(\frac{\#_j(S_{i-1})}{i-1} \right) + 2, \lg n + 1 \right). \end{aligned}$$

Let $x_{i,j}$ be the longest common prefix of the binary representations of $q_{i,j-1}$ and $q_{i,j}$. Then

$$|x_{i,j}| < \min \left(\left\lceil -\lg \left(\frac{\#_{j-1}(S_{i-1})}{i-1} \right) \right\rceil + 2, \left\lceil -\lg \left(\frac{\#_j(S_{i-1})}{i-1} \right) \right\rceil + 2, \lceil \lg n \rceil + 1 \right).$$

Also, since $q_{i,j-1} < q_{i,j}$, we have that $x_{i,j}0$ is a prefix of the binary representation of $q_{i,j-1}$ and $x_{i,j}1$ is a prefix of the binary representation of $q_{i,j}$.

For $1 \leq j \leq n$, let $z_{1,j}$ be the $\lceil \lg n \rceil$-bit binary representation of j. For $1 < i \leq m$ and $1 \leq j \leq n$, let $z_{i,j}$ be the binary string consisting of the first

$$\min \left(\left\lceil -\lg \left(\frac{\#_j(S_{i-1})}{i-1} \right) \right\rceil + 2, \lceil \lg n \rceil + 1 \right)$$

bits of the binary representation of $q_{i,j}$. Notice that, for $1 < j \leq n$, we have that $x_{i,j}0$ is a prefix of $z_{i,j-1}$ and $x_{i,j}1$ is a prefix of $z_{i,j}$. Therefore, the set of strings $\{z_{i,1}, \ldots, z_{i,n}\}$

is lexicographically increasing and self-delimiting, so the binary trie for $z_{i,1}, \ldots, z_{i,n}$ is an alphabetic code tree of height at most $\lceil \lg n \rceil + 1$. Note that, given S_{i-1}, it is possible to compute this code tree. Therefore, it is possible to compute the encoding $z_{1,s_1} \ldots z_{m,s_m}$ of S from S and to compute S from $z_{1,s_1} \ldots z_{m,s_m}$. By Theorem 3.4, $z_{1,s_1} \ldots z_{m,s_m}$ is of length less than $(H(P) + 4.5)m + d \lg n$.

Theorem 3.6. *There is a dynamic, $(\lceil \lg n \rceil + 1)$-restricted, alphabetic coding algorithm that encodes and decodes S in $O(m \log n)$ time.*

Proof. We give an algorithm for dynamic, $(\lceil \lg n \rceil + 1)$-restricted, alphabetic coding that, given S, computes $z_{1,s_1} \ldots z_{m,s_m}$ in $O(m \log n)$ time, and, given $z_{1,s_1} \ldots z_{m,s_m}$, computes S in $O(m \log n)$ time.

Consider an abstract data type that stores a sequence of integer values $x_1, \ldots, x_n$ and supports the following four operations:

- given a positive integer $i \leq n$, return x_i;
- given a positive integer $i \leq n$, compute the ith prefix sum,

  $$\sum_{j=1}^{i} x_j,$$

  of the sequence of values;
- given an increasing function $f : \mathbb{Z}^3 \to \{0,1\}^*$, where the ordering on $\{0,1\}^*$ is lexicographic, and a binary string y, compute the greatest positive number $i \leq n$ such that

  $$f\left(i, x_i, \sum_{j=1}^{i} x_j\right)$$

  is lexicographically less than or equal to y; and,
- given a positive integer $i \leq n$, increment x_i.

Suppose that we have a data structure D for this abstract data type that implements all of these operations in $O(\log n)$ time, assuming that f can be computed from its arguments in $O(1)$ time.

For $1 \leq i \leq m$, when encoding or decoding s_i, D stores $\#_1(S_{i-1}), \ldots, \#_n(S_{i-1})$. Notice that $\#_{s_i}(S_i) = \#_{s_i}(S_{i-1}) + 1$ and, for $a \neq s_i$, we have $\#_a(S_i) = \#_a(S_{i-1})$. Therefore, by incrementing a single frequency, we can update D so that it stores $\#_1(S_i), \ldots, \#_n(S_i)$ in $O(\log n)$ time.

Given n, i, and a character $j \leq n$, we can compute $q_{i,j}$ and $z_{i,j}$ directly from their definitions using D and then update D, in $O(\log n)$ time. Thus, given $S = s_1, \ldots, s_m$, we can compute the encoding $z_{1,s_1} \ldots z_{m,s_m}$ in $O(m \log n)$ time.

Now suppose that we are given n, i, and y, where y is the first

$$\min\left(|z_{i,s_i} \ldots z_{m,s_m}|, \lceil \lg n \rceil + 1\right)$$

bits of $z_{i,s_i} \ldots z_{m,s_m}$. Since z_{i,s_i} is a self-delimiting prefix of y, s_i is the largest character $j \leq n$ such that $z_{i,j}$ is lexicographically less than or equal to y. Therefore, we can compute s_i and then update D, in $O(\log n)$ time. Thus, given $z_{1,s_1} \ldots z_{m,s_m}$, we can compute S in $O(m \log n)$ time.

We can implement the desired abstract data type as a binary search tree T of height $O(\log n)$ on the keys $1, \ldots, n$. We augment T so that, for $1 \leq i \leq n$, the ith node v stores x_i and the sum of the values in v's subtree.

It is possible to implement the first, second, and fourth of the abstract data type's operations in $O(\log n)$ time using T [CLRS01, Sections 14.1 and 14.2]. We now describe how to implement the third operation in $O(\log n)$ time using T, assuming that the function f can be computed from its arguments in $O(1)$ time.

For $1 \leq h \leq n$, suppose that we are given f, a binary string y, the hth prefix sum of $x_1, \ldots, x_n$, and a pointer to the hth node v of T. Then, since v stores h and x_h, we can

compute

$$f\left(h, x_h, \sum_{j=1}^{h} x_j\right)$$

in $O(1)$ time.

Suppose that v has a left child t and that t has key k. Since T is a binary search tree, it follows that $k < h$ and $k+1, \ldots, h-1$ are the keys stored in t's right subtree. Thus, if $k+1 \leq h-1$, then t has a non-empty right subtree and t's right child stores

$$\sum_{j=k+1}^{h-1} x_j.$$

Note that

$$\sum_{j=1}^{k} x_j = \sum_{j=1}^{h} x_j - x_h - \sum_{j=k+1}^{h-1} x_j,$$

so we can compute it in $O(1)$ time.

Suppose that v has a right child u and that u has key k'. Since T is a binary search tree, it follows that $h < k'$ and $h+1, \ldots, k'-1$ are the keys stored in u's left subtree. Thus, if $h+1 \leq k'-1$, then u has a non-empty left subtree and u's left child stores

$$\sum_{j=h+1}^{k'-1} x_j.$$

Note that

$$\sum_{j=1}^{k'} x_j = \sum_{j=1}^{h} x_j + x_{k'} + \sum_{j=h+1}^{k'-1} x_j,$$

so we can compute it in $O(1)$ time.

Therefore, we can find the largest number $i \leq n$ such that

$$f\left(i, x_i, \sum_{j=1}^{i} x_j\right)$$

is lexicographically less than or equal to y. To do this, we start at the root, and go left or right at the node with key h depending on whether

$$f\left(h, x_h, \sum_{j=1}^{h} x_j\right)$$

is lexicographically greater than or less than y. Since T is of height $O(\log n)$, this takes $O(\log n)$ time. □

Suppose that we modify the algorithm given in the proof of Theorem 3.6 to use an augmented dynamic balanced binary search tree, such as an augmented red-black tree [CLRS01, Chapter 13]. Then, we only need to store those characters that have occurred in the prefix of S that we have already encoded or decoded. This modification makes the algorithm slightly more complicated, but reduces the bound on the total time required to $O(m \log d)$. If every character appears in S at least once, then using an augmented splay tree reduces the bound on the total time required to $O((H(P) + 1)m)$ [ST85]. However, the worst-case bound on the time required to encode or decode a single character and update the tree becomes $\Theta(n)$.

As noted in Chapter 1, there are data structures, such as biased search trees, deepsplay trees, and Iacono's unified structure [Iac01], that have both good worst-case and good amortized bounds for access and update times. We conjecture that, when augmented, these can be used to obtain both good worst-case bounds on the time required to encode or decode a single character and update the data structure and good bounds on the total time required to encode or decode.

Chapter 4

Restructuring Trees

In this chapter, we provide an algorithm for restructuring unordered binary trees to have restricted height, and give new and simple proofs of results by Evans and Kirkpatrick [EK00] for restructuring ordered trees. Evans and Kirkpatrick's proofs were based around leaf-oriented and node-oriented alphabetic minimax trees, which we define in Chapter 5.

To *restructure* a binary tree T to have a certain property, we construct another binary tree T' on the same leaves or nodes as T, such that T' has the desired property. We consider both the case when the order of the leaves or nodes can change, and when it cannot.

4.1 Restructuring Unordered Trees

In this section, we show that it is easy to restructure an unordered binary tree to have nearly the same weighted average distance from the root to the leaves, but height at most $\lceil \lg n \rceil + 1$. In fact, we show something stronger: that it is possible to restructure an unordered strict binary tree to have height at most $\lceil \lg n \rceil + 1$ such that no node increases in depth by more than 1. The results in this section are similar to those obtained by Milidiú and Laber [ML97] for restructuring Huffman trees.

Lemma 4.1. *Suppose that we are given a strict binary tree T on n leaves. Then there exists a strict binary tree T' on n leaves of height at most $\lceil \lg n \rceil + 1$ and there exists a bijection f from the nodes of T to the nodes of T' that maps leaves to leaves such that, for all $v \in T$, the depth of $f(v)$ in T' is at most 1 greater than the depth of v in T.*

Proof. If there are no internal nodes in T of depth at least $\lceil \lg n \rceil$, then T' can be chosen to be identically T, and f can be chosen to be the identity function.

Otherwise, replace each internal node of depth $\lceil \lg n \rceil$ by one of its descendants which is a leaf. The result is a strict binary tree of height $\lceil \lg n \rceil$. Let this be the left subtree of T'. Let V be the set of nodes of T that are not in the left subtree of T' and let v be a node in V that is an internal node in T. Make v the root of T'.

A binary tree is strict if and only if the number of leaves is one more than the number of internal nodes. Since T and the left subtree of T' are strict, the number of nodes in $V - \{v\}$ that are internal nodes in T is one less than the number of nodes in $V - \{v\}$ that are leaves in T. Therefore there exists a strict binary tree of height at most $\lceil \lg n \rceil$ whose nodes are $V - \{v\}$, whose leaves are leaves in T, and whose internal nodes are internal nodes in T. Let this be the right subtree of T'.

T' is of height $\lceil \lg n \rceil + 1$. Every node in T''s left subtree is of depth 1 greater in T' than it is in T. The root v of T' and every node in T''s right subtree were of depth at least $\lceil \lg n \rceil$ in T and, therefore, are of depth at most 1 greater in T' than in T. □

Note that Corollary 1.4 follows immediately from Theorem 1.2 and Lemma 4.1.

Using the same approach as in the proof of Lemma 4.1, it is possible to $(\lceil \lg n \rceil + k)$-restrict an unordered tree T, for $k > 1$, such that only the proper descendants of one node u increase in depth, where u is of depth $k - 1$. The nodes that do increase in depth do so by at most 1.

To do this, we replace every internal node of depth $\lceil \lg n \rceil + k - 1$ by one of the leaves in its subtree. Let T' be the resulting tree. We choose u to be some internal node of T' of depth $k - 1$. Let T'' be the subtree rooted at u in T'.

Let V be the set of nodes of T that are not in T'. We replace u by some node $v \in V$ that is an internal node in T, and then make T'' the left subtree of u. As in the proof above, there exists a strict binary tree of height at most $\lceil \lg n \rceil$ whose nodes are $V' - \{v\}$. We make this the right subtree of v.

Lemma 4.1 can be applied to obtain dynamic, length-restricted, non-alphabetic coding. For example, in Chapter 1, we briefly described an algorithm due to Bentley, Sleator, Tarjan, and Wei [BSTW86]. This algorithm uses a move-to-front heuristic to maintain a list of the characters which have occurred. The codeword for a particular symbol which has occurred is an Elias code representation of its position in the list. By Lemma 4.1, we can construct a $(\lceil \lg n \rceil + 1)$-restrict code such that, for $1 \leq i \leq n$, the codeword for i in this new code is of length at most 1 greater than the Elias code representation of i.

Using this $(\lceil \lg n \rceil + 1)$-restricted code instead of an Elias code produces an encoding of S that is at most m bits longer. This bound is not quite as good as the bound on the number of bits used by static or dynamic Huffman coding, or by the coding algorithms we presented in Chapters 2 and 3. However, empirical studies have shown that Bentley, Sleator, Tarjan, and Wei's algorithm performs better in practice than static or dynamic Huffman coding, because, usually, characters that have occurred recently are more likely to occur next.

4.2 Restructuring Leaf-Oriented Binary Search Trees

In this section, we give a new proof of a recent theorem by Evans and Kirkpatrick [EK00]. Our proof relies on Theorems 1.5 and 1.7, which are well-known. Assuming these results, our proof is somewhat simpler than Evans and Kirkpatrick's. The results in this section are similar to those obtained by Laber, Milidiú, and Laber [LMP99] for restructuring alphabetic code trees.

Theorem 4.2 (Evans and Kirkpatrick, 2000). *Suppose that we are given a leaf-*

oriented binary search tree on n leaves of depths $l_1, \ldots, l_n$. Then there exists a leaf-oriented binary search tree on n leaves of height at most $\lceil \lg n \rceil + 1$ such that the ith leaf is of depth at most $l_i + 2$.

