3rd ICALP
1976

J. van Leeuwen

ON THE CONSTRUCTION OF HUFFMAN TREES

ABSTRACT. We develop a feasible linear time algorithm to construct optimal alphabetic binary Huffman trees for ordered sets of records. For arbitrary arrangements very good approximations can be found in linear time also. It leads to a new algorithm for constructing ordinary Huffman trees and we argue that its complexity is essentially optimal. We introduce weakly stable alphabetic trees which are easier to maintain than optimal trees when access-frequencies are modified or changed at run-time. Using the appropriate data structures a weakly stable tree on n records can be constructed in nlogn steps.

INTRODUCTION

Suppose n records $r_1,\dots,r_n$ with probabilities $w_1,\dots,w_n$ are arranged at the leaves of a binary search-tree, and let l_i be the path-length needed for accessing r_i. Any tree achieving a reduced and possibly smallest value of $\sum_{i=1}^{n} w_i l_i$ (the average search time for the file) will be informally called a Huffman tree. Huffman trees were originally conceived in coding theory for finding minimum redundancy codes (Huffman 1952). Huffman gave the first algorithm for constructing an optimal tree based on a "bottom-up" combination of subtrees. (See also Karp 1961). An "unusual" application of Huffman trees is found in determining the order of summation for a finite set of numeric terms to get a minimal worst-case rounding error.

Our present interest relates to optimal file or-

Keywords and Phrases: Huffman trees, sorting, Hu-Tucker algorithm, weakly stable trees, 2-3 trees
CR category : 5.25

ganization and aims at a further study of the combinatorial nature of Huffman's algorithm and similar schemes. Knuth (1968, p. 404) gives a few properties, but little is mentioned about computational aspects except that there is an $O(n \log n)$ algorithm. We shall prove that $O(n \log n)$ is optimal for a wide class of algorithms.

Schwartz & Kallick (1964; see Knuth 1968, p. 589) observed that if $w_1 \leq \ldots \leq w_n$ then an optimal Huffman tree exists with $r_1, \ldots, r_n$ at the leaves in consecutive order. It is known as an alphabetic optimal Huffman tree and the paper of Schwartz & Kallick suggests an $O(n^2)$ algorithm. We shall prove that there is a feasible linear time algorithm to construct it, and the algorithm will produce such a tree with smallest l_{max}. Our method depends on the particular nature of Huffman's algorithm in this case and consists of subtree-rearrangements in a doubly-linked ("chained") tree which require little time. If links are retained in the final tree we automatically have the kind of structure sometimes advocated for flexible files (see e.g. Sussenguth 1963, Patt 1969, and Hu 1972).

In the case of an arbitrary ordering of records it is difficult to understand the precise nature of optimal alphabetic trees. Knuth (1972, p. 433-439) gives an $O(n^2)$-time and -space algorithm based on a dynamic programming principle (see also Gilbert & Moore 1959). The celebrated Hu-Tucker algorithm is hard to prove (Hu & Tucker 1971, see also Knuth 1972 p. 439-445) despite further simplifications of the argument (Hu 1973), but it gives the desired tree in a most interesting manner in $O(n^2)$ time and only linear space. (The algorithm was programmed by Yohe 1972, see also Byrne 1973). Knuth (1972, p. 444) indicated an $O(n \log n)$-implementation.

We shall prove that a very good approximation to an optimal alphabetic tree can be found in only linear time.

Various authors observed that alphabetic trees are hard to keep optimal when the weight of records is updated by run-time information or when records have to be inserted in alphabetic order or deleted from the file (see e.g. Bruno & Coffman 1972, Walker & Gotlieb 1972, and Nievergelt & Reingold 1972). In this paper we therefore introduce weakly stable trees which merely achieve an optimal local weight-balance somewhat in the spirit of ideas of Walker &

Gotlieb (1972) and Mehlhorn (1975), but with a more rigid local stability criterion enforced throughout the tree. Weakly stable trees are always very close to optimal.

Using a data structure which allows for efficient SPLITS we will show that there is an algorithm to construct a weakly stable alphabetic tree in $O(n \log n)$ steps. The restoration of weakly stable trees upon insertion or deletion of a record shows an interesting downward propagation of local re-balancing. We note that Fredman (1975) recently developed a linear time algorithm for another kind of weight-balanced search-trees.

ALPHABETIC HUFFMAN - TREES IN THE ORDERED CASE

The ordinary algorithm for optimal Huffman trees (Huffman 1952, Zimmerman 1959, Schwartz 1964) consists of a repeated "construct-a-subtree and insert-in-order" cycle and requires $O(n \log n)$ time. We shall prove

Theorem 2.1

If records are initially given in order of increasing probability then there is a "stable" linear time algorithm to construct an optimal alphabetic tree.

Let $w_1 \leq \ldots\ldots \leq w_n$ and consider the ordinary Huffman - algorithm. We shall first show that instead of a heap now only a queue is needed to hold the intermediate information about subtree-ordering. Assume there is an input-queue IN containing elements of

```
type item = record
                w : weight;
                π : pointer to record r;
            end;
```

and an auxiliary queue Q for elements of

```
type node = record
                w : accumulated weight;
             left : pointer to node;
            right : pointer to node;
            end;
```

Consider the linked structure produced by

```
Algorithm A
    "initialize"
        attach $(w_1,\pi_1),\ldots,(w_n,\pi_n)$ to IN;
```

```
      attach Q;
   "cycle"
      repeat
         determine $(\alpha,\beta)\varepsilon\{(IN\uparrow,IN\uparrow\uparrow),(IN\uparrow,Q\uparrow),$
               $(Q\uparrow,Q\uparrow\uparrow)\}$
         such that $\alpha\cdot w + \beta\cdot w$ is minimal;
         new $(\gamma)$;
         $\gamma\cdot w$      := $\alpha\cdot w + \beta\cdot w$;
         $\gamma\cdot$left   := $\alpha$ or $\beta$;
         $\gamma\cdot$right  := $\beta$ or $\alpha$;
         delete $\alpha$ and $\beta$ from their source-queues;
         attach $\gamma$ to Q
      until IN is empty $\wedge$ #Q=1;
   "output"
      output $Q\uparrow$;
```

IN remains ordered throughout the algorithm. We shall prove that it holds for Q also.

