ELSEVIER

Theoretical Computer Science 188 (1997) 1-44

Theoretical Computer Science

Tutorial

Optimal binary search trees

S.V. Nagaraj *

The Institute of Mathematical Sciences, CIT Campus Madras 600 113, India

Received October 1996
Communicated by M. Nivat

Abstract

We consider the problem of building *optimal binary search trees*. The binary search tree is a widely used data structure for information storage and retrieval. A binary search tree T for a set of keys from a total order is a binary tree in which each node has a key value and all the keys of the left subtree are less than the key at the root and all the keys of the right subtree are greater than the key at the root, this property holding recursively for the left and right subtrees of the tree T.

Suppose we are given n keys and the probabilities of accessing each key and those occurring in the gap between two successive keys. The *optimal binary search tree problem* is to construct a binary search tree on these n keys that minimizes the expected access time. One variant of this problem is when only the gaps have nonzero access probabilities, and is called the *optimal alphabetic tree problem*. Another related problem is when there is no order between the keys and there are probabilities associated only with the gaps and the objective is to build a binary tree with minimum expected weighted path length from the root. This is called the *Huffman tree problem*.

In this survey, we assess known results on the structural properties of the optimal trees, algorithms and lower bounds to construct and to verify optimal trees and heuristics to construct nearly optimal trees and other related results.

Keywords: Binary search tree; Data structures; Optimal binary search tree

1. Introduction

1.1. Motivation

The *binary search tree* is a widely used data-structure for information storage and retrieval as it supports many dynamic-set operations including Search, Minimum, Maximum, Predecessor, Successor, Insert and Delete.

* E-mail: svn@imsc.ernet.in.

PII S0304-3975(96)00320-9

A *binary tree* is either empty or composed of a root node together with left and right subtrees which are themselves binary trees. A *binary search tree* T for a set of keys from a total order is a binary tree in which each node has a key value and all the keys of the left subtree are less than the key at the root and all the keys of the right subtree are greater than the key at the root, this property holding recursively for the left and right subtrees of the tree T.

Suppose we are given n keys $K_1 < K_2 < \cdots < K_n$ and $2n+1$ probabilities $\beta_1, \beta_2, \ldots, \beta_n, \alpha_0, \alpha_1, \ldots, \alpha_n$ with $\sum \beta_i + \sum \alpha_j = 1$ where β_i is the probability of accessing the key K_i and α_j is the probability of accessing a key which lies between K_j and K_{j+1}, with α_0 and α_n having obvious interpretations. Consider a binary search tree for this set of keys. Let b_i be the number of edges on the path from the root to the interior node K_i and a_j be the number of edges on the path from the root to the leaf (K_j, K_{j+1}). Then the expected cost of accessing all the keys in the binary search tree is $\Sigma\beta_i(b_i+1)+\Sigma\alpha_j a_j$, since the cost of accessing key K_i is b_i+1, while for the gap (K_j, K_{j+1}) it is simply a_j. An *optimal binary search tree* for this set of keys is one which has the minimum cost. The *optimal binary search tree problem* is to construct an optimal binary search tree given the keys and their access probabilities.

A simple dynamic programming algorithm requiring $O(n^3)$ time and $O(n^2)$ space was given by Gilbert and Moore [48] for the special case when only the β_i's are zero. This algorithm was extended by Knuth [86] to include the case when the β_i's are also present. Knuth also improved the algorithm to run in $O(n^2)$ time by observing a *monotonicity property* of the roots of the optimal binary search trees. Another solution for the same problem was given by Yao [161] using her general method for speeding up dynamic programming. This however had the same time and space complexities. Knuth's algorithm is the best known so far for the general optimal binary search tree problem. Recently, Karpinski et al. [74] gave an algorithm with subquadratic expected running time for the special case of the optimal binary search tree problem when the α's are zero. They also mention that they can obtain such a result for the general case.

We study known upper and lower bounds [86, 113, 129, 130, 162] on the cost of optimal and *nearly optimal binary search trees*. For an optimal binary search tree T with a subtree $S(d)$ rooted at a depth d from the root of T, we study the ratio of the weight of $S(d)$, to the weight of T and correct a result obtained by Hirschberg et al. [60]. This result is used by Larmore [95] in his subquadratic algorithm for constructing approximately optimal binary search trees.

Several heuristics for constructing binary search trees which have nearly the optimal cost but requiring significantly less time and often less space than Knuth's algorithm have been proposed [95, 102, 110, 111, 129, 162]. We explore some of these heuristics.

The optimal binary search tree problem when the β_i's are all zero is called the *optimal alphabetic tree problem*. An $O(n^2)$ time, $O(n^2)$ space algorithm for constructing optimal alphabetic trees was first proposed by Hu and Tucker [68]. This was later improved by Knuth [86]. The improved algorithm required only $O(n \log n)$ time and $O(n)$ space and employed better data structures. The original proof of correctness [68] of the Hu–Tucker algorithm as given by its inventors was extremely complicated and lengthy.

A much simpler proof of correctness was given later by Hu [63] and Hu et al. [65]. Another $O(n \log n)$ time, $O(n)$ space algorithm was found by Garsia and Wachs [43]. The two algorithms were shown to be equivalent later [64]. The proof of correctness of the Garsia–Wachs algorithm was simplified considerably much later by Kingston [78]. We discuss this algorithm, its implementation and its proof of correctness.

It has been shown recently by Klawe and Mumey [80] that a class of techniques for finding optimal alphabetic trees which includes all current methods yielding $O(n \log n)$ algorithms are at least as hard as sorting in whatever model of computation is employed. They introduce an idea for finding optimal alphabetic trees which they refer to as *region processing* and use this method to produce $O(n)$ algorithms for the case when all inputs are within a constant factor of one another and when they are exponentially separated, notions which they define. Linear-time algorithms for constructing optimal alphabetic trees when the weights are within a factor of two or when the input is in sorted order are known [64, 116]. We discuss these algorithms.

Another related problem is the well-known *Huffman tree problem* where the external node weights may appear in any order, not necessarily the alphabetic order. The Huffman trees and the alphabetic trees are used for data compression [85, 87]. An optimal $O(n \log n)$ algorithm (under the comparison tree model) for constructing Huffman trees was first proposed by Huffman [69]. We study the linear-time algorithms known for the Huffman tree problem when the weights are within a factor of 2 or when the input is in sorted order [64, 80, 113, 116]. We discuss the problem of verifying the optimality of weighted extended binary trees and Huffman trees.

The main source of motivation in studying the optimal binary search tree problem are the many unresolved problems [65, 80, 116, 137] associated with it. The most important ones are the following:

1. What is the true complexity of the optimal binary search tree problem ?

In other words, is there an $o(n^2)$ time/space algorithm for the general case of the optimal binary search tree problem ?

2. What is the true complexity of the optimal alphabetic tree problem ?

In other words, is there an $o(n \log n)$ time algorithm for the optimal alphabetic tree problem ?

3. Is there an $o(n \log n)$ algorithm for the Huffman tree problem in the general model where we are allowed to compute floors and ceilings of numbers ?

4. How fast can we test whether a given binary search tree or alphabetic tree is optimal ?

In an attempt to address these questions, we first study and survey important known results on optimal binary search trees and related variants.

1.2. Notation

A *binary tree* is a finite set of vertices that is either empty or consists of a root node together with two binary subtrees which are disjoint from each other, and from the root, and are called the left and right subtree.

An *extended binary tree* is a binary tree with external nodes attached to its leaf nodes.

A *binary search tree* T for a set of keys from a total order is a binary tree in which each node has a key value and all the keys of the left subtree are less than the key at the root and all the keys of the right subtree are greater than the key at the root. This property holding recursively for the left and right subtrees of the tree T.

The *level* of any node in a binary tree is the number of internal nodes on the path from the root to that node.

We measure the *cost of searching for a key* K in a binary search tree T by the number of comparisons, $C_T(K)$ needed to locate K or to determine that K is not in T. Assume that T contains keys $K_1, K_2, \ldots, K_n$. The interval between the keys K_i and K_{i+1} is denoted by (K_i, K_{i+1}). For $1 \leqslant i \leqslant n$ we define $C_T(K_i) = 1 +$ number of edges in the path from the root of T to K_i, and for $0 \leqslant j \leqslant n$ and $K_j < K < K_{j+1}$, define $C_T(K) = C_T((K_j, K_{j+1}))$ to be the number of edges in the path from the root of T to (K_j, K_{j+1}).

It is possible to associate real numbers, or *weights* with the keys and the intervals between the keys, of a binary search tree T. We denote the *weight* of the key K_i of T by β_i for $1 \leqslant i \leqslant n$, and the weight of (K_j, K_{j+1}) by α_j for $0 \leqslant j \leqslant n$. We define the *cost of a binary search tree* T to be

$$C(T) = \sum_{i=1}^{n} C_T(K_i)\beta_i + \sum_{j=0}^{n} C_T((K_j, K_{j+1}))\alpha_j.$$

The cost of a binary search tree is also called as its *weighted path length.*

The *weight of a node* r is denoted by $w(r)$. The *weight of a binary search tree* T is defined to be the sum of the weights of its internal and external nodes. If the weights β_i and α_j of a binary search tree T satisfy the condition $\sum_{i=1}^{n} \beta_i + \sum_{j=0}^{n} \alpha_j = 1$, these weights are called *probabilities.*

Let $(\gamma_1, \ldots, \gamma_n)$ be a discrete probability distribution i.e. $\gamma_i \geqslant 0$ and $\Sigma\gamma_i = 1$. Then $H(\gamma_1, \ldots, \gamma_n) = -\sum_{i=1}^{n} \gamma_i . \lg(\gamma_i)$ is called the *entropy* of the distribution. It is denoted by H.

For a binary search tree T, we use the following notation when we discuss subtree weight ratios:

$root(T)$	the root of tree T
$W(T)$	the weight of tree T
T_L, T_R	the left and right subtrees of T
T_{LL}, T_{LR}	the left and right subtrees of $root(T_L)$
T_{RL}, T_{RR}	the left and right subtrees of $root(T_R)$
β_0	the weight of $root(T)$
β_L, β_R	the weights of $root(T_L), root(T_R)$

An *optimal binary search tree*, given a set of keys and the probabilities of accessing them and the gaps is a binary search tree for that set which has the minimum cost. The *optimal binary search tree problem* is to construct an optimal binary search tree.

Given a sequence of n weights $\beta_1, \beta_2, \ldots, \beta_n$ an *alphabetic tree* for this sequence of weights is any binary tree whose leaves have these weights, such that as we traverse the tree in inorder, the weights occur in the given (alphabetic) order. We can alternatively think of these weights as gap probabilities given $n-1$ keys. An alphabetic tree having the minimum cost for a given sequence of weights is called an *optimal alphabetic tree* for that set of weights. The *optimal alphabetic tree problem* is to construct an optimal alphabetic tree for a given sequence of weights. If we relax the condition that the weights should be in alphabetic order then the optimal alphabetic tree problem is called the *Huffman tree problem.*

In the case of alphabetic trees and Huffman trees, the *weight of an internal node* is defined bottom up recursively, as the sum of the weights of its children.

In any binary tree T, the *abstract position* of a node x is defined to be the bit list of descent commands necessary to find x from the root. By $position_T(x)$ we denote the abstract position of x in T.

We define $u \Rightarrow v$ to mean v is the inorder successor of u in some abstract binary tree.

1.3. Overview

In Section 2, we study the general optimal binary search tree problem and solutions to it, including the classical dynamic programming algorithm. We look at Knuth's improved method using the monotonicity property of the roots of the optimal binary search trees. We analyse bounds on the cost of optimal trees and look at subtree weight ratios for optimal binary search trees. Finally, we consider heuristics for constructing nearly optimal binary search trees.

In Section 3, we study the Huffman tree problem. We give the classical $O(n \log n)$ algorithm to construct Huffman trees. We note certain properties of Huffman trees and look at $O(n)$ algorithms to construct Huffman trees in some special cases. We discuss the problem of verifying the optimality of weighted extended binary trees and Huffman trees.

In Section 4, we study the optimal alphabetic tree problem. We look at the Hu–Tucker and Garsia-Wachs algorithms for constructing optimal alphabetic trees. We also note the equivalence of the two algorithms. We discuss algorithms for constructing optimal alphabetic trees in some special cases. We also study the lower bound result for constructing optimal alphabetic trees obtained by Klawe and Mumey.

In Section 5, we discuss open problems and directions for further research.

2. Optimal binary search trees

In this section, we study the general optimal binary search tree problem and solutions to it, including the classical dynamic programming algorithm. We look at Knuth's improved method using the monotonicity property of the roots of the optimal binary

search trees. We analyse bounds on the cost of optimal trees and look at subtree weight ratios for optimal binary search trees. Finally, we consider heuristics for constructing nearly optimal binary search trees.

2.1. The dynamic programming algorithm

2.1.1. The $O(n^3)$ dynamic programming algorithm

The number of different binary search trees on n nodes is the Catalan number viz. $[1/(n+1)]\binom{2n}{n} \approx 4^n/n\sqrt{\pi n}$. This gives an $\Omega(n)$ lower bound for the optimal binary search tree problem in the decision tree model. However, an exhaustive search for the optimum will result in an algorithm which is exponential in n. We can do much better as seen below.

The first algorithm running in time polynomial in n was given for the special case of optimal alphabetic trees by Gilbert and Moore [48] and required $O(n^3)$ time and $O(n^2)$ space. This method was extended by Knuth [85] to include the more general case where the successful and unsuccessful search probabilities are both taken into account. The key fact that makes the optimal binary search tree problem amenable to dynamic programming is that all the subtrees of an optimal tree are optimal (this is the principle of optimality). If K_i appears at the root then its left subtree is an optimum solution for the probabilities $\alpha_0, \alpha_1, \ldots, \alpha_{i-1}$ and $\beta_1, \ldots, \beta_{i-1}$, its right subtree is an optimum solution for the probabilities $\alpha_i, \ldots, \alpha_n$ and $\beta_{i+1}, \ldots, \beta_n$. Therefore, we can get a bottom up algorithm for building an optimal binary search tree for a set of probabilities $(\alpha_0, \ldots, \alpha_n, \beta_1, \ldots, \beta_n)$. We can build up optimal trees $T_{i,j}$ for all the probabilities $\alpha_i, \ldots, \alpha_j$ and $\beta_{i+1}, \ldots, \beta_j$ where $i \leqslant j$ starting from the smallest intervals and working toward the largest.

Let $P_{i,j}$ and $W_{i,j}$ denote the weighted path length and the total weight of an optimal binary search tree for all words $K_i < X \leqslant K_{j+1}$ where $i \leqslant j$. Let $R_{i,j}$ denote the index of the root of this tree when $i < j$. The following formulae determine the cubic time algorithm:

$$P_{i,i} = W_{i,i} = \alpha_i \quad \text{for } 0 \leqslant i \leqslant n, \tag{1}$$

$$W_{i,j} = W_{i,j-1} + \beta_j + \alpha_j, \tag{2}$$

$$P_{i,j} = W_{i,j} + \min_{i<k\leqslant j} (P_{i,k-1} + P_{k,j}) \quad \text{for } 0 \leqslant i < j \leqslant n. \tag{3}$$

Since we choose $R_{i,j}$ from among $j-i$ pairs for each i, j such that $0 \leqslant i < j \leqslant n$ the algorithm runs in $O(n^3)$ time, as there are only $(n+1)(n+2)/2$ choices of $0 \leqslant i \leqslant j \leqslant n$, the space required being $O(n^2)$.

2.1.2. The monotonicity of roots and consequent $O(n^2)$ algorithm

Knuth [85] observed that the $R_{i,j}$'s satisfy the condition $R_{i,j-1} \leqslant R_{i,j} \leqslant R_{i+1,j}$. We will look at the proof in the next subsection. This condition means that we only have to search all the indices between $R_{i,j-1}$ and $R_{i+1,j}$ to compute $R_{i,j}$. The running time

of the algorithm is therefore,

$$O\left(\sum_{i=0}^{n}\sum_{j=0}^{n}(R_{i+1,j}-R_{i,j-1}+1)\right)$$

which is $O(\sum_{i=0}^{n-1} n) = O(n^2)$. This is the best algorithm known so far for the general optimal binary search tree problem. Karpinski et al. [74] recently gave an algorithm with subquadratic expected running time for the special case when the α's are zero. They also claim that they can obtain such a result for the general problem.