Proof. Consider the probability distribution $P' = p'_1, 0, p'_2, \ldots, p'_{n-1}, 0, p'_n$, where

$$p'_i = \frac{1}{2}\left(\frac{2^{-l_i}}{\sum_{j=1}^{n} 2^{-l_j}} + \frac{1}{n}\right).$$

By Theorem 1.5,

$$\sum_{j=1}^{n} 2^{-l_j} \leq 1.$$

Therefore, by the definition of p'_i,

$$\lceil -\lg p'_i \rceil + 1 = \left\lceil -\lg\left(\frac{2^{-l_i}}{\sum_{j=1}^{n} 2^{-l_j}} + \frac{1}{n}\right)\right\rceil + 2 \leq l_i + 2.$$

Corollary 1.9 implies the existence of a leaf-oriented binary search tree on n leaves of height at most $\lceil \lg n \rceil + 1$ such that the ith leaf is of depth at most $l_i + 2$. □

A different algorithm for restructuring leaf-oriented binary search trees to have restricted height, and a bound on the inefficiency of optimal height-restricted leaf-oriented binary search trees, is given by Laber, Milidiú and Pessoa [LMP99]. However, for that algorithm, the probability assigned to each leaf must be known.

4.3 Restructuring Binary Search Trees

Evans and Kirkpatrick [EK00] proved that, given a strict binary tree T on n leaves such that the ith node is of depth l_i, there exists a strict binary tree T' on n leaves of height at most $\lceil \lg n \rceil + 1$ such that the ith node of T' is of depth at most $l_i + \lfloor \lg \lg n \rfloor$. In this section, we give a simple proof of a slightly weaker result.

Theorem 4.3. *Suppose that we are given a strict binary tree T on n leaves such that the ith node is of depth l_i. Then there exists a strict binary tree T' on n leaves of height at most $\lceil \lg n \rceil + 1$, such that the ith node T' is of depth at most $l_i + \lceil \lg \lg n \rceil$.*

Proof. Let $l'_1, \ldots, l'_n$ be the depths of the leaves of T, and let $l''_1, \ldots, l''_{n-1}$ be the depths of the internal nodes, so that

$$l'_1, l''_1, l'_2, \ldots, l'_{n-1}, l''_{n-1}, l'_n = l_1, \ldots, l_{2n-1}.$$

By Lemma 1.11,

$$\sum_{i=1}^{2n-1} 2^{-l_i} \leq \lg n + 1.$$

Since T is strict, by Theorem 1.5,

$$\sum_{i=1}^{n} 2^{-l'_i} = 1.$$

Hence

$$\sum_{i=1}^{n-1} 2^{-l''_i} = \sum_{i=1}^{2n-1} 2^{-l_i} - \sum_{i=1}^{n} 2^{-l'_i} \leq \lg n.$$

Consider the probability distribution $P' = 1/2n, p''_1, 1/2n, \ldots, 1/2n, p''_{2n-1}, 1/2n$, where

$$p''_i = \frac{2^{-l''_i}}{2\sum_{j=1}^{n-1} 2^{-l''_j}} \geq \frac{2^{-(l''_i+1)}}{\lg n} > 0.$$

By Corollary 1.9, there exists a strict binary tree on n leaves of height at most $\lceil \lg n \rceil + 1$ such that the ith internal node is of depth less than

$$-\lg p''_i \leq l''_i + \lg\lg n + 1.$$

Therefore, the ith internal node is of depth at most $l''_i + \lceil \lg\lg n \rceil$. Let $l''_0 = l''_n = -\infty$. The ith leaf of T is a descendant of the $(i-1)$st and ith internal nodes of T (if they exist), and the child of one of these, so $l'_i = \max\{l''_{i-1}, l''_i\} + 1$. Similarly, the ith leaf of T' is of depth at most

$$\max(l''_{i-1} + \lceil \lg\lg n \rceil, l''_i + \lceil \lg\lg n \rceil) + 1 = l'_i + \lceil \lg\lg n \rceil.$$ □

4.4 Restructuring Optimal Binary Search Trees

Suppose that we are given an extended binary search tree T with n leaves and a probability distribution P' over the nodes of T. The *cost* of T is the expected depth of a node

chosen according to P'. Suppose that T is *optimal*; that is, it has the minimum cost of any binary search tree over the same set of nodes.

For any $k \geq 1$, we can restructure T so that it is of height $O(\log n + \log k)$ and the increase in its cost is less than $1/k$, even when we do not know P'. We use the following lemma and corollary, which show that subtrees rooted at deep nodes in an optimal extended binary search tree do not have large total probabilities. The *total probability* of a subtree is the sum of the probabilities of the nodes in that subtree.

Lemma 4.4. *For any node v of depth at least 2 in an optimal extended binary search tree, the total probability of v's subtree is at most half the total probability of v's grandparent's subtree.*

Proof. Let u be v's grandparent. We prove by contradiction that the total probability of v's subtree is at most half the total probability of u's subtree. Suppose not. We consider four cases.

If v is the left child of a left child, then we can rotate u right, as shown in Figure 4.1. This rotation decreases by 1 the depths of v's parent, v, and every proper descendant of v; increases by 1 the depths of u and nodes in u's right subtree; and leaves the depths of nodes in v's parent's right subtree unchanged. Therefore the cost of T is reduced. However, this is contrary to the assumption that T is optimal. The case in which v is the right child of a right child is symmetric.

If v is the right child of a left child, then we can rotate v's parent left and then rotate u right, as shown in Figure 4.2. These rotations decrease by 2 the depth of v and by 1 the depths of v's proper descendants; increase by 1 the depths of u and nodes in u's right subtree; and leave the depths of v's parent and nodes in v's parent's left subtree unchanged. Therefore the cost of T is reduced. However, this is contrary to the assumption that T is optimal. The case in which v is the left child of a right child is symmetric.

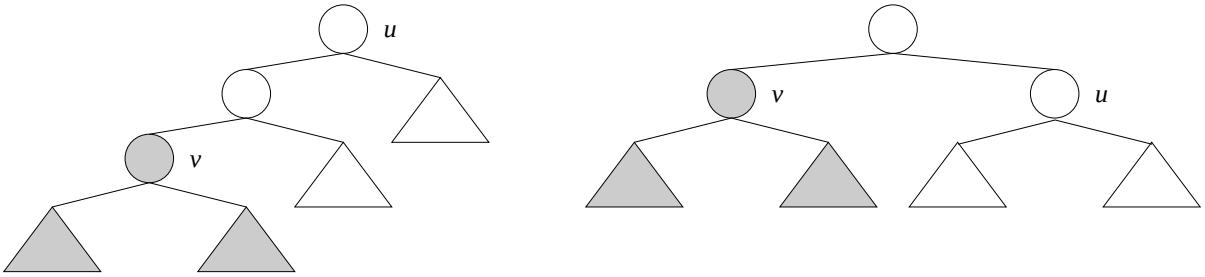


Figure 4.1: Rotating u right.

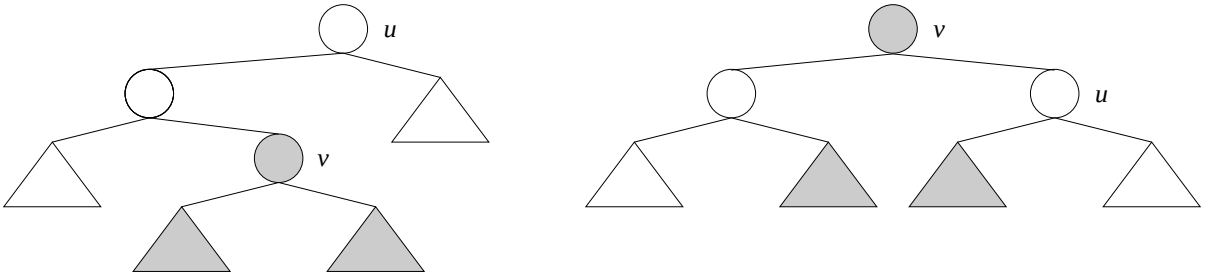


Figure 4.2: Rotating v's parent left and u right.

Therefore the subtree rooted at v has total probability at most half the total probability of the subtree rooted at u. □

Corollary 4.5. *For any node v of depth l in an optimal extended binary search tree, the total probability of v's subtree is at most $2^{-\lfloor l/2 \rfloor}$.*

Proof. By induction on l. If $l = 0$ or $l = 1$, then the total probability of v's subtree is at most $2^{-\lfloor l/2 \rfloor} = 1$. Otherwise, assume that the total probability of v's grandparent's subtree is at most $2^{-\lfloor (l-2)/2 \rfloor}$. Then, by Lemma 4.4, the total probability of v's subtree is at most $2^{-\lfloor l/2 \rfloor}$. □

Using Corollary 4.5, it is easy to prove the following theorem.

Theorem 4.6. *For any optimal binary search tree T on n nodes and any $k \geq 1$, there exists a binary search tree T' of height less than $3 \lg n + 2 \lg k + 2 \lg \lg n + 2$ whose cost is*

at most $1/k$ plus the cost of T.

Proof. First, we truncate T at depth $2\lceil \lg n + \lg k + \lg\lg n \rceil$). By Corollary 4.5, the total probability of each of the removed subtrees is at most

$$\frac{1}{2^{\lceil \lg n + \lg k + \lg\lg n \rceil}} \leq \frac{1}{nk \lg n}.$$

Since there are at most n removed subtrees, the sum of the total probabilities of these subtrees is at most $1/k \lg n$.

Next, we replace each removed subtree by a subtree with on the same nodes, in the same order, but height at most $\lfloor \lg n \rfloor$. Let T' be the resulting tree. Notice that T' is of height at most

$$\lfloor \lg n \rfloor + 2\lceil \lg n + \lg k + \lg\lg n \rceil < 3 \lg n + 2 \lg k + 2 \lg\lg n + 2.$$

For any node v, the depth of v in T' is at most $\lfloor \lg n \rfloor$ greater than the depth of v in T. Let c be the sum of the probabilities of all nodes whose depths are greater in T' than in T. Then $c \leq 1/(k \lg n)$. Therefore the cost of T' at most $c\lfloor \lg n \rfloor \leq 1/k$ plus the cost of T. □

It is possible to obtain slightly better bounds, improving $3 \lg n + 2 \lg k + 2 \lg\lg n + 2$ to $3 \lg n + 2 \lg k + 2 \lg\lg\lg n + O(1)$. We can do this by restructuring subtrees in T as described in Section 4.3 instead of replacing them as in the proof of given here. We have presented this proof of a slightly weaker result because it depends only on Lemma 4.4 and Corollary 4.5.

4.5 Reordering Binary Search Trees

Suppose that we are given an extended binary search tree T on n leaves together with a list of its $2n - 1$ nodes in non-decreasing order by probability, according to some unknown probability distribution P'. Here, the internal nodes of T store keys and its

leaves correspond to failed searches. In this section, we show that, from this information, we can construct an extended binary search tree on the same nodes as T whose cost under P' is within $\lg\lg n + 3$ of optimal.

Let $v_1, \ldots, v_{2n-1}$ be the nodes of T. Consider any binary tree T' consisting of $v_1, \ldots, v_{2n-1}$ in some order, such that, for $0 \leq i < \lfloor \lg(2n-1) \rfloor$, the nodes with ranks from 2^i through $2^{i+1} - 1$ are of depth i in T', and any remaining nodes are of depth $\lfloor \lg(2n-1) \rfloor$. Then T' has minimum expected depth when nodes are chosen according to P'.

The problem with T' is that it may not be an extended binary search tree. The following theorem implies that we can restructure T' so that it is strictly binary, keys are in the correct order, and every node increases in depth by less than $\lg\lg n + 3$. Thus, the resulting tree is an extended binary search tree on the same nodes as T, whose height is less than $\lfloor \lg(2n-1) \rfloor + \lg\lg n + 3$ and whose cost is within $\lg\lg n + 3$ of optimal.

Theorem 4.7. *For any binary tree T on $2n-1$ nodes of depths $l_1, \ldots, l_{2n-1}$ and any permutation π on $2n-1$ elements, there exists a strict binary tree T' on $2n-1$ nodes of depths less than $l_{\pi(1)} + \lg\lg n + 3, \ldots, l_{\pi(2n-1)} + \lg\lg n + 3$.*

Proof. By Lemma 1.11,

$$\sum_{i=1}^{2n-1} 2^{-l_i} \leq \lg n + 1.$$

Consider the probability distribution $P'' = p''_1, \ldots, p''_{2n-1}$, where

$$p''_i = \frac{2^{-l_{\pi(i)}}}{\sum_{j=1}^{2n-1} 2^{-l_j}} \geq \frac{2^{-l_{\pi(i)}}}{\lg n + 1} > 0.$$

By Theorem 1.7, there exists a strict binary tree T' on $2n-1$ nodes such that the ith node is of depth at most

$$\lceil -\lg p''_i \rceil + 1 \leq l_{\pi(i)} + \lceil \lg(\lg n + 1) \rceil + 1 < l_{\pi(i)} + \lg\lg n + 3. \qquad \square$$

Chapter 5

Alphabetic Minimax Trees

In Chapter 3, we considered the problem of constructing an unordered binary tree whose leaf depths are bounded by a given multiset of integers, if such a tree exists. To do this, we multiply each integer in the multiset by -1 and use Golumbic's algorithm to construct a minimax tree for the resulting multiset. In Chapter 4, we considered several instances of the problem of constructing a leaf-oriented or node-oriented binary search tree whose leaf depths or node depths are bounded by a given sequence of integers. We only considered instances in which Theorem 1.7 or Corollary 1.9 guarantees that such a tree exists. However, there are sequences that bound the leaf depths or node depths of a binary tree, but that do not satisfy the conditions of Theorem 1.7 or Corollary 1.9. Consider the example we gave in Chapter 1: neither Theorem 1.7 nor Corollary 1.9 guarantees the existence of a tree on leaves of depths $1, 2, 2$.