Lemma 2.2

At any moment in algorithm A we have $Q\uparrow\cdot w \leq Q\uparrow\uparrow\cdot w \leq$ .... (up to the last record in the queue).

Proof. It holds at the beginning and after the first step. Continue by induction. Suppose it holds at steps k-1 and k. Consider how step k is achieved.

Case I:
at step k-1 we have

$IN = (w_i,*)(w_{i+1},*)(w_{i+2},*)(w_{i+3},*)....$

$Q = (q_1,*)(q_2,*)....(q_j,*)$

and at step k we have

$IN = (w_{i+2},*)(w_{i+3},*)....$

$Q = (q_1,*)(q_2,*)....(q_j,*)(w_i+w_{i+1},*)$

where

$w_i+w_{i+1} \leq w_i+q_1$ and $w_i+w_{i+1} \leq q_1+q_2$.

In step k+1 we determine the minimum of $\{w_{i+2}+w_{i+3},w_{i+2}+q_1,q_1+q_2\}$ and attach a corresponding record to Q after the record for w_i+w_{i+2}. Since $w_i+w_{i+1} \leq w_{i+2}+w_{i+3}, w_i+w_{i+1} \leq w_i+q_i \leq w_{i+2}+q_1$, and $w_i+w_{i+1} \leq q_1+q_2$ Q remains ordered no matter which value is smallest.

Case II:
at step k-1 we have

$IN = (w_i,*)(w_{i+1},*)(w_{i+2},*)....$

$Q = (q_1,*)(q_2,*)\dots(q_j,*)$

and at step k we have

$IN = (w_{i+1},*)(w_{i+2},*)\dots$

$Q = (q_2,*)\dots(q_j,*)(w_i+q_1,*)$

where

$$w_i+q_1 \le w_i+w_{i+1} \text{ and } w_i+q_1 \le q_1+q_2$$

In step k+1 we determine the minimum of $\{w_{i+1}+w_{i+2}, w_{i+1}+q_2, q_2+q_3\}$ and attach a corresponding record to Q after the record for w_i+q_1. Since $w_i+q_1 \le w_{i+1}+w_{i+1} \le w_{i+1}+w_{i+2}$, $w_i+q_1 \le w_{i+1}+q_1 \le w_{i+1}+q_2$, and $w_i+q_1 \le q_1+q_2 \le q_2+q_3$ Q remains ordered no matter which value is smallest.

Case III:

at step k-1 we have

$IN = (w_i,*)(w_{i+1},*)\dots$

$Q = (q_1,*)(q_2,*)(q_3,*)\dots(q_j,*)$

and at step k we have

$IN = (w_i,*)(w_{i+1},*)\dots$

$Q = (q_3,*)(q_4,*)\dots(q_j,*)(q_1+q_2,*)$

where

$$q_1+q_2 \le w_i+w_{i+1} \text{ and } q_1+q_2 \le w_i+q_i$$

In step k+1 we determine the minimum of $\{w_i+w_{i+1}, w_i+q_3, q_3+q_4\}$ and attach a corresponding record to Q after the record for q_1+q_2. Since $q_1+q_2 \le w_i+w_{i+1}$, $q_1+q_2 \le w_i+q_1 \le w_i+q_3$, and $q_1+q_2 \le q_3+q_4$ Q remains ordered no matter which value is smallest.

In all three cases the argument is easily modified when IN or Q have only 3 records or less. ■

It is now fairly straightforward to prove that algorithm A is a stepwise implementation of Huffman's construction, but the heap normally needed remains permanently decomposed into two queues. (Lemma 2.2 is related to an observation of Hu & Tucker 1971; see also Knuth 1972, p. 450). It follows

Lemma 2.3.

If records are initially given in order of increasing probability, then algorithm A yields an optimal Huffman tree in linear time.

If we always choose the record with minimal weight occurring first in $\{(IN\uparrow,IN\uparrow\uparrow),(IN\uparrow,Q\uparrow),(Q\uparrow,Q\uparrow\uparrow)\}$, then algorithm A yields an optimal Huffman tree with smallest l_{max} (Schwartz 1964).

It appears that $r_1,r_2,\ldots,r_n$ may still get mixed up pretty badly in algorithm A. We shall develop a more precise version in which much of the freedom we had in combining subtrees is eliminated to keep the amount of disorder minimal.