2.2. *Properties of optimal binary search trees*

In this subsection we look at certain properties of optimal binary search trees that are helpful to obtain fast algorithms and to obtain nearly optimal binary search trees.

2.2.1. *Monotonicity of the roots*

As observed in the previous subsection, Knuth proved the monotonicity of the roots in an optimal binary search tree. Yao [161] observed that the recurrence obtained for the optimal binary search tree problem can be generalised to solve a larger class of dynamic programming problems, using which she proved the monotonicity of the roots. We look at her proof.

Let $w(i,j)$ for $1 \leqslant i < j \leqslant n$ be real numbers and let $c(i,j)$ be defined by

$$c(i,i) = 0, \tag{4}$$

$$c(i,j) = w(i,j) + \min_{i<k\leqslant j}(c(i,k-1)+c(k,j)) \quad \text{for } i < j. \tag{5}$$

The recurrence of $c(i,j)$ for optimal binary search trees is a special case of this recurrence where we have $w(i,j) = W_{i,j} = \alpha_i + \beta_{i+1} + \cdots + \beta_j + \alpha_j$, $c(i,j) = P_{i,j}$

If w satisfies

$$w(i,j) + w(i',j') \leqslant w(i',j) + w(i,j') \quad \text{for } i \leqslant i' < j \leqslant j' \tag{6}$$

it is said to satisfy *Quadrangle inequality*.

We use $c_k(i,j)$ to denote $w(i,j) + c(i,k-1) + c(k,j)$ and define

$$R(i,i) = i, \tag{7}$$

$$R(i,j) = \max\{k; c_k(i,j) = c(i,j)\} \quad \text{for } i < j. \tag{8}$$

Then $R(i,j)$ is the largest index where the minimum is achieved in the definition of $c(i,j)$.

w is said to be *monotone* if $w(i,j') \leqslant w(i',j)$ for $i \leqslant i' < j \leqslant j'$.

The following theorem (see [113]) proves the *monotonicity of the roots*.

Theorem 2.1. *If w satisfies the quadrangle inequality and is monotone, then $R(i,j) \leqslant R(i,j+1) \leqslant R(i+1,j+1)$ for $i \leqslant j$.*

Note: It is easily seen that the $w(i,j)$ for the optimal binary search tree problem is monotone and satisfies the quadrangle inequality in fact with equality.

We state a few lemmas (see [113]) leading to the proof of the theorem.

Lemma 2.2. *If w satisfies the quadrangle inequality and is monotone, then the function c defined above also satisfies the quadrangle inequality, i.e. $c(i,j)+c(i',j')\leqslant c(i,j')+c(i',j)$ for $i\leqslant i'\leqslant j\leqslant j'$.*

Proof. We use induction on the length $l=j'-i$ to prove the result.

This inequality is trivially true if $i=i'$ or $j=j'$. This proves the quadrangle inequality for c for $l\leqslant 1$. For the induction step we distinguish two cases $i'=j$, $i'<j$.

Case 1: $i<i'=j<j'$. In this case the quadrangle inequality for c reduces to

$$c(i,j)+c(j,j')\leqslant c(i,j'). \tag{9}$$

Let $k=R(i,j')$. We distinguish two symmetric subcases: $k\leqslant j$, $k\geqslant j$.

Case 1.1: $k\leqslant j$. We have

$$\begin{aligned} c(i,j)+c(j,j')\leqslant\ & w(i,j)+c(i,k-1)+c(k,j)+c(j,j') \\ & \text{(by definition of } c(i,j)) \qquad (10) \\ \leqslant\ & w(i,j')+c(i,k-1)+c(k,j)+c(j,j') \\ & \text{(by monotonicity of } w) \qquad (11) \\ \leqslant\ & w(i,j')+c(i,k-1)+c(k,j') \\ & \text{(by the induction hypothesis)} \qquad (12) \\ =\ & c(i,j') \quad \text{(by definition of } c(i,j') \text{ and } k). \qquad (13) \end{aligned}$$

Case 1.2: $k\geqslant j$. As this case is symmetric to case 1.1 the proof is similar.

Case 2: $i<i'<j<j'$. Let $y=R(i',j)$ and $z=R(i,j')$. We have to distinguish two symmetric cases: $z\leqslant y$ or $z\geqslant y$. We only consider the case $z\leqslant y$. We note that $z\leqslant y\leqslant j$ by the definition of y and $i<z$ by the definition of z. We have

$$\begin{aligned} & c(i',j')+c(i,j) \qquad (14) \\ & \leqslant c_y(i',j')+c_z(i,j) \qquad (15) \\ & = w(i',j')+c(i',y-1)+c(y,j')+w(i,j)+c(i,z-1)+c(z,j) \qquad (16) \\ & \leqslant w(i,j')+w(i',j)+c(i',y-1)+c(i,z-1)+c(z,j)+c(y,j') \\ & \quad \text{(by the quadrangle inequality for } w) \qquad (17) \end{aligned}$$

$$\leqslant w(i,j') + w(i',j) + c(i',y-1) + c(i,z-1) + c(y,j) + c(z,j')$$

(by the induction hypothesis)

(i.e. the quadrangle inequality for c applied to $z \leqslant y \leqslant j < j'$) (18)

$$= c(i,j') + c(i',j) \quad \text{(by the definition of } y \text{ and } z\text{).} \tag{19}$$

This completes the induction step and proves the lemma. □

Proof of the main Theorem. The claim is trivially true when $i=j$ so we assume $i<j$. We will show $R(i,j) \leqslant R(i,j+1)$, the argument for $R(i,j+1) \leqslant R(i+1,j+1)$ follows by symmetry. Since $R(i,j)$ is the largest index where the minimum is assumed in the definition of $c(i,j)$, it is sufficient if we show

$$[c_{k'}(i,j) \leqslant c_k(i,j)] \Rightarrow [c_{k'}(i,j+1) \leqslant c_k(i,j+1)] \qquad \text{for all } i < k \leqslant k' \leqslant j \tag{20}$$

We show a stronger inequality, for all $i < k \leqslant k' \leqslant j$

$$c_k(i,j) - c_{k'}(i,j) \leqslant c_k(i,j+1) - c_{k'}(i,j+1), \tag{21}$$

i.e.

$$c_k(i,j) + c_{k'}(i,j+1) \leqslant c_{k'}(i,j) + c_k(i,j+1), \tag{22}$$

or equivalently by expanding all the four terms using their definition

$$c(k,j) + c(k',j+1) \leqslant c(k',j) + c(k,j+1). \tag{23}$$

This is just the quadrangle inequality for c at $k \leqslant k' \leqslant j \leqslant j+1$.

As discussed earlier this result yields an $O(n^2)$ time algorithm for computing $c(1,n)$ and hence for the optimal binary search tree problem.

2.2.2. *Bounds on the cost of optimal binary search trees*

It is useful to obtain upper and lower bounds on the cost of optimal binary search trees and their variants in terms of the access probabilities. Many useful bounds have been obtained by methods of information theory [86, 110, 111, 113, 130, 162].

Mehlhorn [113] obtained a lower bound for the cost of any binary search tree T for the given set of keys, in particular for the optimum tree. It requires the following result.

Lemma 2.3. *Let c be a real number such that $0 \leqslant c < 1$. Let $\bar{\beta}_i = ((1-c)/2)^{b_i} c$ for $1 \leqslant i \leqslant n$, $\bar{\alpha}_j = ((1-c)/2)^{a_j}$ for $0 \leqslant j \leqslant n$ then $\bar{\beta}_i, \bar{\alpha}_j \geqslant 0$ and $\Sigma\bar{\beta}_i + \Sigma\bar{\alpha}_j = 1$ i.e. $(\bar{\alpha}_0, \bar{\beta}_1, \ldots, \bar{\beta}_n, \bar{\alpha}_n)$ is a probability distribution.*

The proof of this lemma is by induction on n. By showing that it holds for the left and right subtrees of the tree T, we can show that it holds for the tree T.

We now prove Mehlhorn's theorem (see [113]) which enables us to get a lower bound on the weighted path length of any binary search tree.

Theorem 2.4 (Lower bound on the weighted path length). *Let* $B = \Sigma\beta_i$ *then*

1. $\max\left\{\frac{H-dB}{\lg(2+2^{-d})}; d \in \Re\right\} \leqslant P$,
2. $H \leqslant P + B[\lg(e) - 1 + \lg(P/B)]$.

Proof. 1. Define $\bar{\beta}_i$ and $\bar{\alpha}_j$ as in the previous lemma. Then

$$b_i + 1 = 1 + (\lg(\bar{\beta}_i) - \lg(c))/\lg(\bar{c}), \tag{24}$$

$$a_j = \lg(\bar{\alpha}_j)/\lg(\bar{c}), \tag{25}$$

where $\bar{c} = (1 - c)/2$.

Now

$$P = \sum\beta_i(b_i + 1) + \sum\alpha_j a_j \tag{26}$$

$$= B\left(1 - \frac{\lg(c)}{\lg(\bar{c})}\right) + \frac{1}{\lg(\bar{c})}(\sum\beta_i \lg(\bar{\beta}_i) + \sum\alpha_j \lg(\bar{\alpha}_j)) \tag{27}$$

$$\geqslant B\left(1 - \frac{\lg(c)}{\lg(\bar{c})}\right) - \left(\frac{1}{\lg(\bar{c})}\right) H \tag{28}$$

$$= \frac{H - B\lg(\bar{c}/c)}{\lg(1/\bar{c})}, \tag{29}$$

where we have used a property of the entropy function, viz.,

$$H(\gamma_1, \ldots, \gamma_n) = H(\gamma_1 + \gamma_2, \gamma_3 \ldots, \gamma_n) + (\gamma_1 + \gamma_2)H\left(\frac{\gamma_1}{\gamma_1 + \gamma_2}, \frac{\gamma_2}{\gamma_1 + \gamma_2}\right). \tag{30}$$

Setting $d = \lg(\bar{c}/c)$ and observing that $c/\bar{c} = 2c/(1 - c)$ is a surjective mapping from $0 \leqslant c < 1$ onto the reals gives us the required result.

2. As there is no closed-form expression for the value of d which minimises the left-hand side of the first inequality we proved, numerical methods have to be used to compute $d_{\max}$ in every single application. A good approximation for $d_{\max}$ is $d = \lg(P/2B)$. It yields

$$H \leqslant P\lg(2 + 2^{-d}) + dB \tag{31}$$

$$= P\lg\left(2 + \frac{2B}{P}\right) + B\lg\left(\frac{P}{2B}\right) \tag{32}$$

$$\leqslant P\left(1 + \left(\frac{B}{P}\right)\lg(e)\right) + B\left(\lg\left(\frac{P}{B}\right) - 1\right)$$

$$(\text{since } \lg(x) \leqslant (x - 1)\lg(e)) \tag{33}$$

$$= P + B\left(\lg(e) - 1 + \lg\left(\frac{P}{B}\right)\right). \quad \square \tag{34}$$

The special case $d = 0$ is also useful as it yields $H/\lg(3) \leqslant P$, i.e. $0.63H \leqslant P$. This result is true for any binary search tree, in particular, it holds for the optimal binary

search tree. Let C_{opt} denote the weighted path length of an optimal binary search tree then by the above result $0.63H \leqslant C_{opt}$. In fact, Mehlhorn [110] has shown that this lower bound is sharp for infinitely many distributions. In the case of alphabetic trees we get a much better result than the one implied by the above result. The following result is attributed to Gilbert and Moore by Knuth (see [86]).

Theorem 2.5. *The weighted path length of an optimal alphabetic tree with* $(\alpha_0, \ldots, \alpha_n)$ *as the gap probabilities, lies between* H *and* $H + 2$.

Proof. To get the lower bound we use induction on n. Let $Q_1 = \sum_{0 \leqslant i < k} \alpha_i$ for some k. If $n > 0$ the weighted external path length is at least

$$1 + \sum_{0 \leqslant i < k} \alpha_i \lg(Q_1/\alpha_i) + \sum_{k \leqslant i \leqslant n} \alpha_i \lg((1 - Q_1)/\alpha_i) \tag{35}$$

$$\geqslant \sum_{0 \leqslant i \leqslant n} \alpha_i \lg(1/\alpha_i) + f(Q_1) \tag{36}$$

$$= H + f(Q_1), \tag{37}$$

where

$$f(Q_1) = 1 + Q_1 \lg(Q_1) + (1 - Q_1)\lg(1 - Q_1). \tag{38}$$

The function $f(Q_1)$ is nonnegative and it takes its minimum value of zero when $Q_1 = \frac{1}{2}$, hence the lower bound result follows.

To get the upper bound, Knuth [85] constructs code words C_i of 0's and 1's, using the most significant $e_i + 1$ binary digits of the function $\sum_{0 \leqslant k < i} \alpha_k + \alpha_i/2$ expressed in binary notation and shows that C_i is never an initial substring of C_j when $i \neq j$, and hence that we can construct a binary search tree corresponding to these code words. The weighted path length of the binary tree constructed by this procedure is

$$\leqslant \sum_{0 \leqslant i \leqslant n} (e_i + 1)\alpha_i < \sum_{0 \leqslant i \leqslant n} \alpha_i(2 + \lg(1/\alpha_i)) = H + 2. \qquad \square \tag{39}$$

Knuth [86] also mentions that the first part of the above proof may be extended to show that the weighted path length of every weighted extended binary tree must be greater than or equal to the entropy H of the probability distribution $(\alpha_0, \alpha_1, \ldots, \alpha_n)$. He mentions that this fundamental result is due to Claude Shannon (see [86]). Hence, we note the following:

Let C denote the cost of any weighted extended binary tree with gap probabilities $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ and C_{HUFF} the cost of any Huffman tree, C_{AT} the cost of an alphabetic tree, C_{OAT} the cost of an optimal alphabetic tree on that set.

The cost of any tree on that set satisfies

$$(1)\ H \leqslant C_{HUFF} \leqslant C, \tag{40}$$

$$(2)\ H \leqslant C_{HUFF} \leqslant C_{OAT} < H + 2, \tag{41}$$

$$(3)\ C_{OAT} \leqslant C_{AT}. \tag{42}$$

Knuth [86] has generalised the above result to show that the cost of any binary search tree on the set of access probabilities $(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_n)$ is

$$< 2 - \alpha_0 \lg(\alpha_0) - \sum_{1 \leqslant i \leqslant n} (\beta_i + \alpha_i)\lg(\beta_i + \alpha_i). \tag{43}$$

Much better upper bounds are known for the cost of the optimal binary search trees. Mehlhorn [110] showed that the cost of an optimal binary search tree on $(\alpha_0,\ldots,\alpha_n, \beta_1,\ldots,\beta_n)$ with $\Sigma\beta_i+\Sigma\alpha_j = 1$ is $<2+1.44H$. In fact, he exhibited a linear time heuristic which constructs a binary search tree for a given distribution $(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_n)$ having a cost $<2+1.44H$. Bayer [14] improved the upper bound to $H+2$. Mehlhorn [111] also exhibited another linear time heuristic and showed that the upper bound is $1+\sum\alpha_j + H$ and is best possible in the sense that if $c_1\sum\beta_i + c_2\sum\alpha_j + c_3H$ is an upper bound on the cost of an optimal binary search tree then $c_1 \geqslant 1$, $c_2 \geqslant 1$ and $c_3 \geqslant 1$. He proved this by exhibiting suitable probability distributions. Yeung [162] has derived upper and lower bounds on the cost of optimal alphabetic trees. He also mentions several other results about upper and lower bounds on the cost of Huffman trees and optimal alphabetic trees. He proposes a linear time heuristic to construct an alphabetic binary tree whose cost is

$$\begin{aligned} &\leqslant H + 2 - f(\alpha_0) - f(\alpha_n) < H + 2 - \alpha_0 - \alpha_n \\ &\text{where } f(x) = x(2 - \lg(x) - \lceil -\lg(x) \rceil). \end{aligned} \tag{44}$$

Improved bounds on the cost of optimal binary search trees and optimal alphabetic trees were obtained by Roberto de Prisco and Alfredo de Santis [129, 130]. They proposed a linear time heuristic that constructs a binary search tree whose cost satisfies the condition $C_{\text{opt}} \leqslant C < H + 1 - \alpha_0 - \alpha_n + \alpha_{\max}$ where $\alpha_{\max}$ is the maximum value among $\alpha_0, \alpha_1,\ldots,\alpha_n$. This result improves on Mehlhorn's upper bound on the cost of optimal binary search trees and also improves on Yeung's upper bound on the cost of optimal alphabetic trees.