In this chapter, we consider the problem of constructing an ordered binary tree whose leaves or nodes are bounded by a given sequence of integers, if such a tree exists. Clearly, this generalizes the problem considered in Chapter 4. It also generalizes the problem considered in Chapter 3. To see why, suppose that we are given a multiset of integers M. Let L be the sequence consisting of the integers in M sorted into non-increasing order. Then, by Corollary 1.6, if there exists an unordered binary tree whose leaf depths are

bounded by M, then there exists an ordered binary tree whose leaf depths are bounded by L.

We define an ordered analogue of minimax trees, leaf-oriented alphabetic minimax trees, and a generalization of leaf-oriented alphabetic minimax trees, node-oriented alphabetic minimax trees. If there exists a binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$, then any leaf-oriented alphabetic minimax tree for $W = w_1, \ldots, w_n$ is such a tree. If there exists a binary tree on n nodes of depths at most $-w_1, \ldots, -w_n$, then any node-oriented alphabetic minimax tree for W is such a tree.

Kirkpatrick and Klawe [KK85] and Coppersmith, Klawe, and Pippenger [CKP86] introduced leaf-oriented alphabetic minimax trees for W and gave $O(n)$-time and $O(n)$-space algorithms for constructing one. They used this to obtain an algorithm that, given a planar circuit with bounded fan-in, constructs an equivalent planar circuit of comparable size and depth but with bounded fan-in and fan-out. They considered trees of arbitrary constant degree; however, we consider only binary trees. Under this restriction, their algorithms are the same.

Evans and Kirkpatrick introduced node-oriented alphabetic minimax trees for W and gave an $O(n)$-time and $O(n)$-space algorithm for constructing one. They also showed that the problem of constructing a leaf-oriented alphabetic minimax tree for W can be reduced, in $O(n)$ time, to the problem of constructing a node-oriented alphabetic minimax tree for another sequence of at most n integers.

We give a new $O(n)$-time algorithm for constructing a node-oriented alphabetic minimax tree, and prove it correct. Our algorithm has the advantages that it is somewhat simpler than Evans and Kirkpatrick's, it works incrementally (so it is easy to update the tree after appending an integer to W), it is easy to update the tree after deleting the last integer in W, and it uses $O(h)$ work space, where h is the height of the resulting tree. We also prove that there is no efficient dynamic data structure that explicitly represents leaf-oriented alphabetic minimax trees.

5.1 Leaf-Oriented Alphabetic Minimax Trees

Given a binary tree T on n leaves of depths $l_1, \ldots, l_n$ and a sequence $W = w_1, \ldots, w_n$ of integers, define the *leaf-oriented alphabetic minimax cost of T with respect to W* to be

$$\alpha(T; W) = \max_{1 \le i \le n} \{w_i + l_i\}.$$

Define the *leaf-oriented alphabetic minimax cost of W* to be

$$\alpha(W) = \min\{\alpha(T; W) \mid T \text{ is a binary tree on } n \text{ leaves}\}.$$

If $\alpha(T; W) = \alpha(W)$, then T is called a *leaf-oriented alphabetic minimax tree for W*.

If $\alpha(W) \le 0$, then, by definition, any leaf-oriented alphabetic minimax tree T for W has its leaves at depths at most $-w_1, \ldots, -w_n$. Conversely, if $\alpha(W) > 0$, then there is no binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$. Hence, to learn whether there exists a binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$, it suffices to compute $\alpha(W)$. If such a tree exists, then any leaf-oriented alphabetic minimax tree for W is such a tree.

We can bound the leaf-oriented alphabetic minimax cost of a sequence from below. Consider a leaf-oriented alphabetic minimax tree T for W. Let $l_1, \ldots, l_n$ be the depths of the leaves in T. By definition,

$$\alpha(W) = \max_{1 \le i \le n} \{w_i + l_i\}.$$

By Theorem 1.5 and the definition of $\alpha(W)$,

$$1 \ge \sum_{i=1}^{n} 2^{-l_i} \ge 2^{-\alpha(W)} \sum_{i=1}^{n} 2^{w_i}.$$

Since $\alpha(W)$ is an integer,

$$\alpha(W) \ge \left\lceil \lg \left(\sum_{i=1}^{n} 2^{w_i} \right) \right\rceil.$$

We can also bound the leaf-oriented alphabetic minimax cost of W from above. Consider the probability distribution $P' = p'_1, 0, p'_2, 0, \ldots, p'_n$, where

$$p'_i = \frac{2^{w_i}}{\sum_{j=1}^{n} 2^{w_j}}.$$

By Theorem 1.7, there exists a binary tree T' on n leaves of depths $l'_1, \ldots, l'_n$, where

$$l'_i \leq \lceil -\lg p'_i \rceil + 1 = \left\lceil -\lg \left(\frac{2^{w_i}}{\sum_{j=1}^{n} 2^{w_j}} \right) \right\rceil + 1 = -w_i + \left\lceil \lg \left(\sum_{j=1}^{n} 2^{w_j} \right) \right\rceil + 1.$$

Therefore,

$$\alpha(W) \leq \alpha(T'; W) = \max_{1 \leq i \leq n} \{w_i + l'_i\} \leq \left\lceil \lg \left(\sum_{j=1}^{n} 2^{w_j} \right) \right\rceil + 1.$$

5.2 Node-Oriented Alphabetic Minimax Trees

The definition of node-oriented alphabetic minimax trees is similar to that of leaf-oriented alphabetic minimax trees. The difference is that all nodes are considered, instead of just leaves. This affects the bounds we can obtain.

Given a binary tree T on n nodes of depths $l_1, \ldots, l_n$ and a sequence $W = w_1, \ldots, w_n$ of integers, define the *node-oriented alphabetic minimax cost of T with respect to W* to be

$$A(T; W) = \max_{1 \leq i \leq n} \{w_i + l_i\}.$$

Define the *node-oriented alphabetic minimax cost of W* to be

$$A(W) = \min\{A(T; W) \mid T \text{ is a binary tree on } n \text{ nodes}\}.$$

If $A(T; W) = A(W)$, then T is called a *node-oriented alphabetic minimax tree for W*. Define the node-oriented alphabetic minimax cost of the empty sequence to be $-\infty$ and the node-oriented alphabetic minimax tree for the empty sequence to be the empty tree.

Notice that $A(W) \geq A(w_1, \ldots, w_{n-1})$. To see why, let T be a node-oriented alphabetic minimax tree for W and let T' be the binary tree obtained by replacing the nth node v of T with v's left subtree. Then

$$A(W) = A(T; W) \geq A(T'; w_1, \ldots, w_{n-1}) \geq A(w_1, \ldots, w_{n-1}).$$

Therefore, appending integers to a sequence cannot decrease the node-oriented alphabetic minimax cost.

If $A(W) \leq 0$, then, by definition, any node-oriented alphabetic minimax tree T for W has its nodes at depths at most $-w_1, \ldots, -w_n$. Conversely, if $A(W) > 0$, then there is no binary tree on n nodes of depths at most $-w_1, \ldots, -w_n$. Hence, to learn whether there exists a binary tree on n nodes of depths at most $-w_1, \ldots, -w_n$, it suffices to compute $A(W)$. If such a tree exists, then any node-oriented alphabetic minimax tree for W is such a tree.

We can bound the node-oriented alphabetic minimax cost of W from below. Consider a node-oriented alphabetic minimax tree T for W. Let $l_1, \ldots, l_n$ be the depths of the nodes in T. By definition,

$$A(W) = \max_{1 \leq i \leq n} \{w_i + l_i\}.$$

By Theorem 1.11 and the definition of $A(W)$,

$$\lg(n+1) \geq \sum_{i=1}^{n} 2^{-l_i} \geq 2^{-A(W)} \sum_{i=1}^{n} 2^{w_i}.$$

Since $A(W)$ is an integer,

$$A(W) \geq \left\lceil \lg \left(\sum_{i=1}^{n} 2^{w_i} \right) - \lg \lg(n+1) \right\rceil.$$

We can also bound the node-oriented alphabetic minimax cost of W from above. Consider the probability distribution $P' = 0, p'_1, 0, p'_2, 0, \ldots, p'_n, 0$, where

$$p'_i = \frac{2^{w_i}}{\sum_{j=1}^{n} 2^{w_j}}.$$

By Theorem 1.7, there exists a binary tree T' on n nodes of depths $l'_1, \ldots, l'_n$, where

$$l'_i < -\lg p'_i = -\lg \left(\frac{2^{w_i}}{\sum_{j=1}^{n} 2^{w_j}} \right) = -w_i + \lg \left(\sum_{j=1}^{n} 2^{w_j} \right).$$

Therefore,

$$A(W) \leq A(T'; W) = \max_{1 \leq i \leq n} \{w_i + l'_i\} < \lg \left(\sum_{j=1}^{n} 2^{w_j} \right).$$

Since $A(W)$ is an integer,

$$A(W) \leq \left\lceil \lg \left(\sum_{j=1}^{n} 2^{w_j} \right) \right\rceil - 1.$$

In Section 5.4, we give an algorithm for constructing a node-oriented alphabetic minimax tree for W. The proof of correctness depends on the following property of node-oriented alphabetic minimax cost.

Lemma 5.1. *For any two non-empty sequences of integers W and W', if $A(WW') = A(W)$, then $A(W) > A(W')$.*

Proof. Consider any node-oriented alphabetic minimax tree T for WW'. If the root of T is one of the first n nodes of T, then we iteratively delete the first n nodes of T, as from a binary search tree [CLRS01, pages 262–263]. Let T' be the resulting tree. Notice that, for $1 \leq i \leq |W'|$, the depth of the ith node of T' is at most the depth of the $(n+i)$th node of T minus 1. Therefore,

$$A(W) = A(WW') > A(T'; W') \geq A(W').$$

If the root of T is one of the last $|W'|$ nodes of T, then we iteratively delete the last $|W'|$ nodes of T, as from a binary search tree. Let T' be the resulting tree. Notice that, for $1 \leq i \leq n$, the depth of the ith leaf of T' is at most the depth of the ith node of T minus 1. Therefore,

$$A(W) \leq A(T; W) < A(WW'),$$

which is a contradiction. □

Notice that the contrapositive of Lemma 5.1 implies that, for any two non-empty sequences of integers W and W', if $A(W) \leq A(W')$, then $A(WW') > A(W)$.

5.3 A Reduction

We use slightly different terminology than Evans and Kirkpatrick. They do not refer to node-oriented alphabetic minimax cost. Instead, for any sequence of integers $W = w_1, \ldots, w_n$ of odd length, they define the *full alphabetic minimax cost of* W to be

$$A'(W) = \min\{A(T'; W) \mid T' \text{ is a strict binary tree on } n \text{ nodes}\}.$$

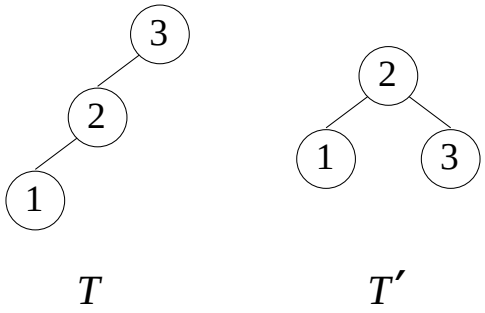


Figure 5.1: $A(T; 1, 2, 3) = 3$ and $A'(1, 2, 3) = A(T'; 1, 2, 3) = 4$.

Accordingly, they define a *full alphabetic minimax tree for* W to be a strict binary tree T' such that $A(T'; W) = A'(W)$.

Since our definition allows all binary trees, if $A'(W)$ is defined, then $A(W) \leq A'(W)$. This inequality is sometimes strict. For example, the only strict binary tree on 3 nodes consists of a root and its two children, so $A'(1, 2, 3) = 4 > A(1, 2, 3)$. This is shown in Figure 5.1.

Suppose that we are given a sequence of integers $W' = w'_1, \ldots, w'_n$, where n is odd, and asked to construct a full alphabetic minimax tree for W'. If $n = 1$, then let T' be the tree consisting of only a root. If $n > 1$, then we show that this problem reduces to the problem of constructing a node-oriented alphabetic minimax tree for $W = w_2, w_4, \ldots, w_{n-1}$, where $w_i = \max(w'_{i-1} + 1, w'_i, w'_{i+1} + 1)$. Note that W can be computed from W' in $O(n)$ time. First, we show that $A'(W') \geq A(W)$. Then we show that, for any node-oriented alphabetic minimax tree T for W, if T' is the strict binary tree obtained by extending T, then $A(W) \geq A(T'; W')$. Since $A(T'; W') \geq A'(W')$, it follows that $A(T'; W') = A'(W')$. Therefore, T' is a full alphabetic minimax tree for W'.