Algorithm B will build a two-way (and later doubly linked) tree using elements of

```
type node = record
                 w : accumulated weight;
            father : pointer to node;
              left : pointer to node;
             right : pointer to node;
              link : pointer to node;
               end;
```

in the following way

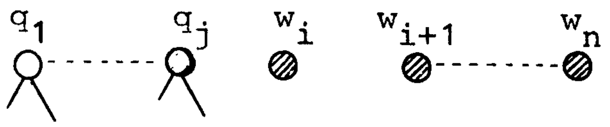
```
Algorithm B
    "initialize"
        for i to n do $\pi_i$ := $\uparrow(w_i,-,r_i,$nil$,-)$;
        attach $\pi_1 \ldots \pi_n$ to IN;
        attach Q;
    "cycle"
        repeat
            determine $(\alpha,\beta)\in\{(IN\uparrow,IN\uparrow\uparrow),(IN\uparrow,Q\uparrow),$
                  $(Q\uparrow,Q\uparrow\uparrow)\}$
            such that $\alpha\cdot w+\beta\cdot w$ is minimal;
            new $(\gamma)$;
            $\gamma$ := $\uparrow(\alpha\cdot w+\beta\cdot w,-,\alpha,\beta,-)$;
            $\alpha$.father := $\beta$.father := $\gamma$;
            delete $\alpha$ and $\beta$ from their source-queues;
            attach $\gamma$ to Q;
        until IN is empty and #Q=1;
    "output"
        output $Q\uparrow$;
```

The tree may still not be alphabetic but it now appears to have a hidden regularity.

Lemma 2.4.

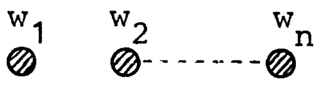
If records are initially given in order of increasing probability, then algorithm B yields an optimal Huffman tree in which at each level the occurring leaves are ordered.

Proof. By induction on the forest obtained by juxtaposing the subtrees in Q and IN (in that order)

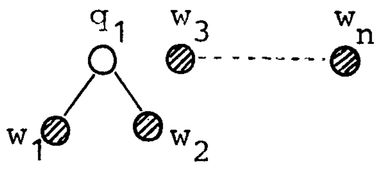


as the algorithm progresses. We shall prove the somewhat stronger <u>enumeration-property</u>: when leaves are enumerated per level from left to right, beginning at the lowest level and going upwards in the forest, then one precisely gets $r_1,\dots,r_n$ in their original order.

It trivially holds at the beginning

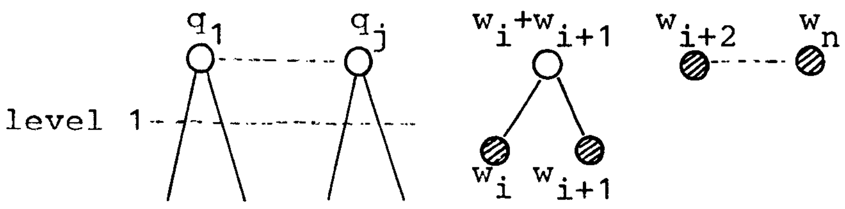


and after the first step



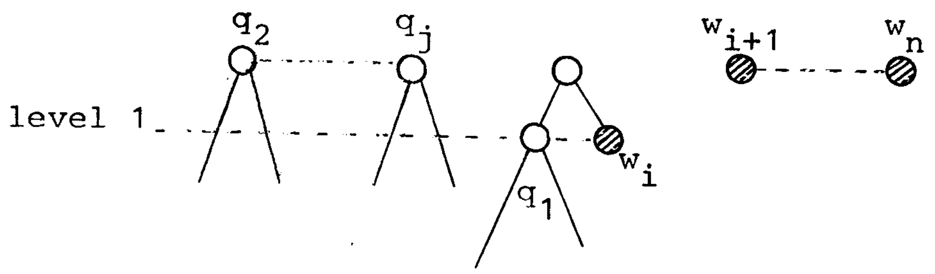
Assume the hypothesis at step k, and consider what can happen at step k+1.

<u>Case</u> I: if $\alpha \approx w_i$ and $\beta \approx w_{i+1}$ then we get



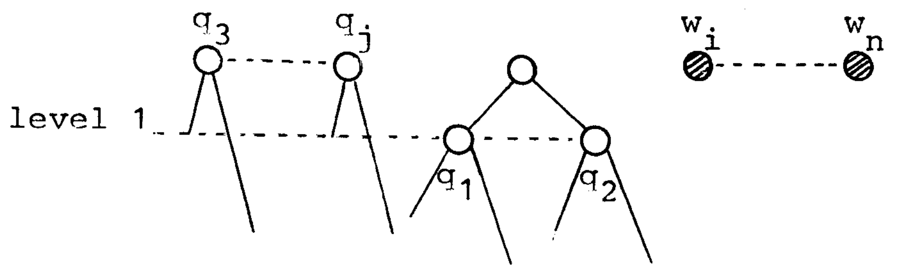
The leaves of the q-trees all occur before w_i in the original ordering and the enumeration-property is preserved.

<u>Case</u> II: if $\alpha \approx w_i$ and $\beta \approx q_1$ then we get



Instead of going back to the beginning of a next level we already find the leaves of q_1 listed immediately following the previous level now, and the order of enumeration hasn't changed.

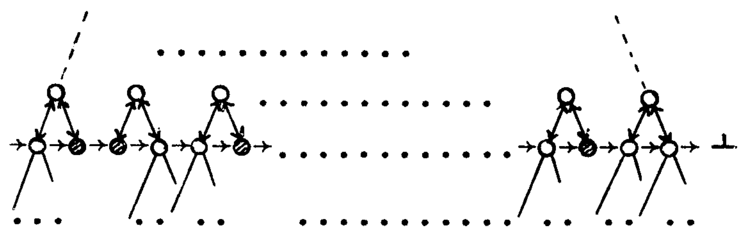
Case III: if $\alpha \approx q_1$ and $\beta \approx q_2$ then we get



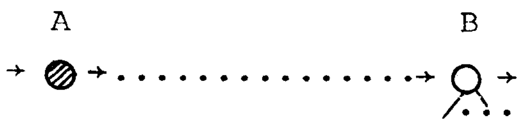
The leaves of q_1 and q_2 remain in order, but we have now attached them at the end one level lower. The enumeration properly remains therefore true. ■

From the proof is clear why the arrangement of subtrees was chosen the way it was in algorithm B. The enumeration-property will be used later to show that the method is stable : equal-weight records remain in order.