New lower bounds on the cost of binary search trees were obtained recently by Prisco and Santis [130]. Their bounds were expressed in terms of the entropy H of the probability distribution $(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_n)$, the number of elements n and the probability $Q = \sum_{i=1}^{n}\beta_i$ that a search is successful. Their bounds exploit relations between trees and codes. Mehlhorn's lower bound of $C \geqslant H/\lg 3$ is tight when the entropy H is small. If further information on the probability distribution is given then a better [4, 5] lower bound of $C \geqslant H - \lg e - Q(\lg\lg(n+1) - 1)$ can be obtained. This bound [130] has been shown to be better than Mehlhorn's bound when $H > 3.909 + 2.710Q\lg\lg(n+1) - 2.710Q$. Prisco and Santis [130] show that in fact $C \geqslant H + H\lg H - (H+1)\lg(H+1)$. They show that this bound is better than Mehlhorn's bound when $H \geqslant \chi$ where $\chi \approx 14.4922$. They also derive several other lower bounds which are a function of H, Q and n or H and Q only. They show that the bounds $C \geqslant H - 1 - Q(\lg\lg(n+1) - 1)$ and $C \geqslant H + Q + H\log H - (H+1)\lg(H+1)$ are better than previously known bounds when the entropy, H is not very small.

It is useful to obtain bounds on the cost of binary search trees produced by heuristics, since we can compare them with the best-known upper bound on the cost of the optimal binary search trees and determine how close, they are to the optimal cost.

2.2.3. Subtree weight ratios

Hirschberg et al. [60] obtained bounds for the ratio of the weight of a subtree $S(d)$ rooted at a depth d from the root of an optimal binary search tree T, to the weight of T. They call the maximum possible value of the ratio of weights as $\rho(d)$. Their work on this problem was motivated by Mehlhorn's heuristic [113] for constructing nearly optimal binary search trees. Mehlhorn showed that a tree, that is constructed by equalising as much as possible the weights of the left and right subtrees, is nearly optimal. Hirschberg et al. [60] consider the related problem: how skewed can an optimal binary search tree be. Their result was used by Larmore [95] in his subquadratic algorithm for constructing approximately optimal binary search trees.

The following theorem was proved by Hirschberg et al. [60]. We include their proof and show why it is erroneous and also show later, how we can correct this error.

Theorem 2.6 (Erroneous: valid only if external node weights are zeroes). *If T is an optimal binary search tree, then the weight of the left or right subtree must be at most $\frac{2}{3}$ the weight of the entire tree.*

Proof. Suppose that $W(T_R) > (\frac{2}{3})W(T)$. $Root(T_R)$ has two subtrees, T_{RL} and T_{RR}. There are two possible cases:

1. The weight of T_{RL} is greater than $(\frac{1}{3})W(T)$. Then make $root(T_{RL})$ the new root of T. using a double left rotation.
2. $\beta_R + W(T_{RR}) > (\frac{1}{3})W(T)$. Then make $root(T_R)$ the new root of T, using a single left rotation.

In either case, the new tree has lower expected search time than T, a contradiction to the optimality of T. By symmetry, the same argument holds for the left subtree of T. □

The proof as given above is valid if the optimal binary search trees considered have no external node weights. However, if there are nonzero external node weights then the proof may not be valid, since we may be trying to make an external node T_{RL} or T_R as the root node.

They also prove the following lemma by arguments similar to those used in the previous theorem.

Lemma 2.7 (Erroneous: valid only if external node weights are zeroes). *In an optimal binary search tree,*

1. $\beta_0 + W(T_R) \geqslant \max\{W(T_{LL}), W(T_{LR})\}$.
2. $\beta_0 + W(T_L) \geqslant \max\{W(T_{RR}), W(T_{RL})\}$.

Their proof of this lemma also has the same error we observed before but holds for the subtrees which have only internal nodes.

Their main theorem, which uses the above lemma is the following:

Theorem 2.8 (Erroneous: valid only if external node weights are zeroes). *For any subtree S with its root at a distance d from the root of an optimal binary search tree T, $W(S)/W(T) \leqslant 2/F_{d+3}$ where F_n is the nth Fibonacci number ($F_1=1$, $F_2=1$, $F_3=2$). Hence $\rho(d) \leqslant 2/F_{d+3}$.*

By fixing the problem encountered at the leaf levels we get the following correct theorem.

Theorem 2.9. *Let T be an optimal binary search tree with a subtree S rooted at a node at a distance d from the root. If S is rooted at an internal node, $W(S)/W(T) \leqslant 1/F_{d+1}$ and if it is rooted at an external node, then $W(S)/W(T) \leqslant 1/F_d$. Hence $\rho(d) \leqslant 1/F_d$.*

Proof. We consider two cases: one when the root of the subtree S is an internal node another when the root of the subtree S is an external node.

1. Let the root of the subtree $S = T_0$ be an internal node, say β_0. We start at the root of S and go up the path to the root of T, one level at a time. At each step i, we are at the root of a bigger subtree. Let us call this subtree as T_i, and let β_i be the weight of $root(T_i)$. T_i has one subtree T_{i-1} and another subtree V_i which was not along the path followed. Also let $W(T_0) = W_0 = W(S)$ and $W(T_1) = W_1$. Since every subtree of an optimal tree is also optimal, we can use the previous lemma, observing that:

$$\beta_2 + W(V_2) \geqslant W(T_0), \tag{45}$$

$$W(T_2) = W(T_1) + \beta_2 + W(V_2), \tag{46}$$

$$W(T_2) \geqslant W(T_0) + W(T_1), \tag{47}$$

and in general

$$W(T_i) \geqslant W(T_{i-1}) + W(T_{i-2}) \quad \forall d \geqslant i \geqslant 2. \tag{48}$$

Solving this recurrence we get,

$$1 = W(T) = W(T_d) \geqslant F_d W_0 + F_{d-1} W_1 \tag{49}$$

$$\geqslant F_d W_0 + F_{d-1} W_0 \tag{50}$$

$$= W_0(F_d + F_{d-1}) \tag{51}$$

$$= W_0 F_{d+1}, \tag{52}$$

$$\frac{W_0}{W_T} = \frac{W_S}{W_T} \leqslant \frac{1}{F_{d+1}} \tag{53}$$

since $W_1 > W_0$. This proves the first part of the theorem.

2. When the root of S is an external node, we have

$$W(T_i) \geqslant W(T_{i-1}) + W(T_{i-2}) \quad \forall d \geqslant i \geqslant 3, \tag{54}$$

$$1 = W(T) = W(T_d) \geqslant F_{d-1} W_0 + F_{d-2} W_1 \tag{55}$$

$$\geqslant F_{d-1} W_0 + F_{d-2} W_0 \tag{56}$$

$$= F_d W_0. \tag{57}$$

So we get $W(S)/W(T) \leqslant 1/F_d$. □

It is useful to obtain bounds on the subtree weight ratios of the optimal binary search trees since they tell us how skewed an optimal binary search tree can be. It also gives a necessary condition to check the optimality of a binary search tree.

2.3. Fast algorithm for a special case

2.3.1. Larmore's subquadratic algorithm

Larmore [95] presented an algorithm for constructing optimal binary search trees. Using subtree weight ratios, that requires subquadratic time if there is no long sublist of very low frequency items. The time required by it is $O(n^{1.6})$ if the frequency of each item is at least ε/n for some constant $\varepsilon > 0$. He also presented a modification which constructs a nearly optimal binary search tree in the general case. We discuss this algorithm in the next subsection.

The following are his main results.

Theorem 2.10. *Let $l < n$ be an integer, and let $0 < \lambda < 1$ be a real number such that $W_{i,j} \geqslant \lambda$ for all i, j such that $j - i > l$. Then there is an algorithm that computes an optimal binary search tree in $O(n(l + \lambda^{-\log_\phi 2} \log n))$ time, where $\phi \approx 1.618$ is the golden ratio.*

Corollary 2.11. *Suppose $\beta_i + \alpha_i \geqslant \varepsilon/n$ for all i, where $\varepsilon > 0$ is a constant. Then there is an algorithm that computes an optimal binary search tree in $O(n^\sigma (\log n)^\tau)$ time, where $\sigma = 1 + 1/(1 + \log \phi) \approx 1.59023$ and $\tau = 1 - 1/(1 + \log \phi) \approx 0.40977$.*

Larmore formulates the optimal binary search tree problem as a problem on a weighted acyclic digraph. For any fixed integer $d \geqslant 0$, he shows how we can construct a weighted acyclic digraph G_d such that the minimum weight of any path in G_d from its source to its sink is the weighted path length P_{opt} of the optimal binary search tree for a set of n keys, $(K_1, \ldots, K_n)$. The edges and vertices of the graph G_d are defined as given below.

Let $\Sigma^{(d)} = \{w \in \Sigma^* \mid |w| \leqslant d\}$. We define $G_d = (V_d, E_d)$, where

$$V_d = \{\{K_1, \ldots, K_n\} \times \Sigma^{(d)}\} \cup \{source, sink\}, \tag{58}$$

$$R_d = \{((K_i,u),(K_{i+1},v)) \mid u,v \in \Sigma^{(d)}, u \Rightarrow v\}$$
$$\cup \{(source,(K_1,v)) \mid v \in 0^*\} \cup \{((K_n,u),sink) \mid u \in 1^*\}, \tag{59}$$
$$S_d = \{((K_i,u),(K_j,v)) \mid i<j;\ u,v \in \Sigma^{(d)},\ u \Rightarrow v, max\{|u|,|v|\} = d\}$$
$$\cup \{(source,(K_j,v)) \mid |v| = d\} \cup \{((K_i,u),sink) \mid |u| = d\}, \tag{60}$$
$$E_d = R_d \cup S_d. \tag{61}$$

The members of R_d are called *regular edges*, while the members of S_d are called *special edges*. The *rank* of each vertex is also defined:

$$rank(K_i,w) = i, \tag{62}$$
$$rank(source) = 0, \tag{63}$$
$$rank(sink) = n+1. \tag{64}$$

Weights are placed both on the vertices and the edges as follows: for vertices

$$weight(source) = weight(sink) = 0, \tag{65}$$
$$weight(K_i,w) = (|w|+1)\beta_i, \tag{66}$$

for the regular edges as

$$weight((K_i,u),(K_{i+1},v)) = (max\{|u|,|v|\}+1)\alpha_i, \tag{67}$$
$$weight(source,(K_1,0^k)) = (k+1)\alpha_0, \tag{68}$$
$$weight((K_n,1^k),sink) = (k+1)\alpha_n, \tag{69}$$

for the special edges as

$$weight((K_i,u),(K_j,v)) = P_{i,j} + (d+1)W_{i,j}, \tag{70}$$
$$weight(source,(K_j,v)) = P_{0,j} + (d+1)W_{0,i}, \tag{71}$$
$$weight((K_i,u),sink) - P_{i,n+1} + (d+1)W_{i,n+1}, \tag{72}$$

where $P_{i,j}$ and $W_{i,j}$ denote the path length and the total weight of all items strictly between K_i and K_j, when $i<j$.

The following lemmas show the relationship between the minimum weight path problem for the graph G_d and the optimal binary search tree problem.

Lemma 2.12. *Let χ be a path in G_d from source to sink. Then there is a binary search tree T such that $position_T(K_i) = w_i$ for each interior node (K_i,w_i) of χ. Furthermore, the weighted path length of T is the weight of χ.*

Lemma 2.13. *Let T be a search tree (not necessarily optimal), and let P be its weighted path length. Let $\mathscr{X}_d = \{(K_i,w_i) \mid w_i = position_T(K_i), |w_i| \leqslant d\}$. Then the ele-*

ments of $\mathscr{X}_d$ are exactly the interior nodes of some path χ in G_d from source to sink. Furthermore, weight$(\chi) \leqslant P$.

Lemma 2.14. *Let T_{opt} be an optimal binary search tree and let P_{opt} be the weighted path length of T_{opt}. Then for any fixed $d \geqslant 0$, P_{opt} equals the weight of a minimum weight path in the graph G_d from source to sink.*

As the graph as defined above has $n(2^{d+1}-1)+2$ vertices, $(n-1)(2^{d-2}-2d-4)+2(d+1)$ regular edges, and $(2^d-1)(n-1)(n-2)+2(n-1)(d+1)$ special edges which are not regular, it becomes necessary to take advantage of the special structure of the graph.

For any d and for any vertex x of G_d, $f_d(x)$ is defined to be the least weight of any path in G_d from *source* to x. The weight of a path was defined as the sum of the weights of the vertices and the edges of that path, the weight of the last vertex being included. Hence, $f_d(sink) = P_{opt}$. If no path exists from *source* to x, $f_d(x) = \infty$. For a fixed integer $l \leqslant n$, let $G_{d,l}$ be the subgraph of G_d consisting of all the vertices and only edges of span not exceeding l. For any vertex x, let $f_{d,l}(x) \geqslant f_d(x)$ be the least weight of any path in $G_{d,l}$ from *source* to x. The *span* of an edge is defined to be the difference of the ranks of its end points.

Larmore gives a result that uses the lemma concerning subtree weight ratios that a subtree $T_{i,j}$ of an optimal binary search tree T rooted at a depth d has weight less than $2/F_{d+3}$. In view of our result concerning subtree weight ratios, his result must be read as follows:

Lemma 2.15. *If $W_{i,j} > 1/F_d$ for all pairs i, j such that $j-i > l$, then $P_{opt} = f_{d,l}(sink)$.*

Larmore's Algorithm

Choose d, l
Compute $f_{d,l}(K_i, 0^d)$ for all i
$w \leftarrow 0^d$
while $w \neq 1^d$ do
begin
    $w \leftarrow$ the inorder successor of w in $\Sigma^{(d)}$
    Compute $f_{d,l}(K_i, w)$ for all i
end
Compute $f_{d,l}(sink)$

For any vertex y, $f_{d,l}(y) = weight(y) + min\{f_{d,l}(x) + weight(x, y)\}$ where the minimum is taken over all the edges (x, y) of the graph $G_{d,l}$. The classic minimum weight path algorithm examines all edges. Larmore's algorithm examines all regular edges, but only a small subset of the special edges.

The first step of the algorithm is to find the smallest integers d, l such that $l = \lfloor 2^d \log n \rceil$ and $W_{i,j} > 1/F_d$ (In Larmore's paper [95] a value of $2/F_{d+3}$ is given, but due to our correction we get the value $1/F_d$) for all pairs i, j such that $j - i > l$. It

requires $O(n)$ time to determine whether a particular candidate value for d is suitable, since it suffices to check $W_{i,i+l}$ for all i, and there are $O(\log n)$ values of d to check. Thus, choosing d, l requires $O(n \log n)$ time.

For any i, there is at most one edge to $(K_i, 0^d)$ and that edge is from *source*. It takes $O(n)$ time to compute $f_{d,l}(K_i, 0^d)$ for all i.