Consider any full alphabetic minimax tree T' for W'. Let $l'_1, \ldots, l'_n$ be the depths of the nodes in T'. Then, for $1 \leq i \leq n$, we have $w'_i + l'_i \leq A'(W')$. For each even i with $1 < i < n$, the ith node of T' is an internal node and an ancestor of the $(i-1)$st and

$(i+1)$st nodes, which are leaves. Therefore,

$$\begin{aligned} l'_i &\leq \min(l'_{i-1} - 1, A'(W') - w'_i, l'_{i+1} - 1) \\ &\leq \min(A'(W') - w'_{i-1} - 1, A'(W') - w'_i, A'(W') - w'_{i+1} - 1) \\ &= A'(W') - \max(w'_{i-1} + 1, w'_i, w'_{i+1} + 1) \\ &= A'(W') - w_i. \end{aligned}$$

Let T be the tree obtained by removing the leaves of T'. Then

$$A(W) \leq A(T; W) = \max(w_2 + l'_2, w_4 + l'_4, \ldots, w_{n-1} + l'_{n-1}) \leq A'(W').$$

Now, consider any node-oriented alphabetic minimax tree T for W. Let $l_2, l_4, \ldots, l_{n-1}$ be the depths of the nodes in T. Then, for each even i with $1 < i < n$,

$$A(W) \geq w_i + l_i = \max(w'_{i-1} + l_i + 1, w'_i + l_i, w'_{i+1} + l_i + 1).$$

Notice that this means that, for each odd i with $1 < i < n$,

$$A(W) \geq \max(w'_i + l_{i+1} + 1, w_i + l_{i-1} + 1) = w'_i + \max(l_{i-1}, l_{i+1}) + 1.$$

Let T' be the strict binary tree obtained by extending T, as described in Section 1.1. Let $l'_1, \ldots, l'_n$ be the depths of the nodes in T'. Then $l'_1 = l_2 + 1$ and $l'_n = l_{n-1} + 1$, so $w'_1 + l'_1 = w'_1 + l_2 + 1 \leq A(W)$ and $w'_n + l'_n = w'_n + l'_{n-1} + 1 \leq A(W)$. For each even i with $1 < i < n$, we have $l'_i = l_i$, so $w'_i + l'_i = w'_i + l_i \leq A(W)$. For each odd i with $1 < i < n$, we have $l'_i = \max(l_{i-1}, l_{i+1}) + 1$, so $w'_i + l'_i = w'_i + \max(l_{i-1}, l_{i+1}) + 1 \leq A(W)$. Therefore, $A(W) \geq A(T'; W')$.

5.4 Construction

In this section, we give a simple incremental algorithm for constructing a node-oriented alphabetic minimax tree for $W = w_1, \ldots, w_n$. This algorithm uses $O(n)$ time and $O(h)$ work space, where h is the height of the resulting tree. We also prove this algorithm

correct. First, we define a kind of node-oriented alphabetic minimax tree called a left-biased tree. Then, we show that it is possible to efficiently construct a left-biased tree for $w_1, \ldots, w_n$ given a left-biased tree for $w_1, \ldots, w_{n-1}$. Finally, we show that it possible to efficiently construct a left-biased tree for $w_1, \ldots, w_{n-1}$ given a left-biased tree for $w_1, \ldots, w_n$.

A similar algorithm was given by Nakatsu [Nak91] for determining whether there exists a binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$. His algorithm is less general than ours in that it does not construct such a tree nor does it consider internal nodes. We can use our algorithm to determine whether there exists a binary tree on n nodes of depths at most $-w_1, \ldots, -w_n$, in $O(n)$ time and $O\left(\max_{1 \leq i \leq n}\{-w_i\}\right)$ space. We conjecture that Nakatsu's algorithm can be modified to construct a leaf-oriented alphabetic minimax tree for $w_1, \ldots, w_n$.

Suppose that T is a node-oriented alphabetic minimax tree for $w_1, \ldots, w_n$. For each node v of T, if v is the ith node of T, then we assign v *weight* w_i. We denote v's subtree by T_v and call the weights of the nodes in T_v its *weight sequence*. We *label* v with the node-oriented alphabetic minimax cost of T_v's weight sequence. We call T a *left-biased* tree for $w_1, \ldots, w_n$ if: $n = 0$ and T is empty; or T's left and right subtrees are left-biased trees for their weight sequences and the root r of T is labeled with the leaf-oriented alphabetic minimax cost of r's left subtree's weight sequence followed by r's weight.

Notice that, by Lemma 5.1, for every node v of a left-biased tree for $w_1, \ldots, w_n$, v's label is greater than the node-oriented alphabetic minimax cost of v's right subtree's weight sequence. If v has a right child, then this is equivalent to saying that v's label is greater than v's right child's label.

We now consider how to construct a left-biased tree for $w_1, \ldots, w_n$ from a left-biased tree for $w_1, \ldots, w_{n-1}$. There are two possible cases: when $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$ and when $A(w_1, \ldots, w_n) = A(w_1, \ldots, w_{n-1})$.

Lemma 5.2. *Suppose that* $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$ *and* T *is a left-biased tree*

for $w_1, \ldots, w_{n-1}$. Then $A(w_1, \ldots, w_n) = \max(A(w_1, \ldots, w_{n-1}) + 1, w_n)$ and we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by making T the left subtree of a new node labeled $A(w_1, \ldots, w_n)$ with weight w_n.

Proof. The first $n-1$ nodes of T' are of depth 1 greater in T' than in T and the nth node of T' is of depth 0. Since $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$ and $A(w_1, \ldots, w_n) \geq w_n$,

$$A(T'; w_1, \ldots, w_n) = \max(A(w_1, \ldots, w_{n-1}) + 1, w_n) \leq A(w_1, \ldots, w_n).$$

Therefore, $A(w_1, \ldots, w_n) = \max(A(w_1, \ldots, w_{n-1}) + 1, w_n)$ and T' is a node-oriented alphabetic minimax tree for $w_1, \ldots, w_n$. By construction, T''s left subtree is a left-biased tree for $w_1, \ldots, w_{n-1}$ and T''s right subtree is empty. Therefore, T' is a left-biased tree for $w_1, \ldots, w_n$. □

Lemma 5.3. *Suppose that $A(w_1, \ldots, w_n) = A(w_1, \ldots, w_{n-1})$, T is a left-biased tree for $w_1, \ldots, w_{n-1}$, and the jth node of T is T's root. Then we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by replacing T's right subtree with a left-biased tree for $w_{j+1}, \ldots, w_n$.*

Proof. Since T is left-biased,

$$A(w_1, \ldots, w_j) = A(w_1, \ldots, w_{n-1}) = A(w_1, \ldots, w_n).$$

Since the first j nodes of T' are of the same depth in T' as in T and T''s right subtree is a left-biased tree for $w_{j+1}, \ldots, w_n$,

$$A(T'; w_1, \ldots, w_n) = \max\left(A(w_1, \ldots, w_j), A(w_{j+1}, \ldots, w_n) + 1\right).$$

By Lemma 5.1, $A(w_1, \ldots, w_j) > A(w_{j+1}, \ldots, w_n)$. Therefore,

$$A(T'; w_1, \ldots, w_n) = A(w_1, \ldots, w_j) = A(w_1, \ldots, w_n),$$

so T' is a node-oriented alphabetic minimax tree for $w_1, \ldots, w_n$. By construction, T''s left and right subtrees are left-biased trees for their weight sequences. Therefore, T' is a left-biased tree for $w_1, \ldots, w_n$. □

We now consider replacing subtrees on the rightmost path.

Lemma 5.4. *Suppose that T is a left-biased tree for $w_1, \ldots, w_{n-1}$, v is a node of T such that T_v consists of the kth through $(n-1)$st nodes of T, and $A(w_k, \ldots, w_n) = A(w_k, \ldots, w_{n-1})$. Then we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by replacing T_v with a left-biased tree for $w_k, \ldots, w_n$.*

Proof. Suppose not. Let u be the deepest ancestor of v in T' such that T'_u is not a left-biased tree for its weight sequence. Then u's right subtree is a left-biased tree for its weight sequence.

Suppose that u is the jth node of T' and T'_u consists of the ith through nth nodes of T'. Then, since T is left-biased and by choice of u,

$$A(w_i, \ldots, w_j) = A(w_i, \ldots, w_{n-1}) > A(w_{j+1}, \ldots, w_{n-1}) = A(w_{j+1}, \ldots, w_n).$$

Since the first $j - i + 1$ nodes of T'_u are of the same depth in T'_u as in T_u and u's right subtree is a left-biased tree for $w_{j+1}, \ldots, w_n$,

$$\begin{aligned} & A(T'_u; w_i, \ldots, w_n) \\ &= \max\left(A(w_i, \ldots, w_j), A(w_{j+1}, \ldots, w_n) + 1\right) \\ &= A(w_i, \ldots, w_j) \\ &= A(w_i, \ldots, w_{n-1}) \\ &\leq A(w_i, \ldots, w_n). \end{aligned}$$

Therefore,

$$A(T'_u; w_i, \ldots, w_n) = A(w_i, \ldots, w_j) = A(w_i, \ldots, w_n),$$

so T'_u is a node-oriented alphabetic minimax tree for $w_i, \ldots, w_n$. By construction and choice of u, the left and right subtrees of u in T' are left-biased trees for their weight sequences. Therefore, T'_u is a left-biased tree for $w_i, \ldots, w_n$, which is a contradiction. □

Lemmas 5.2, 5.3, and 5.4 suggest an incremental algorithm for constructing a left-biased tree for $w_1, \ldots, w_n$ We start with a tree consisting of only a root with weight w_1 and then iteratively insert $n-1$ nodes of weights $w_2, \ldots, w_n$. Each inserted node either becomes the root of the new tree or is recursively inserted into the right subtree of the current tree. There are two problems with this approach. First, we have not said how to efficiently determine whether $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$. Second, even if that can be done, this algorithm is slow in the worst case. For example, we make $\Theta(n^2)$ recursive calls to construct a left-biased tree for $1, \ldots, n$. This is because, if we always start at the root and descend until we find a place to make the next insertion, then we may visit the same node in the tree many times.

We now show that we can solve both of these problems by, instead, starting at the rightmost node of the current tree and moving toward the root until we find a place to make the next insertion. We look for a node t on the rightmost path of the current tree such that T_t satisfies Lemma 5.4 and, if t has a right child u, then T_u satisfies Lemma 5.2.

Lemma 5.5. *Suppose that T is a left-biased tree for $w_1, \ldots, w_{n-1}$ and the root t of T has a label greater than w_n. Furthermore, suppose that, if t has a right child u, then t's label is at least 2 greater than u's label. Then $A(w_1, \ldots, w_n) = A(w_1, \ldots, w_{n-1})$.*

Proof. Suppose that t does not have a right child. Let T' be the binary tree obtained by giving t a right child with weight w_n. Then, since t's label $A(w_1, \ldots, w_{n-1})$ is greater than w_n,

$$A(T'; w_1, \ldots, w_n) = \max(A(w_1, \ldots, w_{n-1}), w_n + 1) = A(w_1, \ldots, w_{n-1}).$$

Therefore, $A(w_1, \ldots, w_n) = A(w_1, \ldots, w_{n-1})$.

Now suppose that t has a right child u and t is the jth node of T. Then t's label $A(w_1, \ldots w_{n-1})$ is at least 2 greater than u's label $A(w_{j+1}, \ldots, w_{n-1})$. Let T' be the binary tree obtained by replacing T_u with a new node y' and then making T_u the left subtree of y'.

Since T is left-biased, $A(w_1, \ldots, w_j) = A(w_1, \ldots, w_{n-1})$. Since T_u is a left-biased tree for $w_{j+1}, \ldots, w_{n-1}$ and t's label is greater than w_n,

$$\begin{aligned} & A(T'; w_1, \ldots, w_n) \\ &= \max\left(A(w_1, \ldots, w_j), A(w_{j+1}, \ldots, w_{n-1}) + 2, w_n + 1\right) \\ &= A(w_1, \ldots, w_j) \\ &= A(w_1, \ldots, w_{n-1}). \end{aligned}$$

Therefore, $A(w_1, \ldots, w_n) = A(w_1, \ldots, w_{n-1})$. □

Lemma 5.6. *Suppose that T is a left-biased tree for $w_1, \ldots, w_{n-1}$ and every node v on T's rightmost path whose label is greater than w_n has a right child whose label is exactly 1 less than v's label. Then $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$.*

Proof. Suppose not. Let the jth node v of T be the deepest node such that T_v consists of the ith through $(n-1)$st nodes of T and $A(w_i, \ldots, w_n) = A(w_i, \ldots, w_{n-1})$.

If v's label $A(w_i, \ldots, w_{n-1}) \leq A(w_n) = w_n$, then, by Lemma 5.1, $A(w_i, \ldots, w_n) > A(w_i, \ldots, w_{n-1})$, which is a contradiction.

If v's label is greater than w_n, then v's right child's label $A(w_{j+1}, \ldots, w_{n-1})$ is exactly 1 less than v's label $A(w_i, \ldots, w_{n-1})$. By our choice of v, $A(w_{j+1}, \ldots, w_n) > A(w_{j+1}, \ldots, w_{n-1})$. Therefore, $A(w_{j+1}, \ldots, w_n) \geq A(w_i, \ldots, w_{n-1})$.

Since T is left-biased, $A(w_i, \ldots, w_j) = A(w_i, \ldots, w_{n-1})$. Therefore, $A(w_i, \ldots, w_j) \leq A(w_{j+1}, \ldots, w_n)$ so, by Lemma 5.1, $A(w_i, \ldots, w_n) > A(w_i, \ldots, w_{n-1})$, which is a contradiction. □

Suppose that T is a left-biased tree for $w_1, \ldots, w_n$. Furthermore, suppose that there exists a node t on T's right path such that T_t satisfies Lemma 5.5 and t is the deepest such node. If t has a right child u, then every node v on T_u's right path whose weight is greater than w_n has a right child whose label is exactly 1 less than v's label. Therefore, T_u satisfies Lemma 5.6.

Theorem 5.7. *A node-oriented alphabetic minimax tree for a sequence of integers $W = w_1, \ldots, w_n$ can be constructed in $O(n)$ time and $O(h)$ work space, where h is the height of the resulting tree.*

Proof. We present an incremental algorithm that constructs a left-biased tree T' for $w_1, \ldots, w_n$ and an amortized analysis showing that this algorithm runs in $O(n)$ time. At any time, we only store in work space the nodes on the current tree's rightmost path and a pointer to the current tree's rightmost node.