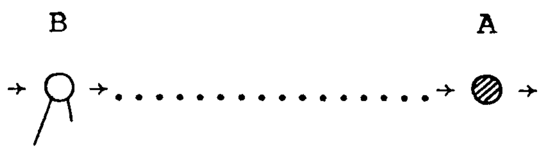
Examine the "highest" level in the tree where leaves occur, and assume that all nodes in the level are linked from left to right into a linear list



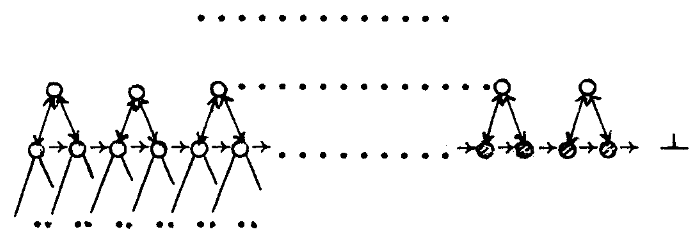
Observe that a transformation of



into



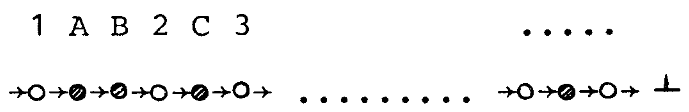
does not affect the weight of the tree. If we always perform the exchange operation on adjacent nodes (beginning with the rightmost ⊘-node) we can let all ⊘-nodes "bubble" to the far right end of the list and obtain a re-structured level of the form



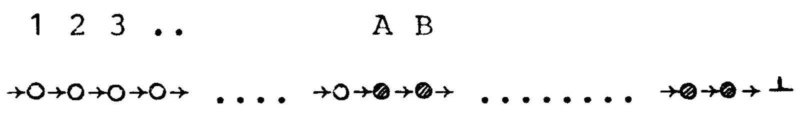
without changing the enumeration property and without changing the total weight of the tree.

If we continue the same procedure at the next lower level, and the next, and so on, then we bring the leaves up order more and more without loosing the enumeration property and optimality, obtaining in the end the alphabetic ordering we looked for! Thus we have a constructive proof (as opposed to Knuth 1968, p. 589) of Schwartz's result that for $w_1 \leq w_2 \leq$ .... there is an optimal Huffman tree which is alphabetic.

We still have to show how to perform the transformation procedure quickly. It is clear that we need an "demerging" algorithm for transforming linear lists



into

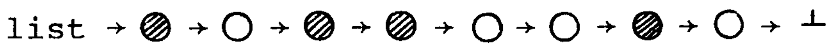


Lemma 2.5

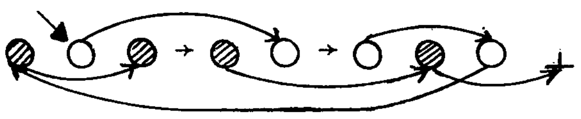
Demerging of two-colored linear lists can be performed in linear time and no extra space.
Proof. Assume the list contains elements of

```
type node = record
              info : (black,white);
              link : pointer to node;
               end
```

The procedure described below will transform a list like



into



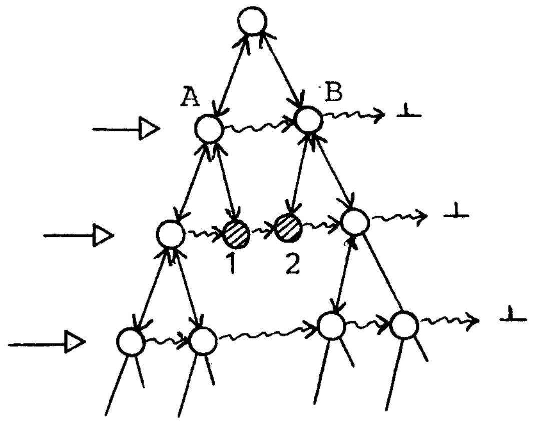
Let "list" be a pointer to the beginning of the list.

```
Algorithm C
   w :=   ( * ,nil);
   b :=   ( * ,nil);
   wlist := w;
   blist := b;
   next  := list;
   repeat
      if next·info = white then wlist·link :=
           next; wlist := next
                        else blist·link := next;
            blist := next fi;
      next := next·link
   until next is nil;
   wlist·link := b·link;
   list := w·link;
   output list;
```

The algorithm provides a stable "demerging" in time proportional to the number of list-elements. ■

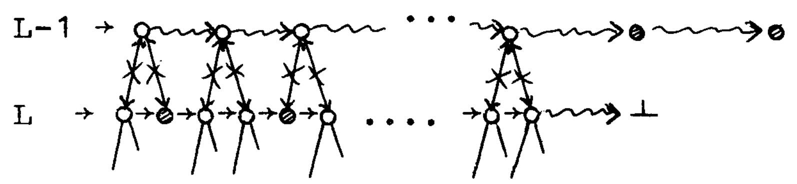
Demerging could be used to make quicksort stable (Knuth 1972, p. 114-123; Sedgewick 1975).

Proof of theorem 2.1. First use algorithm B to build an optimal Huffman tree with the enumeration property in linear time. Make this tree into a doubly-linked or "chained" tree



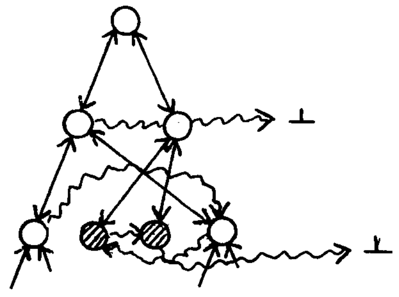
level after level downwards, which we can in time proportional to the number of nodes in the tree (again linear in n). Note that strictly B could never have a black left-son if algorithm B is executed carefully, but one can relax the case (IN$\uparrow$,Q$\uparrow$) without violating the enumeration property. We shall continue the example as a most general situation.