Suppose that $w \neq 0^d$, and that $f_{d,l}(K_i, u)$ has been computed for all $u \in \Sigma^{(d)}$ such that u is an inorder predecessor of v. Let w' be the inorder predecessor of w in $\Sigma^{(d)}$. Larmore defines an $n \times n$ matrix M as follows: $M[j,i] = f_{d,l}(K_i, w') + P_{i,j} + (d+1)W_{i,j}$ provided $0 < j - i \leqslant l$, and $M[j,i] = \infty$ otherwise. Then the minimum value in the jth row of M is precisely the minimum weight of any path in $G_{d,l}$ from *source* to (K_j, w) where the last edge of that path is a special edge. By the quadrangle inequality of W and P, M is monotone, i.e. the column position of the minimum entry in each row is an increasing function of the row. Hence, the minimum row values can be found by using an $O(n \log n)$ algorithm for finding the row minimums of a monotone matrix. Each regular edge is then examined once to find lower values for $f_{d,l}(K_j, w)$ if any. The total time for the main loop of the algorithm is therefore $O(2^d n \log n)$. The final step of computing $f_{d,l}(sink)$ requires examining each of the $l + d$ edges to *sink*. This step takes linear time. Therefore, the algorithm takes $O(nl + 2^d n \log n)$ time if $W_{i,j} > 1/F_d$ for all i,j such that $j - i > l$.

Now given that $W_{i,j} \geqslant \lambda$ for all pairs i,j such that $j - i > l$ we choose d such that $\lambda \geqslant 1/F_d$. Then $\phi^d \geqslant 1/\lambda$ and therefore $d \geqslant \log(1/\lambda)/(\log \phi)$ where ϕ is the golden ratio, $\phi \approx 1.618$. Hence, $2^d = 2^{-\log_\phi \lambda} = 2^{-(\log_2 \lambda)(\log_\phi 2)} = \lambda^{-\log_\phi 2}$. Therefore, $O(nl + 2^d n \log n)$ becomes $O(nl + \lambda^{-\log_\phi 2} n \log n)$ and the theorem follows.

2.4. Nearly optimal binary search trees

Knuth's algorithm [86] for constructing optimal binary search trees requires $O(n^2)$ space and runs in $O(n^2)$ time. Quite often exact probabilities of accesses are not available. So nearly optimal trees that can be obtained faster will be useful. Many heuristics [95, 102, 110, 111, 129, 162] using the properties of the optimal binary search trees of the previous subsection have been proposed for constructing trees which have nearly the optimal cost but which can be constructed much more quickly. Many of these heuristics run in linear time and require $O(n)$ space. Several of these heuristics have resulted in new upper and lower bounds on the cost of optimal trees as noted before in the previous subsection.

2.4.1. The rootmax heuristic

A simple heuristic for constructing approximately optimal binary search trees is the rootmax heuristic. The most frequently occurring key is chosen as the root and the remaining keys are attached to the root node recursively. An $O(n)$ implementation of this heuristic is possible. This is achieved by using a data structure called the "Treap" [13, 90]. A treap is a binary tree in which each node has two keys; the binary search tree property is maintained in one and the heap property is maintained in the other.

A tree constructed by rootmax heuristic has heap property on the weights and binary search tree property on the keys. Hence using the $O(n)$ algorithm to construct the treap [13], we can build the tree by rootmax heuristic.

In this heuristic the gap probabilities are ignored; this could be disadvantageous in some cases. This heuristic produces trees whose cost is very far from the optimal as can be shown by the following reasoning. Suppose that we have n keys whose frequencies are $1,2,3,\ldots,n$. Let us assume for simplicity that n is of the form $2^k - 1$. The root node is the node having the highest frequency, n. The rootmax heuristic produces a left leaning search tree. The cost of this search tree is seen to be

$$\sum_{i=1}^{n} i(n-i+1) = \frac{n(n+1)(n+2)}{6}. \tag{73}$$

The cost of the balanced search tree is seen to be

$$\sum_{i=1}^{k} i2^{k-i-2} \approx \frac{n}{2} + n^2 \log n. \tag{74}$$

Hence, the ratio of the cost of the search tree produced by the rootmax heuristic to that of the optimal binary search tree is $\geqslant n/6 \log n$.

2.4.2. *Bisection heuristic*

Mehlhorn [113] obtained a heuristic which builds a binary search tree, the weight of whose left and right subtree are nearly the same. He showed that the cost of this binary search tree is very close to the optimal.

Mehlhorn's method uses a bisection on the set

$$\left\{ s_i \,\middle|\, s_i = \sum_{p=0}^{i-1} (\alpha_p + \beta_p) + \beta_i + \alpha_i/2 \text{ where } 0 \leqslant i \leqslant n \right\},$$

i.e. the root k is chosen so that $s_{k-1} \leqslant \frac{1}{2}$ and $s_k \geqslant \frac{1}{2}$. The heuristic proceeds recursively on the subsets $\{s_i \mid i \leqslant k-1\}$ and $\{s_i \mid i \geqslant k\}$. If binary search is used to find the k at each recursive step then the running time of the heuristic is $O(n \log n)$. If the search for k is implemented using a combination of exponential and binary search, the running time of the heuristic can be reduced to $O(n)$. Mehlhorn [113] showed that the cost of the binary search tree produced by this heuristic is $< H + 1 + \Sigma\alpha_j$.

2.4.3. *Min–max heuristic*

In this heuristic we choose the root so as to minimize the maximum of the weights of the left and right subtree. The implementation and performance of this heuristic are quite comparable to that of Mehlhorn's heuristic.

2.4.4. *Larmore's heuristic*

Larmore's algorithm [95] for a special case can be modified to get a heuristic which constructs an approximately optimal binary search tree for the general case. This heuristic has one parameter, and exhibits a trade-off between speed and accuracy. It is possible

to choose the parameter so that the time required by it is $O(n^{1.6})$ and the error is $o(1)$. This heuristic is the first subquadratic method to construct an approximately optimal binary search tree, whose cost differs from that of the optimal by $o(1)$, answering a question of B. Allen [8].

Larmore's heuristic is a modification of his algorithm for constructing an optimal binary search tree in subquadratic time if there is no long sublist of low-frequency elements, which we discussed in the previous subsection. The worst case for his algorithm for constructing an optimal binary search tree is when there are a large number of low-frequency elements. The main idea behind his heuristic is to delete these sublists of low-frequency elements, and then to apply his algorithm to construct an optimal binary search tree on the remaining elements. Then a complete balanced tree on the low-frequency elements is formed and attached to the tree so constructed.

The variable parameter of this heuristic is a parameter r satisfying the inequality $0 < r < 1$. Let

$$l = \lceil n^r \rceil, \tag{75}$$

$$\delta = \frac{(\log n)^{\log \phi}}{l^{1+\log \phi}}, \tag{76}$$

$$D = \{1 \leqslant i \leqslant n \mid \alpha_{i-1} + \beta_i + \alpha_i < \delta\}. \tag{77}$$

Larmore uses his algorithm to construct an optimal binary search tree T' for the list obtained by deleting both K_i and (K_i, K_{i+1}) for all i in D. The deleted items are then organised into an almost complete binary tree and attached to the tree T'. Suppose D is the disjoint union of the maximal runs, i.e. $D = [i_1 \ldots j_1] \cup \cdots \cup [i_m \ldots j_m]$, where $i_{k+1} > j_k + 1$. For each k, let T_k be the almost complete binary search tree for the keys strictly between K_{i_k-1} and K_{j_k+1}. A binary search tree T for all the keys is then formed, by removing from T' each external node (K_{i_k-1}, K_{i_k}) and replacing it by T_k.

Larmore shows that the time complexity of his heuristic is $O(n^{1+r})$ where r is a real number between zero and one. The special feature of this heuristic is the trade-off between speed and accuracy it offers.

Let P, P' be the costs of the trees T and T', respectively, and let P_{opt} be the cost of the optimal binary search tree. Larmore shows that the accuracy offered by this heuristic is

$$P - P_{\text{opt}} \leqslant P - P' = O(n\delta \log n) = O(n^{1-r(1+\log \phi)}(\log n)^{1+\log \phi}).$$

Larmore obtained the following result.

Theorem 2.16. *For any choice of real $0 < r < 1$, there is an algorithm that computes a binary search tree T_{approx} in $O(n^{1+r})$ time such that*

$$\text{error} = P_{\text{approx}} - P_{\text{opt}} \leqslant n \left(\frac{n^r}{\log n} \right)^{-(1+\log \phi)},$$

where P_{opt} is the weighted path length of the optimal binary search tree, and P_{approx} is the weighted path length of T_{approx}.

Corollary 2.17. *For any choice of* $r > 1/(1+\log\phi) \approx 0.59023$, *the algorithm computes a binary search tree whose weighted path length differs from that of the optimal binary search tree by* $o(1)$, *in time* $O(n^{1+r})$.

The corollary follows from the theorem since if $r > 1/(1+\log\phi)$ then

$$error = P_{approx} - P_{opt} \leqslant \frac{(\log n)^{1+\log\phi}}{n^{r(1+\log\phi)-1}} = o(1). \tag{78}$$

2.5. Other results

We record a few other results known about optimal binary search trees.

The least upper bound on the cost of optimal binary search trees as a function of the number of external nodes, given the total weight of the nodes to be one, was found by Hu and Tan [66]. They showed that

$$\{(q-1)\lfloor 2^{q-1} - n/2\rfloor + q(n-2^{q-1})\}\frac{1}{\lfloor n/2\rfloor}, \tag{79}$$

where $q = \lceil \lg n \rceil$, is the least upper bound on the cost of any optimal binary search tree built on n external nodes and $n-1$ internal nodes where the total weight of the internal and external nodes is one.

Karpinski et al. [74] recently gave an algorithm with subquadratic expected running time for the special case of the optimal binary search tree problem when the α's are all zero. They also claim that they can obtain such a result for the general case. Their result is:

Lemma 2.18. *Let probabilities* $p_1, \ldots, p_n$ *be given where the* p_i *are randomly permuted. Then there is an* $o(n^2)$ *algorithm that computes a binary search tree where* p_i *are the weights of internal nodes and external nodes have weight zero, which is optimal with probability* $1 - o(1)$. *Furthermore, there is an algorithm which computes an optimal binary search tree in expected time* $o(n^2)$.

They also claim that they can obtain such a result for the general problem where the external node weights are also given.

In the dynamic version of the optimal binary search tree problem where new keys are added or existing keys are deleted, the access probabilities of the keys change is an interesting problem we have not discussed. Mehlhorn [112] studied dynamic binary search trees.

3. Huffman trees

In this section, we study the Huffman tree problem. We give the classic $O(n \log n)$ algorithm to construct Huffman trees. We note certain properties of Huffman trees and look at $O(n)$ algorithms to construct Huffman trees in some special cases. We discuss

the problem of verifying the optimality of weighted extended binary trees and Huffman trees.

3.1. The Huffman algorithm

Huffman [64, 69, 84] proposed an algorithm for constructing Huffman trees. The term Huffman trees is generally reserved for the trees produced by the Huffman algorithm although that algorithm may not produce all the trees having minimum weighted path length. The algorithm given below is recursive.

The Huffman algorithm

Given a sequence of weights $(w_1, w_2, \ldots, w_n)$, the algorithm produces a Huffman tree on that sequence of weights.

$\mathrm{HT}(w_1, w_2, \ldots, w_n)$

1. If $n = 1$ then create an external node of weight w_1 and stop.
2. Find the the two smallest values of the sequence say w_i, w_j
3. Call $\mathrm{HT}(w_1, w_2, \ldots, w_{i-1}, w_i + w_j, w_{i+1}, \ldots, w_{j-1}, w_{j+1}, \ldots, w_n)$
4. Replace an external node of weight $w_i + w_j$ in the tree obtained in the previous step by a tree with one internal node having an external node as its left-son with weight w_i and an external node with weight w_j as its right-son.

Since we need to find the two smallest values at each step, a brute force method of finding the two smallest values using a linear search will give an $O(n^2)$ time algorithm. If we use a priority queue on the w_i's for this purpose then we can delete the w_i and w_j and also insert the $w_i + w_j$ in $O(\log n)$ time, so we get an $O(n \log n)$ implementation for the Huffman algorithm.

The tree produced by Huffman's algorithm is optimal. We can show this by induction on n.

Given any binary tree with minimum weighted path length, if an internal node of maximum path length has two sons which are not the smallest nodes, then we can interchange the position of the two smallest nodes w_1 and w_2 with the sons of that internal node without increasing the cost of the tree. We then merge the two smallest weight nodes w_1, w_2 to get an external node of weight $w_1 + w_2$. In the resulting tree, we have $n - 1$ leaves with weights $w_1 + w_2, w_3, \ldots, w_n$. By the induction hypothesis this tree is optimal so the original tree is also optimal.

Schwartz [143] showed how we can construct a Huffman tree that is as well-balanced as possible (has the smallest maximum depth) among all possible Huffman trees for a given sequence of weights.

We can define t-ary trees just as we defined binary trees, where every internal node has exactly t sons and every external node has no sons. It is known (see [84]) that the Huffman construction of combining the t smallest weight nodes will give an optimum t-ary tree, except that we have to add some zero weight nodes initially so that at every later step we combine exactly t weights. The zero weight nodes do not contribute to the cost of the t-ary Huffman tree.

3.2. Properties of Huffman trees

The following well-known result (see [84]) tells us when it is possible to construct an extended binary tree given its level numbers.

Theorem 3.1. *Given a set of integers $l_1, l_2, \ldots, l_n \geqslant 0$, it is possible to construct an extended binary tree in which these numbers are the path lengths (distance from the root, of the external nodes) in some order if and only if $\sum_{1 \leqslant i \leqslant n} 2^{-l_i} = 1$.*

Proof. It is easily seen by induction on n that the condition above is necessary. Conversely, if $\sum_{1 \leqslant i \leqslant n} 2^{-l_i} = 1$ we want to construct an extended binary tree with these path lengths. If $n = 1$ then $l_1 = 0$ and the construction is trivial. Otherwise, we may assume that the l's are ordered so that

$$l_1 = l_2 = \cdots = l_q > l_{q+1} \geqslant l_{q+2} \geqslant \cdots \geqslant l_n > 0 \tag{80}$$

for some q with $1 \leqslant q \leqslant n$. Now,

$$2^{l_1 - 1} = \sum_{1 \leqslant i \leqslant n} 2^{l_1 - l_i - 1} \tag{81}$$

$$= \frac{q}{2} + \text{an integer (so } q \text{ is even)}. \tag{82}$$

By induction on n there is a tree with path lengths $l_1 - 1, l_3, \ldots, l_n$; take such a tree and replace one of the external nodes at level $l_1 - 1$ by a tree with one internal node having two external nodes as its children, this gives us the required result. □

An interesting result by Schwartz and Kallick [144] is the following.

Theorem 3.2. *Given a sequence of n weights $w_1, w_2, \ldots, w_n$ such that $w_1 \leqslant w_2 \leqslant \cdots \leqslant w_n$ there is an extended binary tree on this sequence of weights such that it minimises the weighted path length and for which the terminal nodes in inorder contain the values $w_1, w_2, \ldots, w_n$.*

Proof. We note that we are using the term extended binary tree instead of Huffman tree because there may be trees having the minimum weighted path length which are not produced by Huffman's construction. First we construct a tree by Huffman's algorithm for the sequence of weights $w_1, w_2, \ldots, w_n$. If $w_j < w_{j+1}$ then $l_j > l_{j+1}$ (or else the tree would not be optimal). The construction given in the proof of the previous theorem now gives us another tree with the same lengths and with the weights in the proper sequence. □

It is not hard to show that there is an $\Omega(n \log n)$ lower bound on the running time of any algorithm for constructing a Huffman tree on a sequence of n weights, in the decision tree model. Hence, the optimality of the Huffman algorithm in this model follows. However, we shall see in the next subsection that by choosing a model of

computation in which we can sort a sequence of numbers faster, we can construct Huffman trees much faster.

3.3. Fast algorithms in special cases

3.3.1. When the weights are in sorted order

A linear-time algorithm for constructing a Huffman tree when the input weights are in sorted order is known [113].

Theorem 3.3. *Huffman trees can be constructed in* $O(n)$ *time if the weights are in sorted order.*

Proof. As we combine the two smallest weight nodes at each step, the weights of the resulting new nodes also come in sorted order. Hence by maintaining two sorted lists (one of external nodes and one of internal nodes), we need to check only three elements to find the smallest weight pair.