Let $n > 1$ and assume that:

- we have constructed a left-biased tree T for $w_1, \ldots, w_{n-1}$;
- we have at least l credits saved, where l is the depth of the $(n-1)$st node y of T;
- we only have in work space the nodes on the rightmost path of T and a pointer to y; and
- we have charged at most $2n-2$ credits and used $O(h)$ work space while constructing T.

We charge 2 credits to construct T' from T. Starting at y, we move towards the root of T until either we find a subtree of T that satisfies Lemma 5.5 or we reach the root of T. We consider the three possible cases:

- Suppose that T_y satisfies Lemma 5.5 and consists of the ith through $(n-1)$st nodes of T. Then $A(w_i, \ldots, w_n) = A(w_i, \ldots, w_{n-1})$ so, by Lemma 5.4, we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by giving y a right child y' labeled $A(w_n) = w_n$ with weight w_n. We store in work space y' and a pointer to y', and remove from work space the pointer to y. Notice that we do not need to update any other node's label. We spend 1 credit to construct T' from T and save 1 credit.
- Suppose that we find a proper ancestor t of y such that t is the jth node of T, T_t satisfies Lemma 5.5, and T_t consists of the ith through $(n-1)$st nodes of T. Then

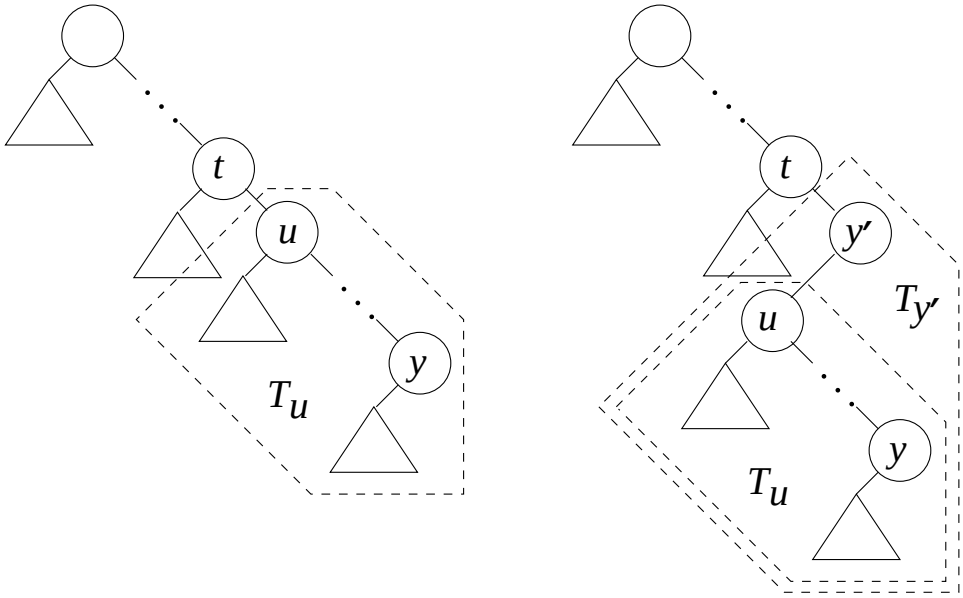


Figure 5.2: Inserting y' into T.

$A(w_i, \ldots, w_n) = A(w_i, \ldots, w_{n-1})$. Also, t has a right child u such that T_u satisfies Lemma 5.6, so $A(w_{j+1}, \ldots, w_n) > A(w_{j+1}, \ldots, w_{n-1})$.

By Lemma 5.2, $A(w_{j+1}, \ldots, w_n) = \max(A(w_{j+1}, \ldots, w_{n-1}) + 1, w_n)$ and we can construct a left-biased tree $T_{y'}$ for $w_{j+1}, \ldots, w_n$ by making T_u the left subtree of a new node y' labeled $A(w_{j+1}, \ldots, w_n)$ with weight w_n. Since u's label is $A(w_{j+1}, \ldots, w_{n-1})$, we can easily compute y''s label. By Lemma 5.4, we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by making y' the right child of t. The result of these operations is shown in Figure 5.2.

Notice that we do not need to update any other node's label. We store in work space y' and a pointer to y', and remove from work space the nodes on the path from y to u and the pointer to y. Since we move from y to t and then insert y' between u and t, we spend $l - l' + 2$ credits to construct T' from T, where l' is the depth of y' in T'.

- Suppose that we reach the root u of T without finding a subtree that satisfies Lemma 5.5. Then u satisfies Lemma 5.6, so $A(w_1, \ldots, w_n) > A(w_1, \ldots, w_{n-1})$.

  By Lemma 5.2, $A(w_1, \ldots, w_n) = \max(A(w_1, \ldots, w_{n-1}) + 1, w_n)$ and we can construct a left-biased tree T' for $w_1, \ldots, w_n$ by making T the left subtree of a new node y' labeled $A(w_1, \ldots, w_n)$ with weight w_n. Since u's label is $A(w_1, \ldots, w_{n-1})$, we can easily compute y''s label.

  Notice that we do not need to update any other node's label. We store in work space y' and a pointer to y', and remove from work space all nodes except y' and the pointer to y. Since we move from y to u and then insert y' above u, we spend $l + 1$ credits to construct T' from T.

In each case, we construct a left-biased tree T' for $w_1, \ldots, w_n$. After constructing T', we have at least l' credits saved, where l' is the depth of y' in T'. Also, we only store in work space the nodes on the rightmost path of T' and a pointer to y'.

Since, in total, we charge $2n$ credits while constructing T', this algorithm uses $O(n)$ time. Since we always only store in work space the nodes on the rightmost path of the current tree, it uses $O(h)$ work space. □

We now consider the problem of updating a left-biased tree $w_1, \ldots, w_n$ after we remove w_n. Suppose that we use the algorithm given above to construct a left-biased tree T' for $w_1, \ldots, w_n$ and, as we do so, we keep a table of pointers to the nodes of T'. Then, we can construct a left-biased tree T for $w_1, \ldots, w_{n-1}$ in $O(1)$ time by replacing the nth node v of T' with v's left subtree.

5.5 A Lower Bound for Dynamic Leaf-Oriented Alphabetic Minimax Trees

We showed in Chapter 3 that it is possible to maintain a dynamic minimax tree T for W such that, if a weight w_i is incremented or decremented, then T can be updated in time proportional to the depth of its ith leaf. This is not possible for alphabetic minimax trees. We give an example in which updating the unique leaf-oriented alphabetic minimax tree for W, after incrementing the weight of a leaf of depth $O(\log n)$ and decrementing the weight of a leaf of depth $O(\log n)$, takes, in total, $\Omega(n)$ time.

Suppose that $n = 2^k - 1$ and W consists of 1 followed by $n-1$ copies of 0. The binary tree T on n leaves of depths $k-1, k, \ldots, k$ has $\alpha(T; W) = k$. In any other binary tree on n leaves, either the first leaf is of depth at least k or some leaf is of depth at least $k+1$. Therefore, T is the only leaf-oriented alphabetic minimax tree for W.

Let W' be a sequence consisting of $n-1$ copies of 0 followed by 1. The binary tree T' on n leaves of depths $k, \ldots, k, k-1$ has $\alpha(T'; W') = k$. In any other binary tree on n leaves, either the nth leaf is of depth at least k or some leaf is of depth at least $k+1$. Therefore, T' is the only leaf-oriented alphabetic minimax tree for W'.

We can obtain W' from W by decrementing w_1 and incrementing w_n. Notice that every leaf-oriented alphabetic minimax tree for the intermediate sequence, which consists of n copies of 0, has height at most k. However, for each odd i with $1 < i < n$, the ith leaf of T is a right child and the ith leaf of T' is a left child. This means that, for $\Omega(n)$ values of i with $1 \leq i < n$, if the ith and $(i+1)$st leaves of T are siblings, then the ith and $(i+1)$st leaves of T' are not.

If we are allowed to construct T' from T by attaching two children to the first leaf of T and deleting the last leaf, for example, then we can do so in $O(\log n)$ time. However, if T and T' must have the same leaves and these must in the same order, then constructing T' from T takes $\Omega(n)$ time.

Chapter 6

Sequence Maxima Trees

In this chapter, we introduce a new abstract data type, a depth sequence, that stores a sequence of non-negative integers $L = l_1, \ldots, l_n$, supports an operation that determines whether or not there exists a binary tree on n leaves of depths at most $l_1, \ldots, l_n$, and supports several operations that update L. Then, we show how depth sequences can be implemented by a new data structure, a sequence maxima tree. Using this implementation, any operation on a depth sequence that stores L takes

$$O\left(\left(\max_{1\leq i\leq n}\{l_i\} - \min_{1\leq i\leq n}\{l_i\} + 1\right)\log n\right)$$

time.

6.1 Depth Sequences

Consider the following abstract data type. A *depth sequence* D stores a sequence of non-negative integers $L = l_1, \ldots, l_n$ and supports the following operations:

- determine whether there exists a binary tree on n leaves of depths at most $l_1, \ldots, l_n$;
- given a positive integer $i \leq n$, return l_i;

- given a non-negative integer l and a positive integer $i \leq n+1$, insert l into the ith position in L;
- given a positive integer $i \leq n$, delete l_i from L;
- given a positive integer $i < n$, split D into depth sequences storing $l_1, \ldots, l_i$ and $l_{i+1}, \ldots, l_n$; and,
- given another depth sequence D' storing L', join D and D' into a depth sequence storing LL'.

As noted in Section 5.1, we can determine whether there exists a binary tree on n leaves of depths at most $l_1, \ldots, l_n$ by determining whether $\alpha(-l_1, \ldots, -l_n) \leq 0$. We want a method of efficiently computing $\alpha(-l_1, \ldots, -l_n)$ when L is updated.

6.2 Sequence Maxima Trees

In this section, we introduce a new data structure, a sequence maxima tree, that we can use to implement a depth sequence. This data structure is based on a recursive algorithm for computing a new function, β, that we define in terms α.

6.2.1 The Function β

Suppose that we are given a sequence of integers $W = w_1, \ldots, w_n$ and an integer a. Then we define $\beta(W; a)$ to be the least positive integer c such that W can be partitioned into c subsequences, each of leaf-oriented alphabetic minimax cost at most a. Notice that, if $a \geq \max_{1 \leq i \leq n}\{w_i\}$, then $\beta(W; a) \leq n$; otherwise, $\beta(W; a)$ is undefined.

Lemma 6.1. *For any sequence of identical integers $W = w_1, \ldots, w_n$ and any integer $a \geq w_1$,*

$$\beta(W; a) = \left\lceil \frac{n}{2^{a-w_1}} \right\rceil.$$

Proof. Notice that $\beta(W;a)$ is the least integer c such that there is a forest of c binary trees on a total of n leaves, each of which is of depth at most $a - w_1$. A binary tree can have 2^{a-w_1} leaves of depth $a - w_1$ but not more. Therefore,

$$\beta(W;a) = \left\lceil \frac{n}{2^{a-w_1}} \right\rceil . \qquad \square$$

Lemma 6.2. *Suppose that a sequence of integers $W = w_1, \ldots, w_n$ contains a proper subsequence $W' = w_i, \ldots, w_j$ of identical integers less than t, where*

$$t = \begin{cases} w_{i-1}, & \text{if } j = n; \\ w_{j+1}, & \text{if } i = 1; \\ \min(w_{i-1}, w_{j+1}), & \text{otherwise.} \end{cases}$$

Let a be an integer such that $\beta(W;a)$ is defined. Let W'' be the sequence obtained from W by replacing W' by $\beta(W';t)$ copies of t. Then $\beta(W;a) = \beta(W'';a)$.

Proof. Notice that W and W'' have the same maximum. Therefore, since $\beta(W;a)$ is defined, $\beta(W'';a)$ is defined.

We now show that $\beta(W;a) \leq \beta(W'';a)$. First, we partition W'' into $\beta(W'';a)$ subsequences, each of leaf-oriented alphabetic minimax cost at most a, and construct a leaf-oriented alphabetic minimax tree for each subsequence. Let F'' be the resulting forest. The first $i-1$ leaves of F'' have weights $w_1, \ldots, w_{i-1}$, the next $\beta(W';t)$ leaves of F'' have weight t, and the last $n-j$ leaves of F'' have weights $w_{j+1}, \ldots, w_n$. Since each of the trees in F'' has leaf-oriented alphabetic minimax cost at most a with respect to the corresponding subsequence of W'', the first $i-1$ leaves of F'' are of depths at most $a - w_1, \ldots, a - w_{i-1}$, the next $\beta(W';t)$ leaves of F'' are of depth at most $a - t$, and the last $n - j$ leaves of F'' are of depths at most $a - w_{j+1}, \ldots, a - w_n$.

Then, we partition W' into $\beta(W';t)$ subsequences, each of leaf-oriented alphabetic minimax cost at most t, and construct a leaf-oriented alphabetic minimax tree for each subsequence. Let F' be the resulting forest. The $j - i + 1$ leaves of F' are all of depth at most $t - w_i = \cdots = t - w_j$.

Finally, for $i \leq k \leq i + \beta(W'; t) - 1$, we replace the kth leaf in F'' with the kth tree in F'. Let F be the resulting forest. There are $\beta(W''; a)$ trees in F and the leaves in F are of depths at most $a - w_1, \ldots, a - w_n$. Therefore, $\beta(W; a) \leq \beta(W''; a)$.