Start at the highest level in the tree where leaves occur, pretend all nodes in the level are temporarily cut loose from their father in the previous level

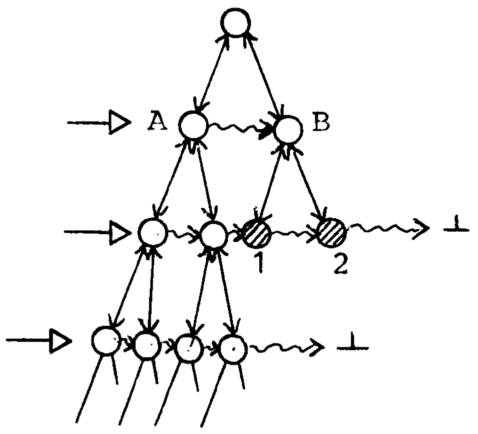


and demerge level L with algorithm C. Note that we do not have to restore the cross-links among the subtrees attached to white nodes due to the stability of algorithm C. When algorithm C is completed, traverse the list in the father-level and in the present level simultaneously to restore the father-son links for nodes now in proper order.

The example tree becomes



which is equivalent to



Repeat the same procedure at the next lower level where leaves occur, and continue until the entire tree is rearranged. This phase takes time proportional to the number of nodes per level, and is therefore completed in linear time as well. ■

HUFFMAN-TREES IN GENERAL AND ALPHABETIC TREES IN GENERAL

Algorithm A (see 2.3) suggests a new organization of Huffman's original algorithm to build an optimal tree for an arbitrary set of records $r_1, r_2, \ldots, r_n$ with weights $w_1, w_2, \ldots, w_n$. First sort the records with a fast (and perhaps stable) procedure and then apply algorithm A to get an optimal tree within only a linear number of more steps. This organization shows that in the ordinary algorithm for obtaining Huffman-trees it isn't really necessary to use a priority queue because all sorting needed can be performed by an especially designed routine once and for all in the beginning.

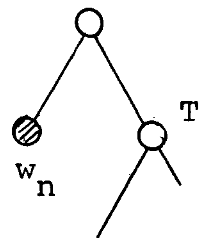
The algorithm gives an optimal Huffman tree in O(n log n) steps, but more so than the original algorithm it shows that this bound is mainly needed for sorting the original set of records. It holds the clue to an argument that we cannot improve on O(n log n) in any algorithm for optimal trees that uses simple primitives.

A binary tree is called skinny if and only if each internal node has at least one son which is a leaf. Assume that we have records $r_1, \ldots, r_n$ with weights $w_1 \leq w_2 \leq \ldots \leq w_n$ given in arbitrary order, and assume further that each w_i is "much larger" than w_{i-1}.

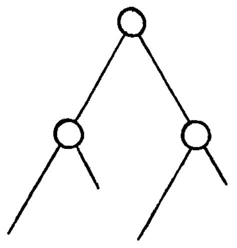
Lemma 3.1.

If $w_j \geq w_{j-1} + \ldots + w_1$ for $2 \leq j \leq n$, then any optimal Huffman tree for $r_1, \ldots, r_n$ is necessarily a skinny tree.

Proof. By induction. Assume the hypothesis for optimal trees on $r_1, \ldots, r_{n-1}$, and consider all possible optimal trees on $r_1, \ldots, r_n$ under the given condition. An optimal tree is either of the form

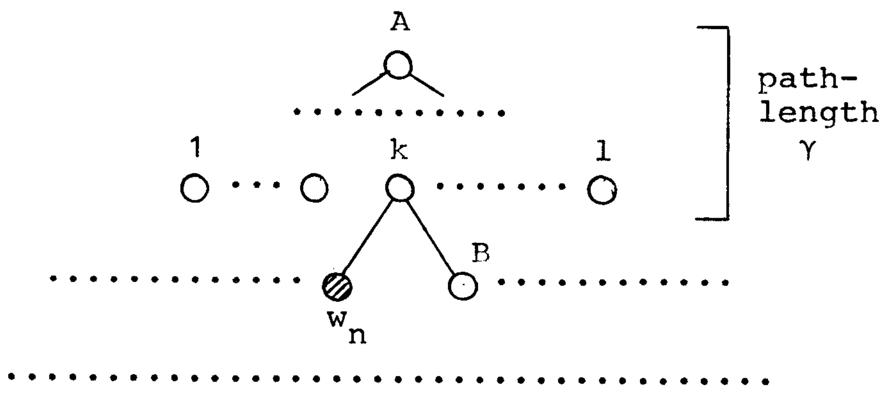


and therefore skinny as a whole by induction on T, or it is of the form

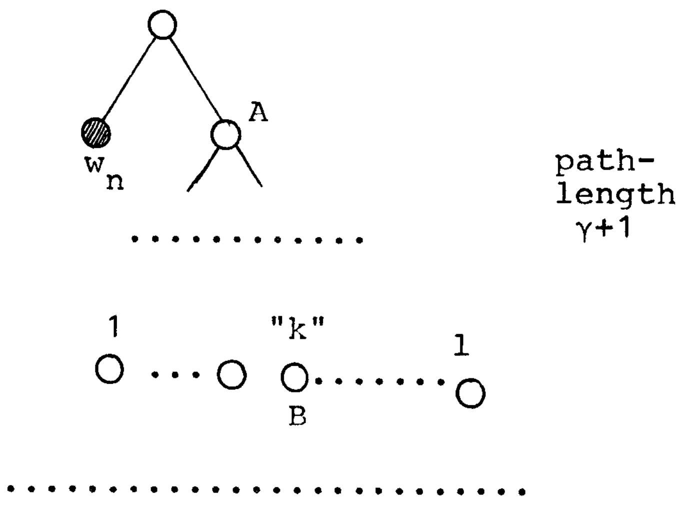


and some further analysis is needed.