In particular, if we are able to sort a sequence of n positive numbers in $F(n)$ time we can construct a Huffman tree on this sequence by just spending an extra $O(n)$ time. Hence if we are able to sort a sequence of n positive numbers in linear time, we can construct a Huffman tree on that sequence in linear time. □

A corollary of this result is

Corollary 3.4. *We can construct a Huffman tree corresponding to a* valley sequence *in* $O(n)$ *time where a* valley sequence *is a sequence of the type* $w_1 > w_2 > \cdots > w_{j-1} \leqslant w_j \leqslant w_{j+1} \ldots \leqslant w_n$.

3.3.2. When the weights are within a factor of two

It is known [116] that if the input weights are within a factor of two, we can construct a Huffman tree on these weights in linear time.

We state the following theorem due to Klawe and Mumey [80, 116].

Theorem 3.5. *Given a sequence of n nodes whose weights are within a factor of two, after the first* $[(n+1)/2]$ *minimum weight pairs have been found and combined the new sequence will consist of* $[n/2]$ *nodes whose weights are again within a factor of two.*

We sketch the main ideas behind this result.

Let the initial sequence of n nodes be $v_1, v_2, \ldots, v_n$ and let c be a real number such that $c \leqslant w(v_i) < 2c$ for $i = 1$ to n. Whenever two nodes forming the minimal weight pair are combined, the weight of the new node formed is greater than $2c$, so it will not be involved in a minimal weight pair combination until there are no two nodes whose weights are less than $2c$. When n is odd, after $(n-1)/2$ pairings have occurred, there will be only one node left, whose weight is less than $2c$, it is the largest weight node

initially present. This forms a minimal weight pair with the smallest newly formed node. When n is even, the largest weight node present in the original sequence merges with another original node. Thus during the $[(n+1)/2]$th pairing the largest weight node present initially, forms a minimal weight pair. Klawe and Mumey [80, 116] show that at this stage the weights will again be within a factor of two.

Klawe and Mumey [80, 116] extend this result to show that if we keep combining minimum weight pairs, the resulting tree will be balanced with the leaves differing in level by at most one. It will be seen that $2(n-2^{\lfloor \lg n \rfloor})$ smallest weights will be at level $\lfloor \lg n \rfloor + 1$ and the others are at level $\lfloor \lg n \rfloor$. Thus the Huffman tree can be constructed in linear-time. They also extend this result to optimal alphabetic trees, which we will see in the next section.

3.4. *Verifying Huffman trees*

Though there is an $\Omega(n \log n)$ lower bound for constructing Huffman trees in the comparison model, it will be interesting to see whether we can verify whether a given weighted extended binary tree is a Huffman tree in $o(n \log n)$ time. We first develop necessary conditions for a given weighted extended binary tree to be optimal, we then show that these conditions can be tested in $O(n)$ time.

Theorem 3.6. *A weighted extended binary tree on a sequence of n weights $w_1, w_2, \ldots, w_n$ is optimal only if for all levels l*

1. *The weight of any node at a level l is greater than or equal to the weight of any node at level $l+1$.*
2. *For any three nodes a, b, c in level l, $w(c) \leqslant w(a) + w(b)$.*

Proof. The first condition is necessary since otherwise we can interchange the position of any two nodes violating this condition, and obtain a tree of cheaper cost. We now show that the second condition is also necessary. Suppose it is not so, then at some level l, there are three nodes a, b, c such that $w(c) > w(a) + w(b)$. Since the cost of the tree does not change if we interchange weights at a given level, we can permute the weights at the level l so that the nodes a and b are siblings and recompute the weights of the internal nodes. Now, the nodes of weight $w(a) + w(b)$ which is at level $l-1$ and $w(c)$, do not satisfy the first condition (which is a necessary condition). Hence, the second condition is also a necessary condition. □

Now though it appears that condition 2 implies condition 1, if an external node appears in a level above the current level, condition 2 need not imply condition 1 for the external node. It is easy to show that the above necessary conditions can be verified in $O(n)$ time.

Theorem 3.7. *Given a weighted extended binary tree, we can test the conditions given in the previous theorem in $O(n)$ time.*

Proof. For every level l, compute the two smallest weights a_l, b_l and the maximum weight c_l. Then for every level l,

1. Check if $a_l \geqslant c_{l+1}$ and
2. if $a_l + b_l \geqslant c_l$

If at any level l either of the two conditions are not satisfied then we know that the tree is not optimal due to our previous theorem.

It is not hard to show that the running time of the above algorithm, if we use linear-time algorithms for selection, is $O(n)$ where n is the number of nodes in the tree because at any level we have to spend time proportional to the number of nodes in that level. □

It is known [87] that a binary tree on the weights $(w_1, \ldots, w_n)$ is a Huffman tree if it satisfies the following properties:

1. The n external nodes have been assigned the weights $(w_1, \ldots, w_n)$ in some order, and each internal node has been assigned a weight equal to the sum of the weights of its two children.

2. The $2n-1$ nodes (external and internal) can be arranged in a sequence $y_1, \ldots, y_{2n-1}$ such that if x_j is the weight of node y_j we have $x_1 \leqslant \cdots \leqslant x_{2n-1}$, and such that nodes y_{2j-1} and y_{2j} are siblings (children of the same parent), for $1 \leqslant j < n$, where this common parent node does not precede y_{2j-1} and y_{2j} in the sequence.

Knuth [87] shows by induction that such a tree is one that Huffman's algorithm might construct.

Ramanan [137] showed that the necessary conditions we derived for a given weighted extended binary tree to be optimal are in fact sufficient when applied to a complete binary tree. Ramanan [137] mentions that it is possible to check whether a given tree is a Huffman tree in linear-time by using his conditions for the optimality of alphabetic trees.

It would be interesting to devise a linear-time algorithm for verifying the optimality of a weighted extended binary tree using our necessary conditions and also Knuth's [87] and Ramanan's [137] conditions for Huffman trees.

4. Optimal alphabetic trees

In this section, we study the optimal alphabetic tree problem. We look at the $O(n \log n)$ Hu–Tucker and Garsia–Wachs algorithms for constructing optimal alphabetic trees. We also note the equivalence of the two algorithms. We discuss algorithms for constructing optimal alphabetic trees in some special cases. We also study the lower bound result for constructing optimal alphabetic trees obtained by Klawe and Mumey.

4.1. The Hu–Tucker algorithm

The Hu–Tucker algorithm [64] and their variants begin by building an intermediate tree (called the **lmcp tree**) on the input weight sequence. The levels of the leaves

in this intermediate tree are recorded and then they are used to build an alphabetic tree where each leaf is at the same level as in the lmcp tree. Thus, the cost of the alphabetic tree is the same as that of the intermediate (possibly nonalphabetic) tree. The intermediate tree is proved to have optimal cost in a class of trees which contains all the alphabetic trees, so it follows that the alphabetic tree constructed is indeed optimal.

The Hu–Tucker algorithm begins with a list of leaf nodes containing the weights $w_1, w_2, \ldots, w_n$ in order. This list is called the *worklist* and is used to determine how the nodes combine to form the intermediate tree. Nodes in the worklist are designated either as *crossable* or *noncrossable*. This affects the way the nodes may pair off. Initially all nodes are noncrossable. When the nodes are paired off, the resulting internal parent node is designated crossable. The weight of the parent node is assigned the sum of the weights of its children. The nodes that are paired off are removed from the worklist and the new parent node occupies the position of its left child. We say that two nodes in the list are *compatible* if they are adjacent in the worklist or if all the nodes which separate them are crossable nodes. We will use the symbol v_x to refer to a node in the worklist and w_x its weight. We denote l_x for the level of that node. We define an order on the nodes in the worklist by $v_x < v_y$ if $w_x < w_y$ or if $w_x = w_y$ and v_x is to the left of v_y in the list. A compatible pair of nodes (v_a, v_b) is said to be *local minimum compatible pair* (*lmcp*) if and only if

1. $v_b \leqslant v_x$ for all nodes v_x compatible with node v_a,
2. $v_a \leqslant v_y$ for all nodes v_y compatible with node v_b.

To obtain the lmcp tree, we keep combining the minimum compatible pairs according to Hu and Tucker [68], any local minimum compatible pairs according to Hu [63]. In fact, it was shown later [64] that we can combine the lmcps in any order since the lmcp tree is unique. We state the Hu–Tucker algorithm.

Hu–Tucker algorithm

1. Given the initial sequence of nodes $v_1, v_2, \ldots, v_n$ form n priority queues, one for each node; consisting of all nodes compatible to the given node.

2. Since all the nodes are compatible within a given priority queue the two smallest nodes at the top of the queue will be the only candidates for being an lmcp.

To find an lmcp we can use a stack-based method. Beginning with the leftmost queue, maintain a pointer to the current queue being considered. By checking neighbouring nodes we can determine in constant time whether or not the current pair is an lmcp. If it is so, we combine the two nodes and place the resulting node in the place previously occupied by the leftmost node; otherwise move the pointer backward.

Combining two nodes will result in the merger of several queues. We must choose a data structure for representing the queues so that they can be merged $O(\log n)$ time. Leftist trees (see [86]) are useful for this purpose. After each lmcp combination updating the queue structure requires $O(\log n)$ time. Hence, forming the lmcp tree this way takes $O(n \log n)$ time. We simply record the level numbers of the nodes. The lmcp tree obtained this way is proved to have the same cost as an optimal alphabetic tree but it

need not be alphabetic because the leaves could have appeared in an order which is not alphabetic.

3. From the lmcp tree, to construct the optimal alphabetic tree we require [141] only linear-time. This requires only using the level numbers of the lmcp tree. This is done as follows: When the level numbers $l_1, l_2, \ldots, l_n$ are known we scan the sequence of level numbers from left to right and locate the leftmost maximum level number, say $l_i = q$. Then $l_{i+1} = q$ also since the level sequence $l_1, l_2, \ldots, l_n$ has the property that the maximum level numbers are always adjacent. We create the father of the pair with level q and assign the father with the level $q-1$. In other words, we replace the level sequence

$$l_1, l_2, \ldots, l_{i-1}, q, q, l_{i+2}, \ldots, l_n$$

by

$$l_1, l_2, \ldots, l_{i-1}, (q-1), l_{i+2}, \ldots, l_n.$$

Then we repeat the same process of combining maximum-level adjacent pairs to the level sequence of $n-1$ numbers. Finally, we create the root with level zero.

The proof of correctness of the Hu–Tucker algorithm as given in the original paper [68] is quite involved so we sketch the ideas used.

We note that the algorithm consists of two major steps

1. combining the lmcps to get the the lmcp tree,
2. constructing an alphabetic tree using the lmcp tree level numbers.

The algorithm must satisfy the conditions

1. The level numbers of the lmcp tree must be realisable by an alphabetic tree.
2. The lmcp tree must be optimum within a class of trees that includes all alphabetic trees.

The crux of the proof is in proving the second condition.

We note the equivalence of the Hu–Tucker and Garsia–Wachs algorithms in Section 4.3 and look at the simple proof of correctness of the Garsia–Wachs algorithm as given by Kingston [78]. This gives another proof of correctness of the Hu–Tucker algorithm.

4.2. The Garsia–Wachs algorithm

In the combination phase of the Hu–Tucker algorithm we successively combine lmcps while the pair in consideration can be separated by many crossable nodes (nodes obtained as a result of combinations). The Garsia–Wachs algorithm eliminates the distinction between crossable and non-crossable nodes (nodes which have not resulted due to a combination step) and arranges the weight sequence such that the lmcp is always an adjacent pair. We note that in a sequence of non-crossable nodes an adjacent pair (w_{j-1}, w_j) is a lmcp if and only if $w_{j-2} > w_j$ and $w_{j-1} \leqslant w_{j+1}$. Before describing the Garsia–Wachs algorithm we give a definition (see [78]).

Definition 1. A pair of leaves α_{i-1}, α_i is right minimal if

1. $1 < i \leqslant n$,

2. $\alpha_{i-2}+\alpha_{i-1}\geqslant\alpha_{i-1}+\alpha_i$ and
3. $\alpha_{i-1}+\alpha_i<\alpha_{j-1}+\alpha_j$ for all $j>i$.

Garsia–Wachs algorithm. The Garsia–Wachs algorithm also constructs initially a minimal tree T_B quite similar to that of lmcp tree of Hu–Tucker. Once this is done, the levels of the α_i in T_B may be used to construct T, an optimal alphabetic tree as we described earlier.

To construct the minimal tree the following two steps are repeated (for $n-1$ times) until only node remains.

1. Locate the rightmost right minimal pair of entries. Let that be α_{i-1},α_i.
2. Locate the first entry to the right of α_i that is greater than or equal to $\alpha_{i-1}+\alpha_i$. Let that be α_{i+k+1}. Then the new list is $\alpha_1,\ldots,\alpha_{i-2},\alpha_{i+1},\ldots,\alpha_{i+k},(\alpha_{i-1}+\alpha_i),\alpha_{i+k+1},\ldots,\alpha_n$.

4.2.1. Implementation

An $O(n\log n)$ implementation of Garsia–Wachs algorithm was given by Garsia and Wachs [43]. An algorithm requiring $O(n\log n)$ comparisons is easily obtained. Given $\alpha_1,\alpha_2,\ldots,\alpha_n$ we first find the biggest i such that $\alpha_{i-2}+\alpha_{i-1}\geqslant\alpha_{i-1}+\alpha_i$. Next we locate the first α_j with $j>i$ such that $\alpha_j>\alpha_{i-1}+\alpha_i$, remove α_{i-1},α_i and insert $\alpha_{i-1}+\alpha_i$ just before α_j (or at the end if such an α_j does not exist). We repeat the above steps on the new list of numbers. The first step can take $O(n)$ as it involves scanning through the list to find the rightmost rightminimal pair. Since $\alpha_{i-2}<\alpha_j$ for $j=i+1$ to n, the second step using binary insertion requires only $O(\log n)$ comparisons. Since the search for the next rightminimal pair starts from where we left off, scan is $O(n)$ time in total. Since there are $n-1$ steps requiring $O(\log n)$ comparisons, therefore $O(n\log n)$ comparisons are required for the whole algorithm. Garsia and Wachs [43] also describe an algorithm due to Tarjan which requires only $O(n\log n)$ time, including data moves and pointer manipulations using balanced trees.

4.2.2. Proof of correctness

The proof of correctness of the Garsia–Wachs algorithm as given by its inventors is complicated. A simpler proof was found much later by Kingston [78]. We sketch the main results that led to Kingston's proof.

Definition 2. If $T_1,\ldots,T_k$ are binary trees, we denote by $F(T_1,\ldots,T_k)$ the set of all trees with k leaves, but with those leaves replaced by $T_1,\ldots,T_k$ from left to right. For a fixed i, we define T_x by a tree with one internal node having as left son; an external node of weight α_{i-1}, as right son an external node of weight α_i and define

$$F=F(\alpha_1,\ldots,\alpha_n),$$
$$F_0=F(\alpha_1,\ldots,\alpha_{i-2},T_x,\alpha_{i+1},\ldots,\alpha_n),$$
$$F_k=F(\alpha_1,\ldots,\alpha_{i-2},\alpha_{i+1},\ldots,\alpha_{i+k},T_x,\alpha_{i+k+1},\ldots,\alpha_n).$$

Definition 3. The level of any leaf α_i is denoted by h_i. The root of T_x is a node α_x of level h_x ($\alpha_x = \alpha_{i-1} + \alpha_i$). The weight of $T \in F, F_0$ or F_k is

$$w(T) = \sum_{i=1}^{n} h_i \alpha_i \tag{83}$$

and we define

$$w(S) = \min_{T \in S} w(T) \tag{84}$$

for any set S of trees. A tree $T \in S$ is called *minimal* for S if $w(T) = w(S)$.