We now show that $\beta(W; a) \geq \beta(W''; a)$. First, we partition W into $\beta(W; a)$ subsequences, each of leaf-oriented alphabetic minimax cost at most a, and construct a leaf-oriented alphabetic minimax tree for each subsequence. Since the $(i-1)$st and $(j+1)$st leaves of T (if they exist) each have weight at least t, they are of depth at most $a - t$. We insert sufficiently many nodes above the ith through jth leaves of T so that they are of depths exactly $a - w_i, \ldots, a - w_j$. Let F be the resulting forest. Suppose that the ith through jth leaves of F have k ancestors of depth exactly $a - t$. Since the trees in F are binary and $w_i = \cdots = w_j < t$,

$$k \geq \left\lceil \frac{j - i + 1}{2^{t - w_i}} \right\rceil .$$

Therefore, by Lemma 6.1, $k \geq \beta(W'; t)$.

Next, we remove from F the ith through jth leaves of F and all of their ancestors that are of depth greater than $a - t$. Notice that, since the $(i-1)$st and $(j+1)$st leaves of F (if they exist) are of depth at most $a - t$, no leaves of F except the ith through jth have any ancestors that are removed. Let F' be the resulting forest. The first $i - 1$ leaves of F' are of depths at most $a - w_1, \ldots, a - w_{i-1}$, the next k leaves of F' are of depths exactly $a - t$, and the last $n - j$ leaves of F' are of depths at most $a - w_{j+1}, \ldots, a - w_n$.

Finally, we remove from F' all nodes that are not ancestors of the first $i - 1 + \beta(W'; t)$ leaves of F' and not ancestors of the last $n - j$ leaves of F'. Let F'' be the resulting forest. There are at most $\beta(W; a)$ trees in F'', the first $i - 1$ leaves in F'' are of depths at most $a - w_1, \ldots, a - w_{i-1}$, the next $\beta(W'; t)$ leaves of F'' are of depth exactly $a - t$, and the last $n - j$ leaves of F'' are of depths at most $a - w_{j+1}, \ldots, a - w_n$. Therefore, $\beta(W; a) \geq \beta(W''; a)$. Thus, $\beta(W; a) = \beta(W''; a)$. □

We now give a recursive algorithm for computing $\beta(W; a)$, whenever $\beta(W; a)$ is defined. First, using Lemma 6.2, we compute a sequence W'' of identical integers such that

$\beta(W; a) = \beta(W''; a)$. Then, using Lemma 6.1, we compute $\beta(W''; a)$.

If W is of length 1 and $a \geq w_1$, then $\beta(W; a) = 1$. Otherwise, let t be the maximum integer in W. We partition W into occurrences of the maximum t and maximal subsequences of non-maxima. For every non-empty maximal subsequence of non-maxima W', we recursively compute $\beta(W'; t)$ and replace W' by $\beta(W'; t)$ copies of the integer t. Let W'' be the resulting sequence of integers, all equal to t. By Lemma 6.2, if $a \geq \max_{1 \leq i \leq n}\{w_i\}$, then $\beta(W; a) = \beta(W''; a)$. By Lemma 6.1,

$$\beta(W''; a) = \left\lceil \frac{|W''|}{2^{a-t}} \right\rceil .$$

This algorithm's execution on a sample input $W = -1, -2, -2, -2, 0, -5, -2, -5, -2, 0$ and any integer $a \geq 0$ is represented by a tree in Figure 6.1. Each node in this tree corresponds to a recursive call. Each internal node shows, on its top line, the subsequence of W that is the argument to the corresponding recursive call. For example, W is initially partitioned into 4 parts: a maximal subsequence of non-maxima, $-1, -2, -2, -2$, which is the argument to a recursive call; a copy of the maximum in W, 0; a maximal subsequence of non-maxima, $-5, -2, -5, -2$, which is the argument to a recursive call; and another copy of the maximum in W, 0. Also, each internal node shows, on its bottom line, the sequence of identical integers with which that argument is replaced. For example, $\beta(-2, -2, -2; -1) = 2$, so the subsequence $-2, -2, -2$ is replaced with $-1, -1$. Each leaf corresponds to an instance of the base case, when the argument to the corresponding recursive call is a single integer. The algorithm replaces $W = -1, -2, -2, -2, 0, -5, -2, -5, -2, 0$ by $W'' = 0, 0, 0, 0, 0$. Therefore, for $a \geq 0$,

$$\beta(-1, -2, -2, -2, 0, -5, -2, -5, -2, 0; a) = \beta(0, 0, 0, 0, 0; a) = \lceil 5/2^a \rceil.$$

6.2.2 Definition of Sequence Maxima Trees

Our algorithm for computing $\beta(W; a)$ suggests a data structure, a *sequence maxima tree*. The sequence maxima tree T for $W = w_1, \ldots, w_n$ has n leaves. If $n = 1$, then T consists

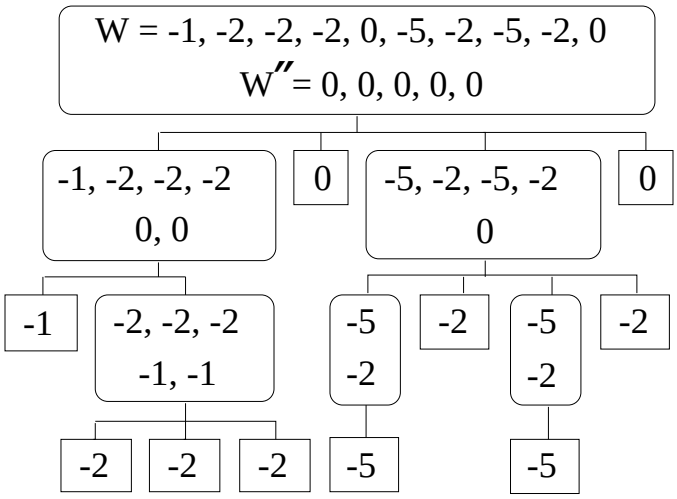


Figure 6.1: For $a \geq 0$, $\beta(-1,-2,-2,-2,0,-5,-2,-5,-2,0) = \beta(0,0,0,0,0;a)$.

of one internal node and one leaf. If $n > 1$, then the root of T has a child, which is a leaf, for every occurrence of the maximum in W. The root of T also has a child for every maximal subsequence W' of non-maxima, which is the root of the sequence maxima tree for W'. Notice that T is unique.

For $1 \leq i \leq n$, we say that the ith leaf of T has *weight* w_i. For every leaf v of T, either v is a child of the root or v's parent has a sibling that is also a leaf and has greater weight than v. Since leaf weights are integers, a leaf of depth l has weight at most the weight of the root's leaf children minus $l-1$. Therefore, T is of height at most

$$\max_{1 \leq i \leq n} \{w_i\} - \min_{1 \leq i \leq n} \{w_i\} + 1.$$

For each node v of T, we denote the subtree rooted at v by T_v. If v is a leaf of T, then the *label* of v is 1. If v is any internal node of T except the root and T_v's leaves are the ith through jth leaves of T, then the *label* of v is $\beta(w_i, \ldots, w_j; t)$, where

$$t = \begin{cases} w_{i-1}, & \text{if } j = n; \\ w_{j+1}, & \text{if } i = 1; \\ \min(w_{i-1}, w_{j+1}), & \text{otherwise.} \end{cases}$$

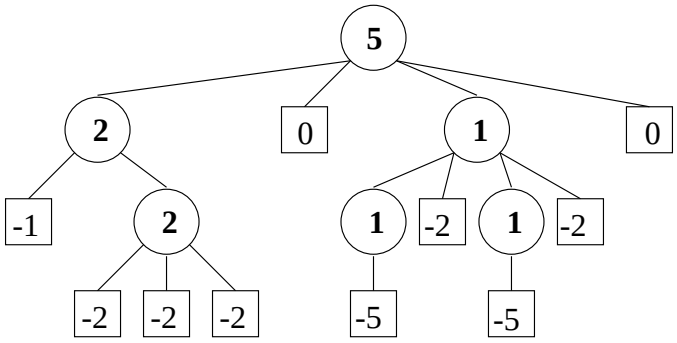


Figure 6.2: The sequence maxima tree for $-1, -2, -2, -2, 0, -5, -2, -5, -2, 0$.

If $\max_{1 \leq i \leq n}\{w_i\} \leq 0$, then the *label* of the root of T is $\beta(W; 0)$; otherwise, it is ∞. In addition to its label, every node of T stores the number of leaves in its subtree.

For example, the root of the sequence maxima tree for the sequence $-1, -2, -2, -2, 0, -5, -2, -5, -2, 0$ has four children: an internal node labeled 2 whose leaf descendants are of weights $-1, -2, -2, -2$; a leaf of weight 0; an internal node labeled 1 whose leaf descendants are of weight $-5, -2, -5, -2$; and a leaf of weight 0. This tree is shown in Figure 6.2. Internal nodes are shown with their labels and leaves are shown with their weights. Notice the close correspondence between this tree and the tree shown in Figure 6.1: the structure of the trees is the same; the information shown at corresponding leaves is the same; and, for any internal node v in Figure 6.1, the length of the sequence shown on the bottom line of v is shown in the corresponding internal node in Figure 6.2.

For each internal node v of T, including the root, the information stored at v can be computed from the information stored at v's children and the siblings immediately to v's left or right (if they exist). In particular, using Lemmas 6.1 and 6.2, we can compute v's label from the weight of its leaf children, the sum of its children's labels, and the weight of its leaf siblings (if they exist). The number of leaves in T_v is the sum of the number of leaves in the subtrees of v's children.

We consider the following operations on T:

- determine whether there exists a binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$;
- given a positive integer $i \leq n$, return the ith leaf of T;
- given positive integers $i \leq n$ and k, increase the ith element of W by k;
- given positive integers $i \leq n$ and k, decrease the ith element of W by k;
- given an integer w and a positive integer $i \leq n+1$, insert w into the ith position of W;
- given a positive integer $i \leq n$, delete the ith element of W;
- given a positive integer $i < n$, split T into sequence maxima trees for $w_1, \ldots, w_i$ and $w_{i+1}, \ldots, w_n$; and,
- given another sequence maxima tree T' for W', join T and T' into a sequence maxima tree for WW'.

We will show how to implement the sequence maxima tree T for $W = w_1, \ldots, w_n$ so that each of the operations supported by T takes $O(h \log n)$ time, where h is the height of T. This implies that we can implement a depth sequence D storing $L = l_1, \ldots, l_n$ by a sequence maxima tree for $-l_1, \ldots, -l_n$. This tree is of height at most

$$\max_{1 \leq i \leq n} \{l_i\} - \min_{1 \leq i \leq n} \{l_i\} + 1,$$

so each of the operations supported by D takes

$$O\left(\left(\max_{1 \leq i \leq n} \{l_i\} - \min_{1 \leq i \leq n} \{l_i\} + 1\right) \log n\right)$$

time.

First, we say how the internal nodes of T are implemented. Then, we say each of the operations supported by T can be implemented in $O(h \log n)$ time.

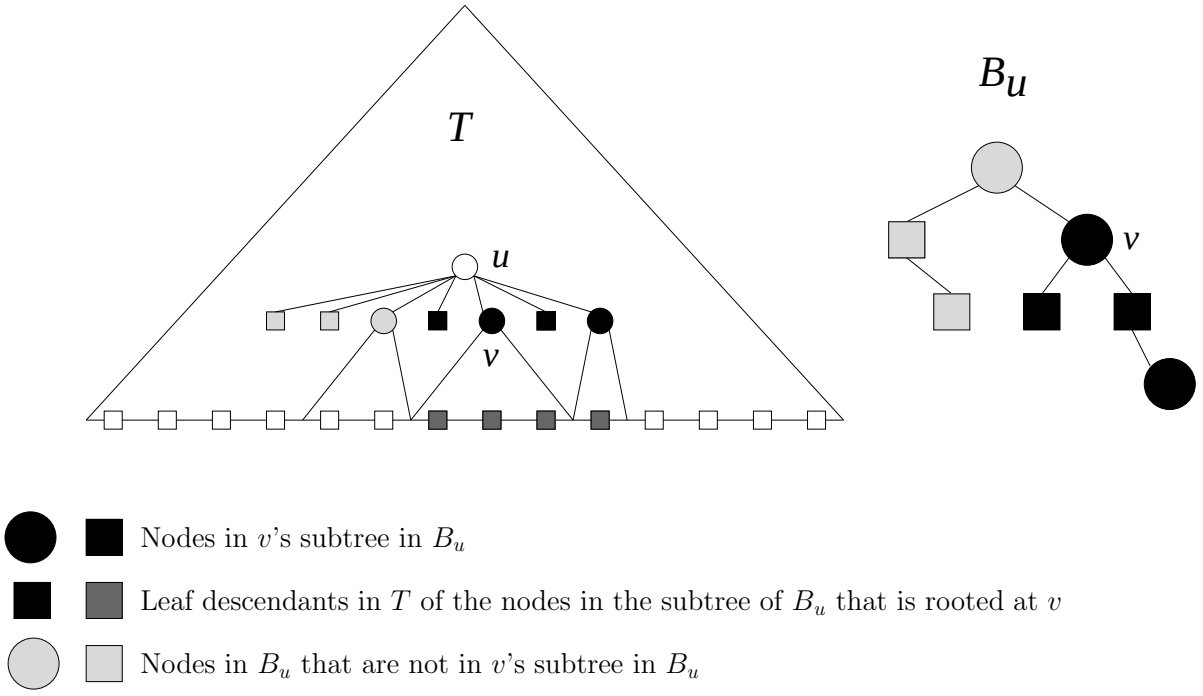


Figure 6.3: The list of u's children in the sequence maxima tree T is implemented by the balanced binary search tree B_u.

6.2.3 Implementation of Internal Nodes

Each internal node u of T can have unbounded degree. Because of this, we implement the list of u's children as a balanced binary search tree B_u, such as an AVL tree [Knu98b, Section 6.2.3]. This is illustrated in Figure 6.3. There u is an arbitrary internal node of T. Since T has n leaves, u has at most n children and so B_u is of height $O(\log n)$. We augment B_u so that each node of B_u stores additional information. We will specify this information as it is needed.