Since the tree is optimal, w_n must occur in the first level of the tree containing leaves.



Compare the tree with



(with subtree B pulled "up" into node k), and calculate their weights.

Let $\delta(B)$ equal 0 when B is a leaf and 1 otherwise. It is easily verified that:

$$\begin{aligned}
&\text{weight first tree} - \text{weight second tree} = \\
&= \gamma.(w_n+\ldots+w_1) + \text{weight trees } 1,.\underset{k}{/}.,1 + w_n + \\
&\quad \textstyle\sum \text{leaves}(B) + \delta(B)\text{weight}(B) \\
&-\{w_n+(\gamma+1)(w_{n-1}+..+w_1)+\text{weight trees } 1,.\underset{k}{/}.,1 + \\
&\quad \delta(B).\text{weight}(B)\} = \\
&= \gamma w_n - (w_{n-1}+\ldots+w_1) + \textstyle\sum \text{leaves}(B) \geq \\
&\geq w_n - (w_{n-1}+\ldots+w_1) + \textstyle\sum \text{leaves}(B) \geq \\
&\geq \textstyle\sum \text{leaves}(B) > 0
\end{aligned}$$

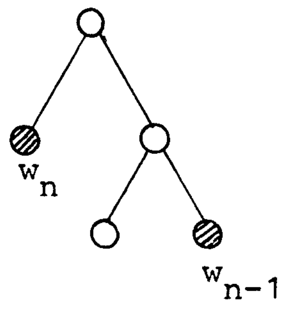
which contradicts minimality of the first tree. This completes the induction-step. ■

From the proof is clear that one can weaken the condition on the weights to

$$w_j > w_{j-1} + \ldots. + w_2 \quad (j \geq 3)$$

and still get the result stated.

Suppose there is an algorithm that can produce an optimal Huffman tree on n records with given weights in $T(n)$ steps. If we let it operate on a set of records $r_1,\ldots,r_n$ with weights $w_1 \leq w_2 \leq \ldots. \leq w_n$ sufficiently far apart (as made precise in 3.1) initially given in some arbitrary order, then it must necessarily deliver a skinny tree



. . . .

We only need a linear number of additional steps to traverse the tree and retrieve the weights in order. (If the weights w didn't satisfy 3.1, then work with sufficiently high powers w^k instead). <u>It shows that any algorithm for optimal Huffman trees can be used for sorting and O(n log n) must be a lower bound on</u> $T(n)$ unless rather sophisticated analytical tests are employed in such an algorithm.

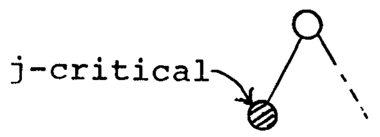
It is much harder to give a combinatorial analysis of "good" alphabetic trees. An optimal alphabetic Huffman tree need not be optimal in the original sense (see e.g. Knuth 1972, p 443). Known algorithms for optimal alphabetic trees therefore use much less straight-forward iterative constructions, and are more time consuming than one would like.

Gilbert & Moore (1959; also Knuth 1972, p. 445) proved that the weighted path-length of an optimal alphabetic tree is bounded by $-\sum w_i \log w_i$ (the entropy) and $-\sum w_i \log w_i + 2$. The upperbound is constructive. We shall give a simpler combinatorial construction which yield an almost optimal alphabetic tree in <u>linear</u> time. (Fredman 1975 gave linear time algorithm for certain internally balanced trees which by Mehlhorn's argument (1975) must have a weighted path-length close to the entropy also).

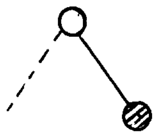
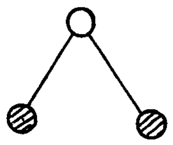
Let g be a linearly bounded, integer function which is $\neq 0$ for $\neq 0$-arguments (like $\lceil\;\rceil$). Make $g(w_j n)$ copies of r_j for each $j=1,\dots,n$, and build a binary tree of minimum path-length$\leq \lceil \log(g(w_1 n)+\dots+g(w_n n)) \rceil$. A node is j-<u>critical</u> if and only if all its descendant leaves carry r_j and there is no node less deep in the tree with that property. Make it a binary search-tree by assigning queries <u>topdown</u>

as follows:

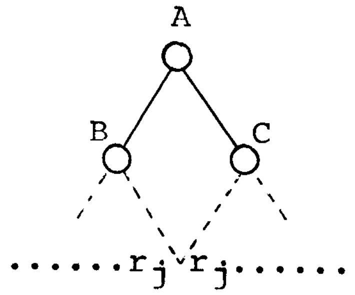
(i) for each node of the form



→ <u>if</u> $x \le \text{key}(r_j)$ <u>then</u> r_j <u>else</u> go right;

(and likewise for  or )

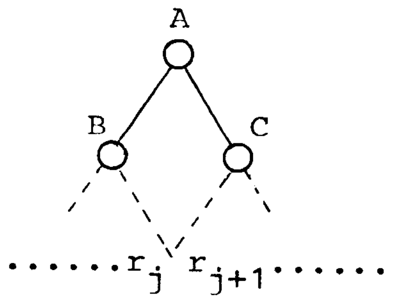
(ii) for each node of the form



where A is not j-dedicated and neither B nor C are j-critical, there <u>must</u> be a j-critical node below B or below C (say, below B). Assign to A:

<u>if</u> $x \le$ key (r_j) <u>then</u> go left <u>else</u> go right;

(iii) for each node of the form



where neither B nor C are critical, simply assign to A:

<u>if</u> $x \le$ key (r_j) <u>then</u> go left <u>else</u> go right;

(iv) nodes not in the previous categories may be purged.