Definition 4. Let $U \in F(\alpha_1, \ldots, \alpha_n)$ and $V \in F(\gamma_1, \ldots, \gamma_n)$. Let h_i be the level of α_i in U, and k_i be the level of γ_i in V. U is a rearrangement of V (briefly $U \sim V$), if there is a permutation $(\sigma_1, \ldots, \sigma_n)$ of $(1, 2, \ldots, n)$ such that $\alpha_i = \gamma_{\sigma_i}$, $h_i = k_{\sigma_i}$ for $1 \leqslant i \leqslant n$. Informally, a rearrangement merely moves leaves around within a tree, without altering their level. Consequently, " $\sim$ " is an equivalence relation, and $U \equiv V$ implies $w(U) = w(V)$.

The main theorem proved by Kingston [78] is the following:

Theorem 4.1. *Let α_{i-1}, α_i be the rightmost rightminimal pair, and let $k \geqslant 0$ be such that*

$$\alpha_{i+j} < \alpha_{i-1} + \alpha_i \quad \text{for } 1 \leqslant j \leqslant k \text{ and } \alpha_{i+k+1} \geqslant \alpha_{i-1} + \alpha_i.$$

Then $w(F) = w(F_k)$ and every minimal tree for F_k has a rearrangement in F.

We now state the lemmas leading to the proof of this theorem.

Lemma 4.2. *Suppose we have a sequence of at least three nodes $\alpha_a, \alpha_{a+1}, \ldots, \alpha_b$ such that*

$$\alpha_{j-1} + \alpha_j \leqslant \alpha_j + \alpha_{j+1} \quad \text{for } a < j < b.$$

Then

$$h_a \geqslant h_{a+1} \geqslant \cdots \geqslant h_{b-1}.$$

Lemma 4.3. *If α_{i-1}, α_i is the rightmost rightminimal pair, then $h_i \geqslant h_{i+1} \ldots \geqslant h_n$ in every minimal tree.*

Lemma 4.4. *If α_{i-1}, α_i is the rightmost rightminimal pair, then $h_{i-1} = h_i$ in some minimal tree.*

Lemma 4.5. *Let α_{i+k+1} be the first entry to the right of the rightmost rightminimal pair α_{i-1}, α_i such that $\alpha_{i+k+1} \geqslant \alpha_{i-1} + \alpha_i$. Then in some minimal tree T for which*

$h_{i-1} = h_i$, *either*

1. $h_{i+k} = h_i - 1$; *or*
2. $k_{i+k} = h_i$ *and* α_{i+k} *is a right child.*

Lemma 4.6. *Let T be a minimal tree of F. Then there is a tree in T_k such that $w(T_k) \leqslant w(T)$.*

Lemma 4.7. *Let T_k be any minimal tree for F_k. Then $w(T_k) \geqslant w(F)$, and if equality holds, T_k has a rearrangement in F.*

The proof of all these lemmas simply involve rotating (once or twice) or permuting nodes at some level of the minimal tree to obtain a tree satisfying the claim of the lemma.

4.3. Proof of equivalence of the two algorithms

The Garsia–Wachs algorithm can be considered as a modification of the Hu–Tucker algorithm. The main observation made by Garsia and Wachs is that crossable nodes may be moved past smaller nodes, regardless of whether the latter are crossable or not. This follows from the fact that the smaller nodes will combine and become crossable before the moved crossable node will be involved in an lmcp. By moving the newly formed nodes carefully, they make sure that all the lmcp combinations are between adjacent nodes, and hence no information about crossability of nodes needs to be maintained. Hence intuitively we would expect the Garsia–Wachs and the Hu–Tucker algorithms to produce the same intermediate tree. Finding the lmcp is the major activity while inserting the elements is not in the Hu–Tucker algorithm. However, in the Garsia–Wachs algorithm adjacent nodes are compatible eventhough much time is spent in inserting the elements. The equivalence of Garsia–Wachs and Hu–Tucker algorithms proved formally, as given below is attributed by Hu [64] to Kuo. Consider the following version of the Garsia–Wachs algorithm (see [64]) to construct the minimal tree.

Given a sequence of weights $\alpha_1, \alpha_2, \ldots, \alpha_n$:

1. Find the leftmost minimal adjacent pair, α_{j-1}, α_j
2. Combine α_{j-1} and α_j as a single node with weight $\alpha_{j*} = \alpha_{j-1} + \alpha_j$
3. Move α_{j*} to the left, skipping over all nodes with weight less than or equal to α_{j*}. Obtain the new working sequence of $n-1$ nodes

$$\alpha_1, \ldots, \alpha_i, \alpha_{j*}, \alpha_{i+1}, \ldots, \alpha_{j-2}, \alpha_{j+1}, \ldots, \alpha_n, \text{ where}$$

$$\alpha_i > \alpha_{j*} \geqslant \max(\alpha_{i+1}, \ldots, \alpha_{j-2}). \tag{85}$$

Repeat the process until we get only one node in the node sequence. This is the tree which may be non-alphabetic but having the same cost as the optimal alphabetic tree on the sequence of weights $(\alpha_1, \ldots, \alpha_n)$. Having obtained the level numbers of a minimal tree we can construct an optimal alphabetic tree as described earlier.

The two algorithms will obtain the same tree after their completion as shown below. Consider the following two sequences (A) and (B)

(A) $\alpha_1, \alpha_2, \ldots, \alpha_i, \alpha_{j*}, \alpha_{i+1}, \ldots, \alpha_{j-2}, \alpha_{j+1}, \ldots, \alpha_n$ where all the nodes are noncrossable nodes.

(B) $\alpha_1, \alpha_2, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_{j-2}, \alpha_{j*}, \alpha_{j+1}, \ldots, \alpha_n$ where all the nodes are noncrossable nodes except α_{j*} which is a crossable node.

Note that (A) consists of $n-1$ noncrossable nodes and is derived from a sequence of n noncrossable nodes after one combination of Garsia–Wachs and (B) consists of $n-2$ non-crossable nodes and one crossable node and is derived from the same sequence after one combination of Hu–Tucker. We will show that if we apply the Hu–Tucker algorithm to both (A) and (B), they will give the same tree, i.e. all the combinations will be identical. (If a node α_p combines with α_{j*} of (A), the same node α_p will combine with α_{j*} in (B)).

In fact, we can show that the tree constructed by any number of combinations of Garsia–Wachs followed by Hu–Tucker will give the same lmcp tree as defined in the Hu–Tucker algorithm.

To show that (A) and (B) give the same tree T, we can make three observations about merging of the lmcp in the sequence (B).

1. In the subsequence of nodes $\alpha_1, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_{j-2}$ a node cannot be merged before a node to its right is merged.

This is because $\alpha_{j-1} + \alpha_j$ $(= \alpha_{j*})$ is the leftmost minimal adjacent pair.

2. When α_{j*} is merged, $\alpha_{i+1}, \ldots, \alpha_{j-2}$ have all been merged (or merged with α_{j*}). If this is not true, let α_l be the rightmost node among $\alpha_{i+1}, \ldots, \alpha_{j-2}$ which has not been merged. From 1 all the nodes between α_l and α_{j*} have been merged and α_l is compatible to any node compatible to the crossable node α_{j*} in (B) and $\alpha_l \leqslant \alpha_j*$. Hence α_{j*} cannot form an lmcp with another node.

3. When an lmcp is between a node in $\alpha_1, \ldots, \alpha_i$ and a node in $\alpha_{i+1}, \ldots, \alpha_n$ then α_{j*} has been merged.

Let the lmcp be (α_x, α_y) where α_x is either a node in $\alpha_1, \ldots, \alpha_i$ or a sum of at least two nodes in $\alpha_1, \ldots, \alpha_i$. Since $\alpha_{j*} = \alpha_{j-1} + \alpha_j$ is the leftmost minimal adjacent pair

$$\alpha_1 + \alpha_2 > \alpha_2 + \alpha_3 > \cdots > \alpha_{i-1} + \alpha_i \tag{86}$$

therefore,

$$\alpha_1 > \alpha_3 > \alpha_5 > \cdots \tag{87}$$

and

$$\alpha_2 > \alpha_4 > \alpha_6 > \cdots \tag{88}$$

Thus one of the two adjacent nodes is larger than α_i. Thus $\alpha_x > \alpha_i \geqslant \alpha_{j*}$. Let α_y be the right node. From 1 when α_{j*} is merged all the nodes between α_y and α_{j*} have been merged and $\alpha_y \equiv \alpha_{j*}$. But $\alpha_x > \alpha_i > \alpha_{j*}$, this contradicts that (α_x, α_y) is a lmcp, if α_{j*} has not been merged.

From the above three observations we can consider the sequences (A) and (B). Before α_{j*} in either is merged, from 3 the sequences (A) and (B) can do exactly the same combinations since $\alpha_1, \ldots, \alpha_i$ are not involved and α_{j*} is a crossable node in (B). From 2 when α_{j*} is merged in (B), all the nodes between α_i and α_{j+1} are crossable nodes. So the noncrossable node α_{j*} in (A) and the crossable node α_{j*} in (B) are compatible to the same set of nodes. After α_{j*} is merged,the sequences (A) and (B) result in the same node sequence.

4.4. Fast algorithms in special cases

4.4.1. When the input is a valley sequence

Consider a weight sequence $w_1, w_2, \ldots, w_{j-1}, w_j, \ldots, w_n$, where

$$w_1 > w_2 > \cdots > w_{j-1} \leqslant w_j \leqslant w_{j+1} \leqslant \cdots \leqslant w_n. \tag{89}$$

In other words, the weights are first decreasing and then increasing. Such a weight sequence is called a *valley sequence*. As two special cases of a valley sequence, we have

$$w_1 > w_2 > \cdots > w_n, \tag{90}$$

$$w_1 \leqslant w_2 \leqslant \cdots \leqslant w_n. \tag{91}$$

The notion of *valley sequence* was defined by Hu [64], who obtained the following results.

Lemma 4.8. *If the weight sequence is a valley sequence, then the cost of the optimal alphabetic tree is the same as the cost of the optimum tree without the alphabetic constraint.*

Proof. Assume that the weight sequence is a valley sequence as defined above and the lmcp is $w_{j-1} + w_j$ (or $w_{j-2} + w_{j-1}$). Then $w_{j-1*,j}$ is a crossable node and the next minimum weight pair may be one of the following five pairs: $w_{j-2} + w_{j+1}$, $w_{j-3} + w_{j-2}$, $w_{j+1} + w_{j+2}$, $w_{j-2} + w_{j-1*,j}$, $w_{j-1*,j} + w_{j+1}$.

In any case the node constructed say w_A is a crossable node. In general, let $w_A \leqslant w_B \leqslant \cdots \leqslant w_G$ be crossable nodes created, while on the left we have $w_1 > w_2 > \cdots > w_{j-2}$ and on the right we have $w_{j+1} \leqslant \cdots \leqslant w_n$. Then the next lmcp (the only one) is one of the following six pairs: $w_{j-2} + w_{j+1}$, $w_{j-3} + w_{j-2}$, $w_{j-1} + w_{j+2}$, $w_{j-2} + w_A$, $w_A + w_{j-1}$, $w_A + w_B$.

So the next crossable node created, w_H is again compatible with all the crossable nodes created so far. In other words, the minimum weight compatible pair is always the minimum weight pair. Hence the cost of the optimal alphabetic tree is the same as Huffman's tree. □

Lemma 4.9. *We can construct an optimal alphabetic tree for a valley sequence in* $O(n)$ *time.*

Proof. The proof is due to the result in Section 3.3. □

4.4.2. When the weights are within a factor of two

Klawe and Mumey [80, 116] introduce a new technique for finding optimal alphabetic trees. The input weights w_i are first classified according to their order of magnitude, base 2. They define a *category* of a node of weight w to be $\lfloor \lg(w) \rfloor$. A maximal length sequence in the *worklist* of weights with the same category is called a *region*. By keeping a stack of regions and considering only regions whose adjacent regions have a higher category, we can restrict most of our attention to the pairings occurring within these regions. They call this as *region-processing*. Their idea is motivated by the situation when the input weights are within a factor of two. In this case the optimal alphabetic tree is the same as Huffman's tree. Hence they use the result we described in Section 3.3.

Theorem 4.10. *There is a linear-time algorithm for finding an optimal alphabetic tree on a sequence of input weights which differ at most by a factor of two.*

Proof. 1. Initialise the worklist to contain the original input sequence. Note that all the nodes are noncrossable.

2. Use a stack-based method to find lmcps and pair them off, removing each pair of nodes from the worklist and placing the parent in a temporary list but not in the worklist. These newly formed nodes are to be left out of the worklist because their weights are greater than the weight of any of the original weights and hence need not be considered in the search for lmcps. This process continues until there are zero or one nodes left in the worklist the stack-based algorithm requires only $O(n)$ time because of the absence of crossable nodes in the worklist. If a single node x remains (n is odd) scan through the temporary list of newly formed crossable nodes to find the smallest node y. Pair x with y and replace y in the temporary list by its parent.

3. At this stage we have $m = \lceil n/2 \rceil$ crossable nodes in the temporary list. Moreover, the new nodes are still within a factor of two, by the same argument as in the proof of the Theorem 3.4. Now as all the nodes are crossable, the optimal alphabetic tree is the same as Huffman tree for these nodes. As the weights are again within a factor of two we can find the lmcp tree for these weights in $O(n)$ time using the algorithm of Section 4.1. □

4.4.3. When the weights are exponentially separated

Klawe and Mumey [80, 116] define an input weight sequence $w_1, w_2, \ldots, w_n$ to be *exponentially separated* if there exists a constant C such that for all n,

$$|\{i: \lceil \lg w_i \rceil = k\}| < C \quad \text{for all } k \in Z.$$

They also give an $O(n)$ algorithm for constructing an optimal alphabetic tree when the input weights are exponentially separated. They use their idea of region processing for this purpose. They observe that there are at most $2C$ nodes in any region processed.

They also show that every region of size r can be processed in $O(r)$ time and use their region processing method to construct the lmcp tree in $O(n)$ time. Given the level numbers of the leaves of the lmcp tree we can construct an optimal alphabetic tree in $O(n)$ time as described in Section 4.1.

4.4.4. When the input weights are small integers

Recently, Larmore et al. [98] have obtained an $O(n\sqrt{\log n})$ algorithm for the optimal alphabetic binary tree problem when the input weights are integers in the range $[0, n^{O(1)}]$. They also give an $O(n \log k)$ time algorithm for the general optimal alphabetic binary tree problem where the parameter k is bounded by the number of local minima in the input sequence.

4.5. Lower bound for constructing lmcp trees

It has been shown by Klawe and Mumey [80, 116] that constructing the intermediate lmcp tree produced by the Hu–Tucker-based algorithms in any model of computation is at least as hard as sorting in that model. Given an unsorted list of n numbers we can transform the problem of sorting the list into a problem of constructing a tree in $O(n)$ time. They show that from the information recorded in the structure of the tree produced we can compute the sorted order of numbers in $O(n)$ time. This gives an $\Omega(n \log n)$ lower bound for constructing the lmcp tree in the comparison model. The following lemma is used by them.