We want to store at u the sum of the labels of all of u's children in T, in such a way that it is easy to compute this sum after we update W. To do this, we augment B_u so that every node v of B_u stores the sum of the labels of all nodes in the subtree of B_u that is rooted at v. For the example shown in Figure 6.3, we store at v the sum of the

black nodes' labels.

Suppose that we want to update the information stored at each of the proper ancestors of a leaf v in T. We consider the ancestors of v in T, starting at v's parent and moving towards the root. Suppose that r is an ancestor of v in T and g is r's parent in T. Let B_g be the balanced binary search tree implementing the list of g's children. At each node q on the path from r to the root of B_g, we store the sum of the labels of all nodes in the subtree of B_g that is rooted at q. We also store the total number of leaf descendants in T of all nodes in the subtree of B_g that is rooted at q. Notice that this information can be computed from q's label, the number of leaf descendants of q in T, and information stored at q's children in B_g. Since B_g is of height $O(\log n)$, updating g takes $O(\log n)$ time. In total, updating T takes $O(h \log n)$ time.

6.2.4 Implementation of Queries

By definition, $\beta(W; 0) = 1$ if and only if $\alpha(W) \leq 0$. As noted in Section 5.1, there exists a binary tree on n leaves of depths at most $-w_1, \ldots, -w_n$ if and only if $\alpha(W) \leq 0$. Therefore, there exists such a tree if and only if $\beta(W; 0) = 1$. Since the root of T is labeled with $\beta(W; 0)$, we can determine whether there exists such a tree in $O(1)$ time.

We now show how, given a positive integer $i \leq n$, we can find the ith leaf x of T. For each internal node u of T, we augment the balanced binary search tree B_u that implements the list of u's children so that, for each node v of B_u, we store at v the total number of leaf descendants in T of all nodes in the subtree of B_u that is rooted at v. For the example shown in Figure 6.3, we store at v the total number of dark grey leaves and black leaves. Suppose we know that x is the jth leaf in T_u. We want to find the child v of u in T and the positive integer k such that x is the kth leaf in T_v. It is possible to do this in $O(\log n)$ time using B_u [CLRS01, Sections 14.1 and 14.2]. Therefore, we can find x starting from the root of T in $O(h \log n)$ time.

6.2.5 Implementation of Increase, Insert, and Split

Next, we show how to update T after increasing an element of W. Using this, we then describe how to insert an element into W and, given a positive integer $i < n$, how to split T into sequence maxima trees for $w_1, \ldots, w_i$ and $w_{i+1}, \ldots, w_n$.

Given positive integers $i \leq n$ and k, we can increase the ith element of W by k in $O(h \log n)$ time. To do this, we work bottom-up. First, we find the ith leaf v of T. Then, we increase v's weight in at most h stages, until it is equal to $w_i + k$. At each stage, except possibly the last, we also decrease v's depth.

Let u be the parent of v and let t be the weight of u's leaf siblings (if they exist). We consider the two possible cases: when u has leaf siblings and $t \leq w_i + k$, and when either u is the root or u has leaf siblings and $t > w_i + k$.

Suppose that u has leaf siblings and $t \leq w_i + k$. An example of how we deal with this case is shown in Figure 6.4. First, we increase v's weight to t. Second, we split the balanced binary search tree B_u implementing the list of children of v's parent u into two new balanced binary search trees B_x and B_y, that store v's left and right siblings, respectively. This takes $O(\log n)$ time. Third, we replace u by v in T. Notice that this decreases v's depth in T. If B_x is non-empty, then we insert a new internal node x immediately to the left of v as v's sibling in T. To do this, we insert x into the balanced binary search tree implementing the list of children of v's new parent in T. The children of x in T are the nodes in B_x. Similarly, if B_y is non-empty, then we insert a new internal node y immediately to the right of v as v's sibling in T. To do this, we insert y into the balanced binary search tree implementing the list of children of v's new parent in T. The children of y in T are the nodes in B_y. In T, after these operations, x or y may be an internal node whose only child is an internal node. When this is the case, we replace it with its only child. Finally, we compute the labels of any internal nodes that are new children of v's new parent in T. Notice that there are at most two such nodes, which can only occur immediately to the left and right of v. In Figure 6.4, these are x' and y.

Recall that the label of an internal node can be computed from the sum of its children's labels, which is stored at the root of the balanced binary search trees that implements the list of its children. This takes $O(1)$ time. In total, this stage takes $O(\log n)$ time. Notice that, after this stage, T is a sequence maxima tree for its leaf weights, except that the information stored at each of v's proper ancestors may need to be updated. Also, v's weight may still be less than $w_i + k$.

Now suppose that either u is the root or u has leaf siblings and $t > w_i+k$. An example of how we deal with this case is shown in Figure 6.5. The construction is similar to the previous case. First, we increase v's weight to w_i+k. Second, we split the balanced binary search tree B_u implementing the list of children of v's parent u into two new balanced binary search trees B_x and B_y, that store v's left and right siblings, respectively. This takes $O(\log n)$ time and destroys B_u. Third, we remove all of u's children in T except v. To do this, we replace B_u with a new balanced binary search tree B'_u storing only v. If B_x is non-empty, then we insert a new internal node x immediately to the left of v as v's sibling in T. To do this, we insert x into B'_u. The children of x in T are the nodes in B_x. Similarly, if B_y is non-empty, then we insert a new internal node y immediately to the right of v as v's sibling in T. To do this, we insert y into B'_u. The children of y in T are the nodes in B_y. In T, after these operations, x or y may be an internal node whose only child is an internal node. When this is the case, we replace it with its only child. Finally, we compute the labels of any internal nodes that are children of u in T. Notice that there are at most two such nodes, which can only occur immediately to the left and right of v. In Figure 6.5, these are x' and y. Recall that the label of an internal node can be computed from the sum of its children's labels, which is stored at the root of the balanced binary search trees that implement the list of its children. This takes $O(1)$ time. In total, this stage takes $O(\log n)$ time. Notice that, after this stage, T is a sequence maxima tree for its leaf weights, except that the information stored at each of v's proper ancestors may need to be updated.

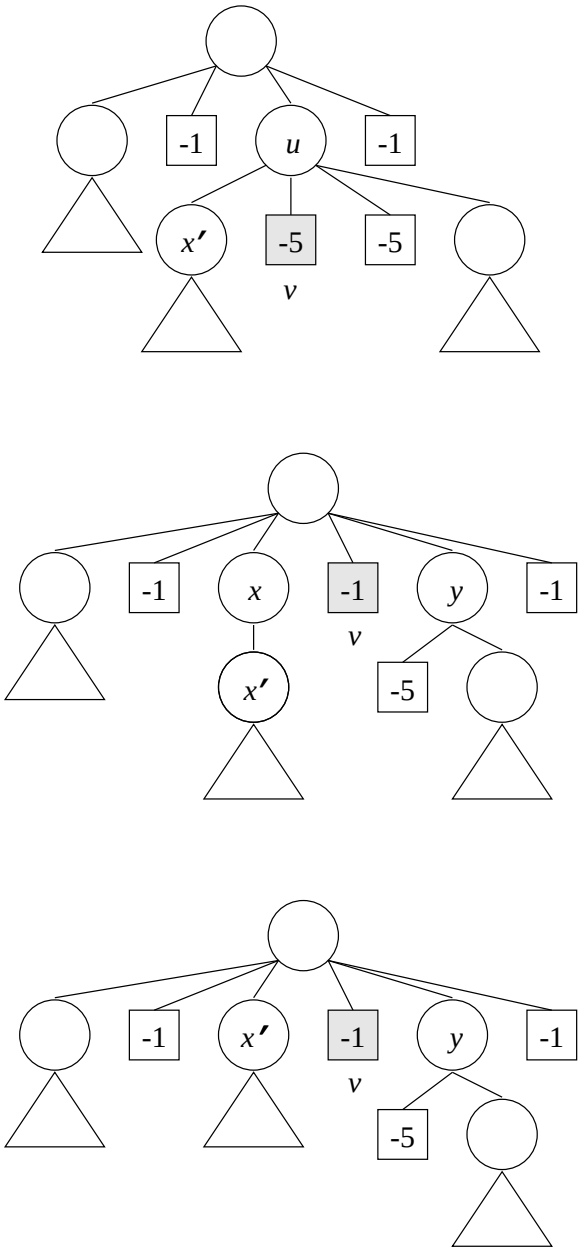


Figure 6.4: Increasing v's weight to equal u's leaf siblings' weight.

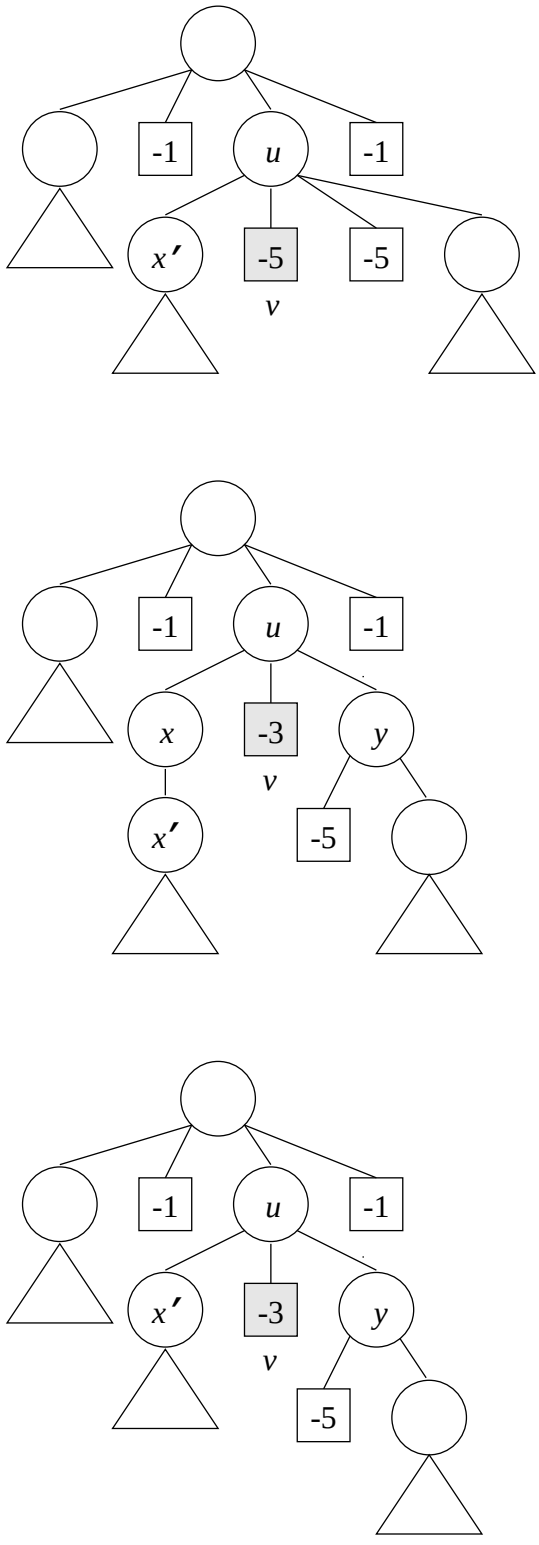


Figure 6.5: Increasing v's weight to less than u's leaf siblings' weight.

After the last stage, we update the information stored at each of v's proper ancestors.

We now show how, given an integer w and a positive integer $i \leq n+1$, we can insert w into the ith position of W in $O(h \log n)$ time. Notice that this lengthens W by 1. We consider two cases: when $i = 1$ or $i = n+1$, that is, so the new element is inserted at one of the ends of W, and when $1 < i < n$.

Suppose that $i = 1$; the case when $i = n$ is symmetric. First, we attach a new internal node u with label 1 as the left sibling of the first leaf of T. Second, we attach a leaf v with weight $\min(w, w_1 - 1)$ as u's only child. Third, we update the information stored at each of v's proper ancestors. Notice that the resulting tree is a sequence maxima tree for $W' = \min(w, w_1 - 1), w_1, \ldots, w_n$. Finally, if $w > w_1 - 1$, then we increase the first element of W' from $w_1 - 1$ to w.

Now suppose that $1 < i < n$. Also, suppose that $w_{i-1} \geq w_i$; the case when $w_{i-1} < w_i$ is symmetric. First, we attach a new internal node u with label 1 as the sibling immediately to the left of the ith leaf of T. Second, we attach a leaf v with weight $\min(w, w_i - 1)$ as u's only child. Third, we update the information stored at each of v's proper ancestors. Notice that, since $\min(w, w_i - 1) < w_i \leq w_{i-1}$, the resulting tree is a sequence maxima tree for $W' = w_1, \ldots, w_{i-1}, \min(w, w_i - 1), w_i, \ldots, w_n$. Finally, if $w > w_i - 1$, then we increase the ith element of W' from $w_i - 1$ to w.

We now show how, given a positive integer $i < n$, we can split T into sequence maxima trees for $w_1, \ldots, w_i$ and $w_{i+1}, \ldots, w_n$ in $O(h \log n)$ time. Let t be the maximum integer in W. First, we insert $t+1$ into the $(i+1)$st position of W. After this, T's root has three children: an internal node x whose subtree is the sequence maxima tree for $w_1, \ldots, w_i$, except that x is labeled with $\beta(w_1, \ldots, w_i; t+1)$; the $(i+1)$st leaf v of T, which has weight $t+1$; and an internal node y whose subtree is the sequence maxima tree for $w_{i+1}, \ldots, w_n$, except that y is labeled with $\beta(w_{i+1}, \ldots, w_n; t+1)$. Then, we remove v and the root of T, label x with $\beta(w_1, \ldots, w_i; 0)$, and label y with $\beta(w_{i+1}, \ldots, w_n; 0)$.