The queries always direct the search for x (presumably $\equiv r_i$) to an i-critical node, and it is obviously the fastest way of identifying it. The height of the father of an i-critical node is bounded by the smallest t such that $d_t=0$ in the recurrence : $d_0=g(w_i n)$, $d_s=[\frac{d_{s-1}-1}{2}]$ for s=1,2,..., which is $t=[\log(g(w_i n)+1)]$. The path-length to an i-critical node is bounded by $\lceil \log(g(w_1 n)+\ldots+g(w_n n)) \rceil - [\log(g(w_i n)+1)]+1$, but it will normally be less as the tree condenses while purging the redundant nodes.

For $\alpha>0$ let $g(x)=2^{\log(\alpha x+1)+\beta_x}$, where β_x is the smallest non-negative number needed to make the exponent integer >0 (thus $\beta_x<1$). The weighted path-length in the tree obtained is bounded by

$$\sum_1^n w_i\{\lceil \log(\alpha+1+2^{\beta_{max}})n\rceil - [\log(g(w_i n))]+1\} \leq$$

$$\leq \sum_1^n w_i\{\log(\alpha+1+2^{\beta_{max}})n-\log(\alpha w_i n)-\beta_{w_i}+2\} \leq$$

$$\leq -\sum_1^n w_i \log w_i+2-\sum_1^n w_i \beta_{w_i}+\log(1+\frac{1+2^{\beta_{max}}}{\alpha}) \leq$$

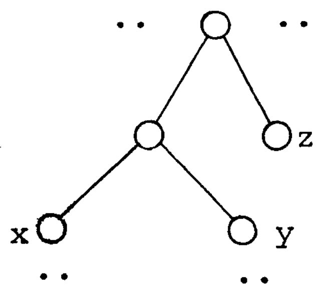
$$\leq -\sum_1^n w_i \log w_i+2+\log(1+\frac{3}{\alpha})$$

By choosing α sufficiently large the bound can be made arbitrarily close to "the entropy + 2", which must in turn be very close to optimal.

CONSTRUCTING AND UPDATING WEAKLY STABLE ALPHABETIC TREES

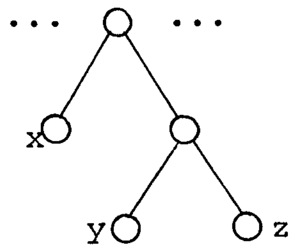
It appears that optimal alphabetic trees are even harder to maintain dynamically than ordinary optimal trees are, and with good reason one may ask for weaker, yet "good" alphabetic trees which allow more easily for updates when records are inserted or deleted or when current weights of records change. The need was recognized before (see Bruno and Coffman 1972; Walker and Gotlieb 1972; Nievergelt and

Reingold 1972), and the answer was usually found in "balancing". Examine what specific kind of balance must appear in optimal alphabetic Huffman trees, and investigate it as a criterion for "near" optimality. A binary tree is called _weakly stable_ if and only if there is _no_ local structure



with $x > z$

or



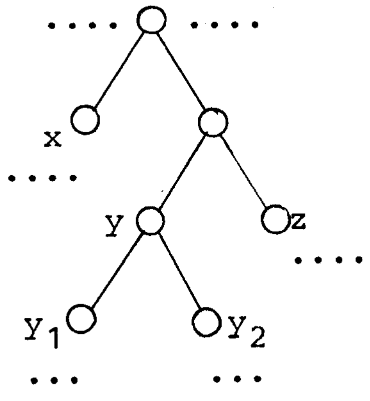
with $x < z$

with x, y, _and_ z _denoting the sum of the weights of the leaves in the subtree under each particular node_.

Lemma 4.1.

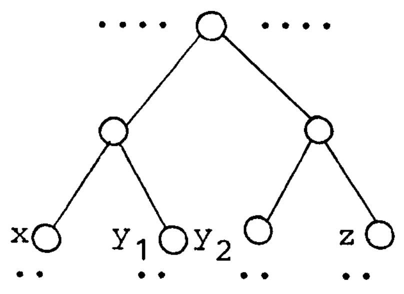
Any optimal alphabetic Huffman tree on n records must be weakly stable.

Note that there are more stabilities in optimal trees. For instance, in a local structure



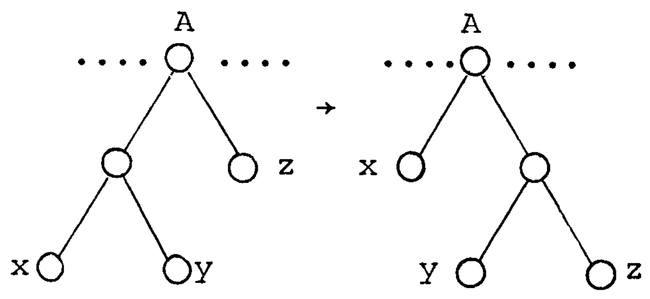
(with $x \geq z$)

one must also have $x \geq y$, since otherwise the alphabetically equivalent re-structured tree

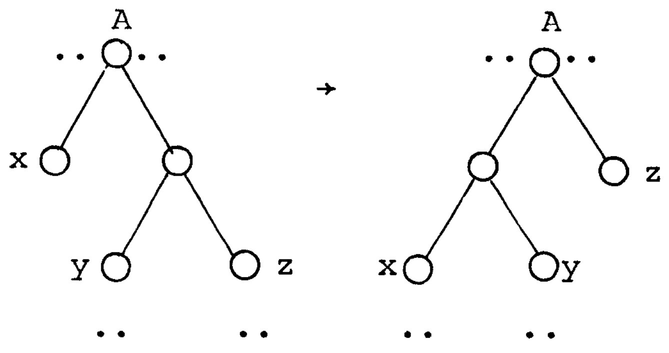


would beat it. Can similar properties together yield a combinatorial characterization of optimal alphabetic trees?