Lemma 4.11. *Let $x_1, x_2, \ldots, x_n$ be distinct real numbers drawn from $[2,4)$. Let $y_i = \frac{1}{2}x_{\lfloor i/2 \rfloor} + 1$, for $i = 1$ to $2n$. If $(y_1, \ldots, y_{2n})$ is given as input to any lmcp finding algorithm, the set of the first n lmcps found, disregarding order, will be $\{(y_1, y_2), (y_3, y_4), \ldots, (y_{2n-1}, y_{2n})\}$.*

Their result is

Theorem 4.12. *Sorting can be reduced to finding the lmcp tree in $O(n)$ time.*

Proof. Assume n is even. Let $x_1, \ldots, x_n$ be drawn from $[2,4)$. Define the y_i as above and consider the behaviour of some lmcp combining algorithm on the input sequence $y_1, \ldots, y_{2n}$. According to the previous lemma, after n lmcps have been combined there will be n nodes present in the node list. They will have weights $x_1, \ldots, x_n$. Since these nodes are all crossable, there will be only one lmcp present, the smallest pair of nodes in $\{x_1, \ldots, x_n\}$. This pair will combine to form a new node having a weight of at least 4. The next lmcp will be the second smallest pair of nodes from $\{x_1, \ldots, x_n\}$ and so on. Hence, the next $n/2$ lmcps found after the first n lmcp combinations have occurred sort $\{x_1, \ldots, x_n\}$ by pairs (only consecutive pairs may need to be switched in order for the list to be totally sorted). This information can be easily recovered from an lmcp tree produced by any method of searching it depth-first, always searching the least weight subtree first. We will encounter the nodes corresponding to $\{x_1, \ldots, x_n\}$ in fully sorted

order (a node with weight x_i will be the parent of leaf nodes with weights y_{2i-1} and y_{2i}). Hence, we can reduce sorting to find the lmcp tree in $O(n)$ time. □

4.6. *Verifying optimal alphabetic trees*

We mentioned in the last section that Huffman trees can be verified in $O(n)$ time. We noted that the best-known algorithms for constructing optimal alphabetic trees run in $O(n \log n)$ time while we have only a $\Omega(n)$ lower bound for this problem in the decision tree model. It is will be interesting to see whether we can close the gap between the lower bound and the upper bound. In this direction we note that if we are able to construct an optimal binary search tree on a given set of keys, then we can test whether a given binary search tree for that set of keys is optimal, simply by comparing their costs. An appropriate traversal of the tree can be used to compute the costs in $O(n)$ time. Therefore, the problem of testing the optimality of a binary search tree is linear-time transformable to the problem of constructing an optimal binary search tree. Thus, a lower bound for the former is also a lower bound for the latter. Some conditions on the weights, for a given alphabetic tree, to be optimal have been obtained by Ramanan [137]. He shows that the optimality of very skewed trees (trees in which the number of nodes in any level is bounded by some constant) can be tested in linear time. He also shows that the optimality of well-balanced trees (trees in which the levels of any two leaves is bounded by some constant) can also be tested in linear time. He also considers a class of trees that is neither skewed nor well balanced and discusses the difficulty involved in testing its optimality in linear time.

4.7. *Other results*

The known heuristics for constructing nearly optimal binary search trees can be used for constructing nearly optimal alphabetic trees as the optimal alphabetic tree problem is a special case of the optimal binary search tree problem [110, 111, 129, 162]. These heuristics except Larmore's (Larmore's offers a tradeoff between speed and accuracy, as mentioned earlier, by spending $O(n^{1.6})$ time it produces a binary search tree whose weighted path length differs from the optimal by $o(1)$) produce trees which are within an additive factor of about two from the optimal; however, the small additive factor does not ensure a low multiplicative factor when the cost of the optimal alphabetic tree is very small. Levcopoulos et al. [102] show that for an arbitrarily small, positive real number ε, they can construct an $O(n)$ heuristic yielding an alphabetic tree whose cost is within a factor of $(1+\varepsilon)$ from the optimum.

5. Conclusions and directions for further research

We have looked at algorithms for optimal binary search trees, optimal alphabetic trees and Huffman trees. The best-known algorithm for the general optimal binary search tree problem is Knuth's $O(n^2)$ time algorithm. For the general optimal alphabetic

tree problem, the best-known algorithms are the Garsia–Wachs and the Hu–Tucker algorithms which have a time complexity of $O(n \log n)$. Huffman's $O(n \log n)$ algorithm to construct Huffman trees is optimal for the decision tree model. There are many aspects of optimal binary search trees and their variants for which we would like to obtain answers. We discuss some of the open problems in this area.

5.1. Optimal binary search trees

1. Is there any $o(n^2)$ time algorithm for the optimal binary search tree problem?
We have looked at Larmore's subquadratic time algorithm for constructing an optimal binary search tree, if there is no long sublist of low-frequency elements. It makes use of subtree weight ratios of the optimal binary search trees. It may be the case that we can use some other properties of optimal binary search trees to obtain a subquadratic time algorithm for the general case. It may also be possible to improve Knuth's quadratic time dynamic programming algorithm. We should also note the recent result of Karpinski et al. [74] in this direction.

2. Is there any $o(n^2)$ space algorithm for the optimal binary search tree problem?
We can find an optimal binary search tree using linear space and exponential time by generating all possible binary search trees [141]. If we can avoid generating many of these binary search trees by using properties of the optimal binary search trees like monotonicity, subtree weight ratios, we might be able to get a subexponential algorithm using $O(n)$ space. Perhaps we can get an $o(n^2)$ space algorithm by keeping only $o(n^2)$ entries of the $P_{i,j}$ we compute in the dynamic programming algorithm. We can recompute from scratch the $P_{i,j}$'s we don't remember. It will be worth exploring whether such a technique would result in an $o(n^2)$ space, polynomial-time algorithm.

3. Does there exist a subquadratic algorithm for verifying the optimality of a binary search tree?

We do know that the monotonicity of roots and the subtree weight ratio conditions are necessary for a given binary search tree to be optimal. These conditions can be verified in $O(n)$ time. It is worth exploring other necessary and sufficient conditions that enable us to verify in $o(n^2)$ time whether a given binary search tree is optimal. An answer to this question will enable us to know whether we can close the gap between constructing an optimal binary search tree and testing its optimality. We must also take a note of the recent result of Karpinski et al. [74] as mentioned before.

4. Do the algorithms and properties of optimal binary search trees extend to optimal multiway search trees?

5.2. Optimal alphabetic trees

1. Can we construct optimal alphabetic trees in $o(n \log n)$ time?

Existing $O(n \log n)$ algorithms construct first an lmcp tree. The lower bound result of Klawe and Mumey for lmcp trees says that any algorithm for constructing an lmcp tree will take $\Omega(n \log n)$ time. Hence, we must follow a different approach to obtain

an $o(n \log n)$ algorithm. It is also possible that there is an $\Omega(n \log n)$ lower bound for constructing optimal alphabetic trees.

2. Can we verify the optimality of alphabetic trees in linear time?

Ramanan [137] has given some necessary and sufficient conditions for the optimality of an alphabetic tree. These conditions are verifiable in $O(n)$ time for some special cases. Some of the properties of optimal alphabetic trees proved enroute to the proof of correctness of Garsia–Wachs algorithm for constructing optimal alphabetic trees may be noted along with Ramanan's conditions.

3. Does there exist algorithms similar to the Hu–Tucker or the Garsia–Wachs algorithms for the optimal ternary tree problem?

For the binary case we combine lmcp pairs of nodes, therefore we may attempt to combine triples of nodes. However, Hu [64] has observed that combining lmcp triples does not give an optimal ternary tree. It may be the case that the Garsia–Wachs algorithm may generalise, though both the algorithms are equivalent in the binary case.

5.3. Huffman trees

1. Is there an $o(n \log n)$ algorithm for constructing Huffman trees in a model where we are allowed to compute the floor's and ceiling's of numbers?

We have an $\Omega(n \log n)$ lower bound on the time required for constructing Huffman trees in the decision tree model. It is possible that we can construct Huffman trees in $o(n \log n)$ time in models where we are allowed to compute the floor's and ceilings of numbers. In fact, the $O(n)$ algorithms for the special case when the weights are within a factor of two do compute floor's and ceiling's.

2. Dynamic Huffman codes

Knuth [87] gives an $O(l)$ algorithm to increase or decrease the weight of a node at a level l of a Huffman tree by 1. This immediately gives an $O(lw)$ algorithm to increase or decrease the weight of a node at level l by w. To insert or delete a node of weight w, this gives an $O(Lw)$ algorithm where L is the maximum level of the tree. But his result uses the fact that the tree is produced by Huffman's algorithm. It is useful to see whether this result can be extended for any optimal extended binary tree. Perhaps the necessary and sufficient conditions we used for Huffman trees may be useful here.

The reader may consult the references for several other interesting problems related to binary search trees.

Acknowledgements

The author is thankful to all those who helped him in preparing this article, especially Dr Venkatesh Raman of the Institute of Mathematical Sciences (email id: vraman@imsc.ernet.in) for his expert guidance, valuable suggestions, constant encouragement and continuous motivation that made it all possible.

References

[1] J. Abrahams, Parallelized Huffman and Hu–Tucker searching, *IEEE Trans. Inform. Theory* **40** (1994).

[2] E.N. Adams, Another representation of binary tree traversal, *Inform. Processing Lett.* **2** (1973) 52–54.

[3] G.M. Adel'son-Vel'skii and E.M. Landis, An algorithm for the organization of information, *Sov. Math. Dokl.* **3** (1962) 1259–1263.

[4] R. Ahlswede and I. Wegener, *Search Problems* (Wiley, New York, 1987).

[5] M. Aigner, *Combinatorial Search* (Wiley–Teubner, New york, 1988).

[6] H. Akdag, Performances of an algorithm constructing a nearly optimal binary tree, *Acta Inform.* **20** (1983) 121–132.

[7] B. Allen, On binary search trees, Research Report CS-77-27, Department of Computer Science, University of Waterloo, Waterloo, September 1977.

[8] B. Allen, On the costs of optimal and near-optimal binary search trees, *Acta Inform.* **18** (1982) 255–263.

[9] B. Allen and I. Munro, Self-organising binary search trees, in: *Proc. 17th Ann. IEEE Symp. on Foundations of Computer Science* (1976) 166–172.

[10] B. Allen and I. Munro, Self-organizing binary search trees, *J. ACM* **25** (1978) 526–535.

[11] A. Andersson, C. Icking, R. Klein and T. Ottman, Binary search trees of almost optimal height, *Acta Inform.* **28** (1990) 165–178.

[12] A. Andersson, A note on searching in a binary search tree, *Software-Practice Experience* **21** (1991) 1125–1128.

[13] Ashok Subramanian, Design and analysis of algorithms, Tech. Report IISC-CSA-93-01 Dept. of Computer Science, Indian Institute of Science, Bangalore, April 1993.

[14] P.J. Bayer, Improved bounds on the cost of optimal and balanced binary search trees, M. Sc. Thesis, Massachusetts Institute of Technology, Cambridge, 1975.

[15] J. Bell and G. Gupta, An evaluation of self-adjusting binary search tree techniques, *Software-Practice Experience* **23** (1993) 369–382.

[16] J.L. Bentley, Multidimensional binary search trees used for associative searching, *Comm. ACM* **18** (1975) 509–517.

[17] A. Berztiss, A taxonomy of binary tree traversals, *BIT* (1987).

[18] C. Blundo and R. de Prisco, New bounds on the expected length of one-to-one codes, *IEEE Trans. Inform. Theory* **42** (1996) 246–249.

[19] W.H. Burge, An analysis of binary search trees formed from sequences of non-distinct keys, *J. ACM* **23** (1976) 451–454.

[20] R.M. Capocelli and A. de Santis, Improved bounds on the redundancy of Huffman codes, IBM Tech. Report, RC-14151, October 1988.

[21] R.M. Capocelli and A. de Santis, A note on d-ary Huffman codes, *IEEE Trans. Inform. Theory* **37** (1991) 174–179.

[22] R.M. Capocelli and A. de Santis, New bounds on the redundancy of Huffman codes, *IEEE Trans. Inform. Theory* **37** (1991) 1095–1104.

[23] R.M. Capocelli and A. de Santis, Variations on a theme by Gallager, in: J.A. Storer ed., *Image and Text Compression* (Kluwer, Dordrecht, 1992) 181–213.

[24] H. Chang and S. Sitharama Iyengar, Efficient algorithms to globally balance a binary search tree, *Comm. ACM* **27** (1984) 695–702.

[25] R.P. Cheetham, B.J. Oomen and D.T.H. Ng, On using conditional rotation operations to adaptively structure binary search trees, in: *Proc. 2nd Internat. Conf. on Database Theory*, Lecture Notes in Computer Science, Vol. 326 (Springer, Berlin, 1988) 161–175.

[26] G. Chen, M.S. Yu and L.T. Liu, Two algorithms for constructing a binary tree from its traversals, *Inform. Proc. Lett.* **28** (1988) 297–299.

[27] D. Cohen and M.L. Fredman, Weighted binary trees for concurrent searching, *J. Algorithms* **20** (1996) 87–112.

[28] J. Cooper and S.G. Akl, Efficient selection on a binary tree, *Inform. Processing Lett.* **23** (1986) 123–126.

[29] D. Coppersmith, M.M. Klawe and N.J. Pippenger, Alphabetic minimax trees of degree at most t, *SIAM J. Comput.* **15** (1986) 189–192.

[30] J. Culberson, The effect of updates in binary search trees, in: *Proc. 17th Ann. ACM Symp. on Theory of Computing* (1985) 6–8.
[31] J.C. Culberson and J.I. Munro, Analysis of the standard deletion algorithms in exact fit domain binary search trees, *Algorithmica* **5** (1990) 295–311.
[32] W. Cunto and J.L. Gascon, Improving time and space efficiency in generalised binary search trees, *Acta Inform.* **24** (1987) 583–594.
[33] A.C. Day, Balancing a binary tree, *Comput. J.* **19** (1976).
[34] L. Devroye, A note on the height of binary search trees, *J. ACM* **33** (1986) 489–498.
[35] L. Devroye and J.M. Robson, On the generation of random binary search trees, *SIAM J. Comput.* **24** (1995) 1141–1156.
[36] L. Devroye and B. Reed, On the variance of the height of random binary search trees, *SIAM J. Comput.* **24** (1995) 1157–1162.
[37] M.C. Er, A new algorithm for generating binary trees using rotations, *Comput. J.* **32** (1989) 470–473.
[38] Fenner and Loizou, A study of binary tree traversal algorithms and a tag-free threaded representation, *Internat. J. Comput. Math.* **20** (1986).
[39] Filho, Optimal choice of discriminators in a balanced k–d binary search tree, *Inform. Processing Lett.* **13** (1981).
[40] A.S. Fraenkel and S.T. Klein, Bounding the depth of search trees, *Comput. J.* **36** (1993) 668–678.
[41] M.R. Garey, Optimal binary identification procedures, *SIAM J. Appl. Math.* **23** (1972) 173–186.
[42] M.R. Garey, Optimal binary search trees with restricted maximal depth, *SIAM J. Comput.* **3** (1974) 101–110.
[43] A.M. Garsia and M.L. Wachs, A new algorithm for minimum cost binary trees, *SIAM J. Comput.* **6** (1977) 622–642.
[44] N. Gabrani and P. Shankar, A note on the reconstruction of a binary tree from its traversals, *Inform. Processing Lett.* **42** (1992) 117–119.
[45] R.G. Gallager, Variations on a theme of Huffman, *IEEE Trans. Inform. Theory* **24** (1978) 668–674.
[46] T.E. Gerasch, An insertion algorithm for a minimal internal path length binary search tree, *Comm. ACM* **31** (1988) 579–585.
[47] E.N. Gilbert, Codes based on inaccurate source probabilities, *IEEE Trans. Inform. Theory* **17** (1971) 304–314.
[48] E.N. Gilbert and E.F. Moore, Variable length binary encodings, *Bell System Tech. J.* **38** (1959) 933–968.
[49] J. Glenn, Binary trees, Tech. Report DCS-TR86-127, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1986.
[50] S.W. Golomb, Sources which maximize the choice of a Huffman coding tree, *Inform. and Control* **45** (1980) 263–272.
[51] D. Gries and J.L.A. van de Snepscheut, Inorder traversal of a binary tree and its inversion, in: E.W. Dijkstra, ed., *Formal Development of Programs and Proofs* (Addison-Wesley, Reading, MA, 1990).
[52] L.J. Guibas, A principle of independence for binary tree searching, *Acta Inform.* **4** (1974) 293–298.
[53] R. Guttler, K. Mehlhorn and W. Scneider, Binary search trees: average and worst case behaviour, *Elektron. Informat. Kybernet.* **16** (1980) 41–61.
[54] E.N. Hanson, The interval skip list: a data structure for finding all intervals that overlap a point, Tech. Report WSU-CS-91-01, Washington State University, 1991.
[55] E.N. Hanson and M. Chaabouni, The IBS-tree: A data structure for finding all intervals that overlap a point, Tech. Report WSU-CS-90-11, Washington State University, 1990.
[56] F. Harary, E.M. Palmer and R.W. Robinson, Counting free binary trees admitting a given height, Tech. Report UGA-CS-TR-90-001, University of Georgia, 1990.
[57] J.H. Hester and D.S. Hirschberg, Generation of optimal binary split trees, Tech. Report UCI/ICS-TR-85-13, Department of Information and Computer Science, University of California, Irvine, March 1985.
[58] T.N. Hibbard, Some combinatorial properties of certain trees with applications to sorting and searching, *J. ACM* **9** (1962) 13–28.
[59] T. Hikita, Listing and counting subtrees of equal size of a binary tree, *Inform. Processing Lett.* **17** (1983) 225–229.