6.2.6 Implementation of Decrease, Delete, and Join

Next, we show how to update T after decreasing an element of W. Using this, we then describe how to delete an element of W and, given another sequence maxima tree T' for a sequence of integers W', how to join T and T' into the sequence maxima tree for WW'.

Given positive integers $i \leq n$ and k, we can decrease the ith element of W by k in $O(h \log n)$ time. To do this, we work top-down. First, we find the ith leaf v of T. Then, we decrease v's weight in at most h stages, until it is equal to $w_i - k$. At each stage, except possibly the first, we also increase v's depth.

Let x and y be the internal nodes immediately to v's left and right (if they exist). Let B_x and B_y be the balanced binary search trees implementing the lists of children of x and y. Let t be the maximum of $w_i - k$, the weight of x's leaf children, and the weight of y's leaf children.

First, we decrease v's weight to t. Second, if v has any leaf siblings, then we replace x (if it exists), v, and y (if it exists) by a new internal node u whose children are x, v, and y. Third, if x exists and the weight of x's leaf children is t, then we remove x from T and make the children of x children of v's parent u in T. To do this, we remove x from the balanced binary search tree B_u implementing the list of children of u. Then, we join B_x and B_u. This takes $O(\log n)$ time. Notice that each of the nodes in B_x is either a leaf in T with weight t or an internal node that is a sibling in T of such a leaf. Thus, the labels of these nodes is unchanged. Similarly, if y exists and the weight of y's leaf children is t, then we remove y from T and make the children of y children of u in T. To do this, we remove y from B_u. Then, we join B_u and B_y. This takes $O(\log n)$ time. Notice that each of the nodes in B_y is either a leaf in T with weight t or an internal node that is a sibling in T of such a leaf. Thus, the labels of these nodes is unchanged. Finally, we compute the labels of x and y (if they are in T). Recall that the label of an internal node can be computed from the sum of its children's labels, which is stored at the root of the balanced binary search tree that implements its list of children. This takes $O(1)$ time.

In total, this stage takes $O(\log n)$ time. Notice that, after this stage, T is a sequence maxima tree for its leaf weights, except that the information stored at each of v's proper ancestors may need to be updated. Also, v's weight may still be greater than $w_i - k$.

After the last stage, we update the information stored at each of v's proper ancestors.

We now show how, given a positive integer $i \leq n$, we can delete the ith element of W in $O(h \log n)$ time. First, if either w_{i-1} or w_{i+1} exists and is less than w_i, then we decrease the ith element of W from w_i to $t - 1$, where

$$t = \begin{cases} w_{i-1}, & \text{if } i = n; \\ w_{i+1}, & \text{if } i = 1; \\ \min(w_{i-1}, w_{i+1}), & \text{otherwise.} \end{cases}$$

Since the ith element of W is now less than both w_{i-1} and w_{i+1} (if they exist), v is an only child. Finally, we remove v and its parent from T, and update the information stored at each remaining node in T that was a proper ancestor of v.

We now show how, given another sequence maxima tree T' for W', we can join T and T' into the sequence maxima tree for WW' in $O(h \log n)$ time. Let t be the maximum integer in WW'. First, we label the root x of T with $\beta(W; t + 1)$ and label the root y of T' with $\beta(W'; t + 1)$. Then, we construct a sequence maxima tree for $W'' = W, t + 1, W'$ by attaching x, a new leaf v with weight $t + 1$, and y as children to a new node with label $\beta(W, t + 1, W'; t + 1)$. Finally, we delete the $(n + 1)$st element of W'', where n is the length of W.

Chapter 7

Conclusion

In this thesis, we gave a new algorithm for dynamic coding. This algorithm uses at most $\lceil \lg n \rceil + 1$ bits and $O(\log n)$ time to encode each character; and $(H(P) + 4.5)m + d \lg n$ bits and $O\left((H(P) + 1)m + d \log^2 n\right)$ time overall. Therefore in both the average case and the worst case, the number of bits and time used to encode a character is close to optimal.

We also gave new proofs of recent results by Evans and Kirkpatrick [EK00] for restructuring ordered trees on n leaves to be of height at most $(\lceil \lg n \rceil + 1)$ such that no leaf increases in depth by more than 2 or such that no node increases in depth by more than $\lceil \lg \lg n \rceil$. These proofs are somewhat simpler than those given by Evans and Kirkpatrick.

The alphabetic minimax cost of a sequence of integers is interesting because there exists a binary tree with leaves of depths $l_1, \ldots, l_n$ if and only if $\alpha(-l_1, \ldots, -l_n) \leq 0$. An alphabetic minimax tree for $-l_1, \ldots, -l_n$ is such a tree, if one exists. We gave a new algorithm for constructing a node-oriented alphabetic minimax tree in $O(n)$ time that allows new weights to be appended efficiently. We also gave an efficient implementation of an abstract data type that stores a dynamic list of non-negative integers and supports an operation to determine whether there exists a binary tree on nodes of those depths.

In this thesis, we focussed on $(\lceil \lg n \rceil + 1)$ as a height restriction for trees. In other re-

search [Gil71, Gar74, Ita76, Wes76, ML97, LMP99], the height restriction is a parameter. Generally, average-case performance of the resulting code tree or search tree improves as this parameter is increased. It is not clear how to generalize dynamic minimax coding so that the height restriction is a parameter.

Finally, it seems likely that most of our results can be generalized from binary to k-ary codes or trees. For example, we conjecture that it is always possible to restructure a ternary tree on n leaves to have height at most $\lceil \log_3 n \rceil + 1$ such that no leaf increases in depth by more than $\log_3 \log_3 n + O(1)$.

Bibliography

[Aig88] M. Aigner. *Combinatorial Search*. Wiley-Teubner, 1988.

[AM95] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. Technical Report TR-95-039, International Computer Science Institute, 1995.

[AW87] R. Ahlswede and I. Wegener. *Search Problems*. Wiley, 1987.

[AW96] S. Albers and J. Westbrook. Self-organizing data structures. In *Online Algorithms*, pages 13–51, 1996.

[BE98] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[BK93] A. Bookstein and S. T. Klein. Is Huffman coding dead? *Computing*, 50:279–296, 1993.

[BSA99] A. A. Belal, M. S. Selim, and S. M. Arafat. Towards a dynamic optimal alphabetic tree. Technical Report 99-22, Center for Discrete Mathematics and Theoretical Computer Science, 1999.

[BST85] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14:545–568, 1985.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[CKP86] D. Coppersmith, M. M. Klawe, and N. J. Pippenger. Alphabetic minimax trees of degree at most t. *SIAM Journal on Computing*, 15:189–192, 1986.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 2nd edition, 2001.

[DD96] R. De Prisco and A. De Santis. New lower bounds on the cost of binary search trees. *Theoretical Computer Science*, 156:315–325, 1996.

[EK00] W. S. Evans and D. G. Kirkpatrick. Restructuring ordered binary trees. In *Proceedings of the 11th Symposium on Discrete Algorithms*, pages 477–486, 2000.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[Fal73] N. Faller. An adaptive system for data compression. In *Proceedings of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.

[Fan49] R. Fano. The transmission of information. Technical Report 65, Research Laboratory of Electronics, M.I.T., 1949.

[FK93] A. S. Fraenkel and S. T. Klein. Bounding the depth of search trees. *The Computer Journal*, 36:668–678, 1993.

[Gal78] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24:668–674, 1978.

[Gar74] M. R. Garey. Optimal binary search trees with restricted maximal depth. *SIAM Journal on Computing*, 3:101–110, 1974.

[Gil71] E. N. Gilbert. Codes based on inaccurate source probabilities. *IEEE Transactions on Information Theory*, 17:304–314, 1971.

[GM59] E. N. Gilbert and E. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–968, 1959.

[Gol76] M. Golumbic. Combinatorial merging. *IEEE Transactions on Computers*, 24:1164–1167, 1976.

[GRVW95] D. Grinberg, S. Rajagopalan, R. Venkatesan, and V. K. Wei. Splay trees for data compression. In *Proceedings of the 6th Symposium on Discrete Algorithms*, pages 522–530, 1995.

[GW77] A. M. Garsia and M. L. Wachs. A new algorithm for minimum cost binary trees. *SIAM Journal on Computing*, 6:622–642, 1977.

[Hag98] T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science*, pages 366–398, 1998.

[HKT79] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal on Applied Mathematics*, 37:246–256, 1979.

[HT71] T. C. Hu and A. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21:514–532, 1971.

[HT72] T. C. Hu and K. C. Tan. Path length of binary search trees. *SIAM Journal on Applied Mathematics*, 22:225–234, 1972.

[Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IERE*, 40:1098–1101, 1952.

[Iac01] J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the 12th Symposium on Discrete Algorithms*, pages 516–522, 2001.

[Ita76] A. Itai. Optimal alphabetic trees. *SIAM Journal on Computing*, 5:9–18, 1976.

[Jon88] D. W. Jones. Application of splay trees to data compression. *Communications of the ACM*, 31:996–1007, 1988.

[KK85] D. G. Kirkpatrick and M. M. Klawe. Alphabetic minimax trees. *SIAM Journal on Computing*, 14:514–526, 1985.

[KN76] G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22:337–340, 1976.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:79–110, 1971.

[Knu85] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[Knu98a] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 2nd edition, 1998.

[Knu98b] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[Kra49] L. G. Kraft. A device for quantizing, grouping, and coding amplitude-modulated pulses. Master's thesis, Massachusetts Institute of Technology, 1949.

[Lar87] L. L. Larmore. Height restricted optimal alphabetic trees. *SIAM Journal on Computing*, 16:1115–1123, 1987.

[LH90] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37:464–473, 1990.

[LM01a] M. Liddell and A. Moffat. Length-restricted coding in static and dynamic frameworks. In *IEEE Data Compression Conference*, pages 133–142, 2001.

[LM01b] M. Liddell and A. Moffat. Length-restricted coding using modified probability distributions. In *Proceedings of the 24th Australasian Computer Science Conference*, pages 117–124, 2001.

[LM02] M. Liddell and A. Moffat. Incremental calculation of minimum-redundancy length-restricted codes. In *IEEE Data Compression Conference*, pages 182–191, 2002.

[LMP99] E. S. Laber, R. L. Milidiú, and A. A. Pessoa. Practical constructions of L-restricted alphabetic prefix codes. In *String Processing and Information Retrieval*, pages 115–119, 1999.

[LP94] L. L. Larmore and T. M. Przytycka. A fast algorithm for optimum height-limited alphabetic binary trees. *SIAM Journal on Computing*, 23:1283–1312, 1994.

[LP98] L. L. Larmore and T. M. Przytycka. The optimal alphabetic tree problem revisited. *Journal of Algorithms*, 28:1–20, 1998.

[Meh75] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.

[Meh77] K. Mehlhorn. A best possible bound for the weighted path length of binary search trees. *SIAM Journal on Computing*, 6:235–239, 1977.

[ML96] R. L. Milidiú and E. S. Laber. WARM-UP algorithm: A Lagrangean construction of length-restricted Huffman codes. Technical Report 15, Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, 1996.

[ML97] R. L. Milidiú and E. S. Laber. Improved bounds on the inefficiency of length restricted codes. Technical Report 33, Departamento de Informática, Pontificia Universidade Católica do Rio de Janeiro, 1997.

[MLP99] R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Bounding the compression loss of the FGK algorithm. *Journal of Algorithms*, 32:195–211, 1999.

[MPL98] R. L. Milidiú, A. A. Pessoa, and E. S. Laber. In-place length-restricted prefix coding. In *String Processing and Information Retrieval*, pages 50–59, 1998.

[MPL99] R. L. Milidiú, A. A. Pessoa, and E. S. Laber. Efficient implementation of the WARM-UP algorithm for the construction of length-restricted prefix codes. In *Algorithm Engineering and Experimentation*, pages 1–17, 1999.

[MT02] A. Moffat and A. Turpin. *Compression and coding algorithms*. Kluwer, 2002.

[Nak91] N. Nakatsu. Bounds on the redundancy of binary alphabetical codes. *IEEE Transactions on Information Theory*, 37:1225–1229, 1991.

[Rob55] H. Robbins. A remark on Stirling's formula. *American Mathematical Monthly*, 62:26–29, 1955.

[Sch64] E. Schwartz. An optimum encoding with minimum longest code and total number of digits. *Information and Control*, 7:37–44, 1964.

[Sch95] B. Schieber. Computing a minimum-weight k-link path in graphs with the concave Monge property. In *Proceedings of the 6th Symposium on Discrete Algorithms*, pages 405–411, 1995.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[She89] M. W. Sherk. *Self-adjusting k-ary search trees and self-adjusting balanced search trees*. PhD thesis, University of Toronto, 1989.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7:166–169, 1964.

[ST85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[TM95] A. Turpin and A. Moffat. Practical length-limited coding for large alphabets. *The Computer Journal*, 38:339–347, 1995.

[Tur98] A. Turpin. *Efficient prefix coding*. PhD thesis, University of Melbourne, 1998.

[Van74] D. Van Voorhis. Constructing codes with bounded codeword lengths. *IEEE Transactions on Information Theory*, 20:288–290, 1974.

[Vit87] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34:825–845, 1987.

[vL76] J. van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages and Programming*, pages 382–410, 1976.

[WAF01] J. Watkinson, M. Adler, and F. Fich. New protocols for asymmetric communication channels. In *Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity*, pages 337–350, 2001.

[Wat00] J. Watkinson. New protocols for asymmetric communication channels. Master's thesis, University of Toronto, 2000.

[Wes76] R. L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Information Processing Letters*, 4:90–94, 1976.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30:520–540, 1987.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–349, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences by variable rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.