[60] D.S. Hirschberg, L.L. Larmore and M. Moldowitch, Subtree weight ratios for optimal binary search trees, Tech. Report 86-02 ICS Dept. Univ. of Calif. Irvine, 1986.
[61] Y. Horibe, An improved bound for weight balanced trees, *Inform. and Control* **34** (1977) 148–151.
[62] Y. Horibe and T.O.H. Nemetz, On the max-entropy rule for a binary search tree, *Acta Inform.* **12** (1979) 63–72.
[63] T.C. Hu, A new proof of the T–C algorithm, *SIAM J. Appl. Math.* **25** (1973) 83–94.
[64] T.C. Hu, *Combinatorial Algorithms* (Addison-Wesley, Reading, MA, 1982).
[65] T.C. Hu, D.J. Kleitman and J.K. Tamaki, Binary trees optimum under various criteria, *SIAM J. Appl. Math.* **37** (1979) 246–256.
[66] T.C. Hu and K.C. Tan, Least upper bound on the cost of optimal binary search trees, *Acta Inform.* **1** (1972) 307–310.
[67] T.C. Hu and K.C. Tan, Path lengths of binary search trees, *SIAM J. Appl. Math.* **22** (1972) 225–234.
[68] T.C. Hu and A.C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Appl. Math.* **21** (1971) 514–532.
[69] D.A. Huffman, A method for the construction of minimum redundancy codes, *Proc. IRE* **40** (1952) 1098–1101.
[70] A. Itai, Optimal alphabetic trees, *SIAM J. Comput.* **5** (1976) 9–18.
[71] G. Jacobson, Succinct static data structures, Tech. Report CMU-CS-89-112 Dept. of Computer Science Carnegie-Mellon University, January 1989.
[72] V. Kamakoti and C.P. Rangan, An optimal algorithm for reconstructing a binary tree, *Inform. Processing Lett.* **42** (1992) 113–115.
[73] P.C. Karlton, S.H. Fuller, R.E. Scroggs and E.B. Kaehler, Performance of height-balanced trees, *Comm. ACM* **19** (1976)
[74] M. Karpinski, L.L. Larmore and W. Rytter, Sequential and parallel subquadratic work algorithms for constructing approximately optimal binary search trees, in: *Proc. 7th Ann. ACM-SIAM Symp. on Discrete Algorithms* (1996) 36–41.
[75] G.O.H. Katona and T.O.H. Nemetz, Huffman codes and self-information, *IEEE Trans. Inform. Theory* **22** (1976) 337–340.
[76] R. Kemp, Binary search trees constructed from nondistinct keys with/without specified probabilities, *Theoret. Comput. Sci.* **156** (1989) 181–203.
[77] A.C. Kilgour, Generalized nonrecursive traversal of binary trees, *Software-Practice Experience* **11** (1981) 1299–1306.
[78] J.H. Kingston, A new proof of the Garsia-Wachs algorithm, *J. Algorithms* **9** (1988) 129–136.
[79] D.G. Kirkpatrick and M. Klawe, Alphabetic minimax trees, *SIAM J. Comput.* **14** (1985) 514–526.
[80] M. Klawe and B. Mumey, Upper and lower bounds on constructing alphabetic binary trees, in: *Proc. 4th Ann. ACM-SIAM Symp. on Discrete Algorithms* (1993) 185–193.
[81] R. Klein and D. Wood, On the path length of binary trees, *J. ACM* **36** (1989) 280–289.
[82] G.D. Knott, A balanced tree storage and retrieval algorithm, in: *Proc. ACM Symp. on Inform. Storage and Retrieval* (1971).
[83] G.D. Knott, A numbering system for binary trees, *Comm. ACM* **20** (1977) 113–115.
[84] D.E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms* (Addison-Wesley, Reading, MA, 1968).
[85] D.E. Knuth, Optimum binary search trees, *Acta Inform.* **1** (1971) 14–25.
[86] D.E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).
[87] D.E. Knuth, Dynamic Huffman coding, *J. Algorithms* **6** (1985) 163–180.
[88] A.P. Korah and M.R. Kaimal, Dynamic optimal binary search tree, *Internat. J. Found. Comput. Sci.* **1** (1990) 449–463.
[89] C.H.A. Koster and Th.P. van der Weide, Hairy search trees, *Comput. J.* **38** (1995) 691–694.
[90] D.C. Kozen, *Design and analysis of algorithms* (Springer, New York, 1992).
[91] H.T. Kung and P.L. Lehman, Concurrent manipulation of binary search trees, *ACM Trans. Database Systems* **5** (1980) 354–382.
[92] T.W. Lai and D. Wood, Adaptive heuristics for binary search trees and constant linkage cost, in: *Proc. 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms* (1991) 28–30.
[93] C.E. Langenhop and W.E. Wright, Probabilities related to father-son distances in binary search trees, *SIAM J. Comput.* **15** (1986) 520–530.

[94] L.L. Larmore, A subquadratic algorithm for constructing approximately optimal binary search trees, Tech. Report UCI//ICS-TR-86-03, Department of Information and Computer Science, University of California, Irvine, February 1986.
[95] L.L. Larmore, A subquadratic algorithm for constructing approximately optimal binary search trees, *J. Algorithms* **8** (1987) 579–591.
[96] L.L. Larmore, Length limited coding and optimal height-limited binary trees, Tech. Report UCI//ICS-TR-88-01, Dept. of Information and Computer Science, University of California, Irvine, March 1989.
[97] L.L. Larmore and D.S. Hirschberg, A fast algorithm for optimal length-limited Huffman codes, *J. ACM* **37** (1990) 464–473.
[98] L.L. Larmore and T.M. Przytycka, The optimal alphabetic tree problem revisited, in: *Proc. Internat. Coll. on Automata Languages and Programming, ICALP 94* (1994).
[99] L.L. Larmore and T.M. Przytycka, Constructing Huffman trees in parallel, *SIAM J. Comput.* **24** (1995) 1163–1169.
[100] L.L. Larmore and T.M. Przytycka, A parallel algorithm for optimum height-limited alphabetic binary trees, *J. Parallel Distri. Comput.* **35** (1996) 49–56.
[101] D.T. Lee and C.K. Wong, Worst case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees, *Acta Inform.* **9** (1977) 23–29.
[102] C. Levcopoulos, A. Lingas and J. R. Sack, Heuristics for optimum binary search trees and minimum weight triangulation problems, *Theoret. Comput. Sci.* **66** (1989) 181–203.
[103] J.M. Lucas, D.R. van Baronaigien and F. Ruskey, On rotations and the generation of binary trees, *J. Algorithms* **15** (1993) 343–366.
[104] E. Makinen, Left distance binary tree representations, *BIT* **27** (1987) 163–169.
[105] E. Makinen, Constructing a binary tree from its traversals, *BIT* **29** (1989) 572–578.
[106] E. Makinen, A linear time and space algorithm for finding isomorphic subtrees of a binary tree, *BIT* **31** (1991).
[107] E. Makinen, A note on Gupta's binary tree codings, *Bull. EATCS* **49** (1993).
[108] G. Markowsky, Best Huffman codes, *Acta Inform.* **16** (1981) 363–370.
[109] H.W. Martin and B.J. Or, A random binary tree generator, in: *Proc. 17th Ann. ACM Computer Science Conf.*, Louisville, KY (1989) 37-38.
[110] K. Mehlhorn, Nearly optimal binary search trees, *Acta Inform.* **5** (1975) 287–295.
[111] K. Mehlhorn, A best possible bound for the weighted path length of binary search trees, *SIAM J. Comput.* **6** (1977) 235–239.
[112] K. Mehlhorn, Dynamic binary search, *SIAM J. Comput.* **8** (1979) 175–198.
[113] K. Mehlhorn, *Data structures and algorithms* Vol. 1: *Sorting and Searching*, EATCS Monographs on Theoretical Computer Science (Springer, Berlin, 1984).
[114] K. Mehlhorn and A. Tsakalidis, Data Structures, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Vol. A (Elsevier, Amsterdam, 1990) 303–341.
[115] A. Moitra and S. Sitharama Iyengar, Derivation of a maximally parallel algorithm for balancing binary search trees, Tech. Report TR 84-638, Dept. of Computer Science Cornell University, Ithaca, New York, September 1984.
[116] B.M. Mumey, Some new results for constructing optimal alphabetic binary trees, M. Sc. Thesis, Univ. of British Columbia, Canada, 1992.
[117] J.I. Munro and P.V. Poblete, A discipline for robustness or storage reduction in binary search trees, in: *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1983) 21–23.
[118] J.I. Munro and P.V. Poblete, Fault tolerance and storage reduction in binary search trees, *Inform. and Control* **62** (1984) 210–218.
[119] J. Nievergelt, Binary search trees and file organisation, *ACM Comput. Surveys* **6** (1974) 195–207.
[120] J. Nievergelt and E.M. Riengold, Binary search trees of bounded balance, in: *Proc. 4th Ann. ACM Symp. on Theory of Computing* (1972) 1–3.
[121] J. Nievergelt and E.M. Riengold, Binary search trees of bounded balance, *SIAM J. Comput.* **21** (1973) 33–43.
[122] J. Nievergelt and C.K. Wong, Upper bounds for the total path length of binary trees, *J. ACM* **20** (1973) 1–6.
[123] O. Nurmi and E. Soisalon-Soininen, Uncoupling updating and rebalancing in chromatic binary search trees, in: *Proc. 10th ACM SIGACT-SIGART Symp. on Principles of Database Systems* (1991) 29–31.

[124] S. Olariu, C. Overstreet and Z. Wen, An optimal parallel algorithm to reconstruct a binary tree from its traversals, in: *Proc. Advances in Computing and Information–ICCI: Internat. Conf. on Computing and Information*, Lecture Notes in Computer Science, Vol. 497 (Springer, New York, 1991) 484–495
[125] S. Olariu, C. Overstreet and Z. Wen, Reconstructing a binary tree from its traversals in doubly logarithmic CREW time, *J. Parallel Distri. Comput.* **27** (1995).
[126] Ottmann and Wood, How to update a balanced binary tree with a constant number of rotations, in: *Proc. Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science (Springer, New York, 1990).
[127] J.M. Pallo, Enumerating ranking and unranking binary trees, *Comput. J.* **29** (1986).
[128] P. V. Poblete and J. I. Munro, The analysis of a fringe heuristic for binary search trees, *J. Algorithms* **6** (1985) 336–350.
[129] R. de Prisco and A. de Santis, On binary search trees, *Inform. Proc. Lett.* **45** (1993) 249–253.
[130] R. de Prisco and A. de Santis, New lower bounds on the cost of binary search trees, *Theoret. Comput. Sci.* **156** (1996) 315–325.
[131] R. de Prisco and A. de Santis, On the redundancy achieved by Huffman codes, *Inform. Sci.* **88** (1996) 131–148.
[132] R. de Prisco, G. Parlati and G. Persiano, Minimal path length of binary trees, *Theoret. Comput. Sci* **143** (1995) 175–188.
[133] R. de Prisco and G. Persiano, Characteristic inequalities for binary trees, *Inform. Processing Lett.* **53** (1995) 201–207.
[134] A. Proskurowski, On the generation of binary trees, *J. ACM* **27** (1980) 1–2.
[135] A. Proskurowski and F. Ruskey, Binary tree gray codes, *J. Algorithms* **6** (1985) 225–238.
[136] K.J. Raeihae and S.H. Zweben, An optimal insertion algorithm for one-sided height-balanced binary search trees, *Comm. ACM* **22** (1979) 508–512.
[137] P. Ramanan, Testing the optimality of alphabetic trees, *Theoret. Comput. Sci.* **93** (1992) 279–301.
[138] Ramarao, O(log N) parallel algorithms for binary tree traversals, in: *Proc. 22th Ann. Allerton Conf. on Commun. Control and Comput.* (Allerton House, Monticello, IL, 1984).
[139] J. Rissanen, Bounds for weighted balanced trees, *IBM J. Res. Develop.* (1973) 101–105.
[140] F. Ruskey, Generating t-ary trees lexicographically, *SIAM J. Comput.* **7** (1978) 424–439.
[141] F. Ruskey and T.C. Hu, Generating binary trees lexicographically, *SIAM J. Comput.* **6** (1977) 745–758.
[142] A. de Santis and G. Persiano, Tight upper and lower bounds on the path length of binary trees, *SIAM J. Comput.* **23** (1994) 12–23.
[143] E.S. Schwartz, An optimum encoding with minimum longest code and total number of digits, *Inform. and Control* **7** (1964) 37–44.
[144] E.S. Schwartz and B. Kallick, Generating a canonical prefix encoding, *Comm. ACM* **7** (1964) 166–169.
[145] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees, *J. ACM* **32** (1985) 652–686.
[146] M. Solomon and R.A. Finkel, A note on enumerating binary trees, *J. ACM* **27** (1980) 3–5.
[147] D. Spuler, The optimal binary search tree for Andersson's search algorithm, *Acta Inform.* **30** (1993) 405–407.
[148] A. E. Trojanowski, Ranking and listing algorithms for k-ary trees, *SIAM J. Comput.* **7** (1978) 492–509.
[149] Troutman and Karlinger, A note on subtrees rooted along the primary path of a binary tree, *Discrete Appl. Math. Comb. Oper. Res. Comput. Sci.* **42** (1993).
[150] K. Unterauer, Dynamic weighted binary search trees, *Acta Inform.* **11** (1979) 341–362.
[151] D.R. van Baronaigien, A loopless algorithm for generating binary tree sequences, *Inform. Processing Lett.* **39** (1991) 189–194.
[152] P. van Emde Boas, An $O(n \log \log n)$ on-line algorithm for the insert-extract min problem, Tech. Report TR 74-221, Dept. of Computer Science Cornell University, December 1974.
[153] J.G. Vaucher, Pretty-printing of trees, *Software–Practice Experience* **10** (1980) 553–561.
[154] J. Vuillemin, A data structure for manipulating priority queues, *Comm. ACM* **21** (1978) 309–315.
[155] W.A. Walker and C.C. Gotlieb, A top-down algorithm for constructing nearly optimal lexicographic trees, in: *Graph Theory and Computing* (Academic Press, New York, 1972) 303–323.
[156] R.L. Wessner, Optimal alphabetic search trees with restricted maximal height, *Inform. Processing Lett.* **4** (1976) 90–94.

[157] R. Wilber, Lower bound for accessing binary search trees with rotations, in: *Proc. 27th Ann. Sympo. Found. Comput. Sci.* (1986) 27–29.
[158] R. Wilber, Lower bounds for accessing binary search trees with rotations, *SIAM J. Comput* **18** (1989) 56–57.
[159] W.E. Wright, Binary search trees in secondary memory, *Acta Inform.* **15** (1981) 3–17.
[160] F.F. Yao, Efficient dynamic programming using quadrangle inequalities, in: *Proc. 12th Ann. ACM Sympo. on Theory of Computing* (1980) 429–435.
[161] F.F. Yao, Speed-up in dynamic programming, *SIAM J. Alg. Discrete Math.* **3** (1982) 532–540.
[162] R.W. Yeung, Alphabetic codes revisited, *IEEE Trans. Inform. Theory* **37** (1991) 564–572.
[163] R.W. Yeung, Local redundancy and progressive bounds on the expected length of a Huffman code, *IEEE Trans. Inform. Theory* **37** (1991) 687–691.
[164] J.M. Yohe, Hu–Tucker minimum redundancy alphabetic coding method, *Comm. ACM* **15** (1972) 360–362.