Local rotations

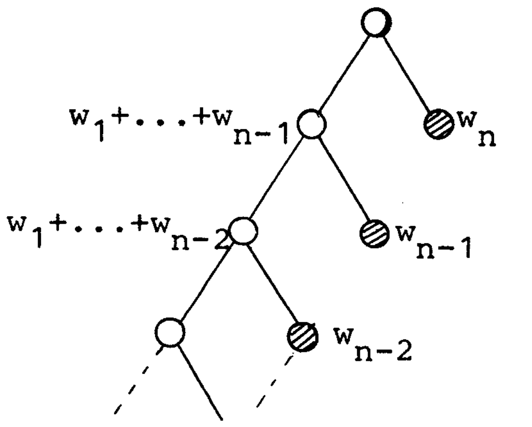


when $x > z$

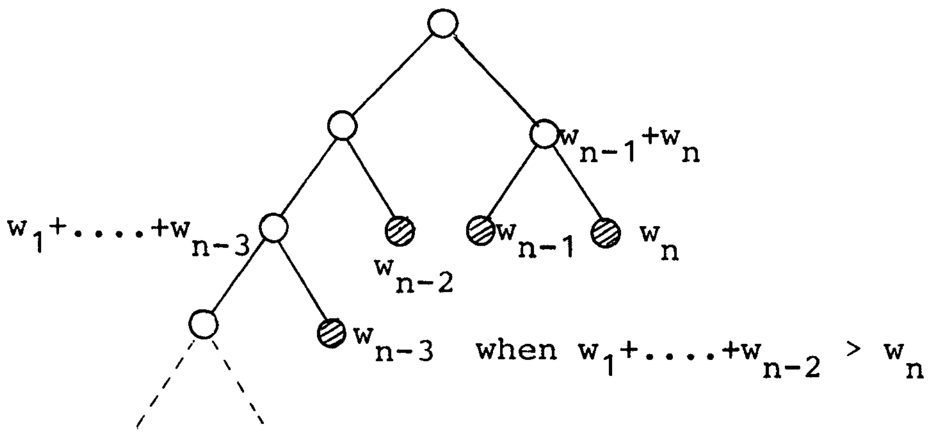


when $x < z$

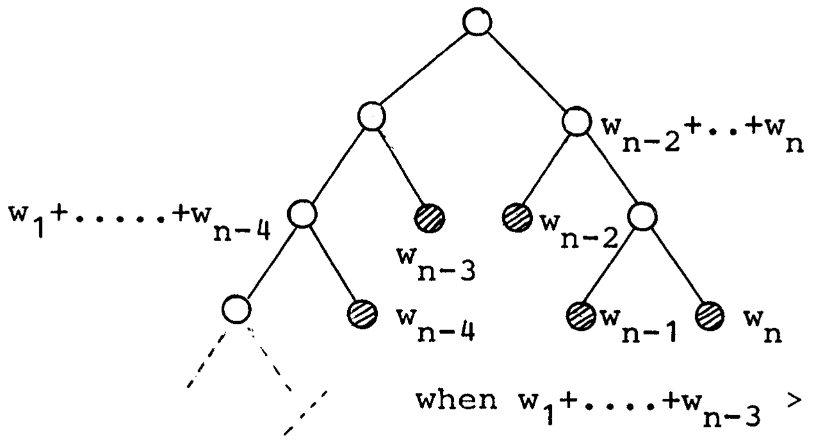
provide a key to stepwise balancing of the tree. Each such a rotation reduces the current weight of the tree and it follows that the process must eventually terminate at a weakly stable tree, no matter what sequence of (legal) rotations is chosen. (There is apparently no point in rotating when $x = z$, see however Appendix). In principle one could start with <u>any</u> given tree on a set of records and "transform" it but some initial trees are better than others. Let us begin with a "first approximation"



and "rotate" to a stable configuration at the root:

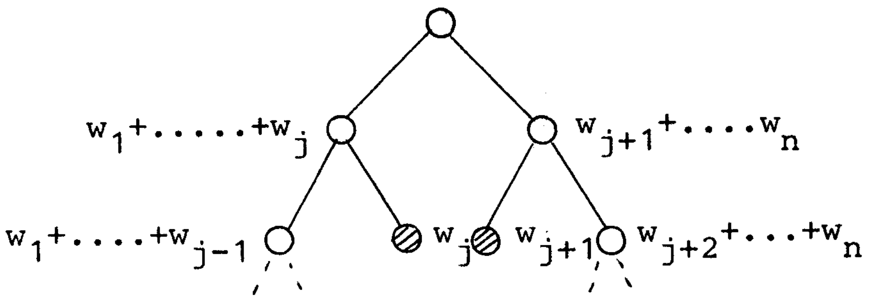


when $w_1+\ldots.+w_{n-2} > w_n$



when $w_1+\ldots.+w_{n-3} > w_{n-1}+w_n$

and so forth, until a configuration



is reached where

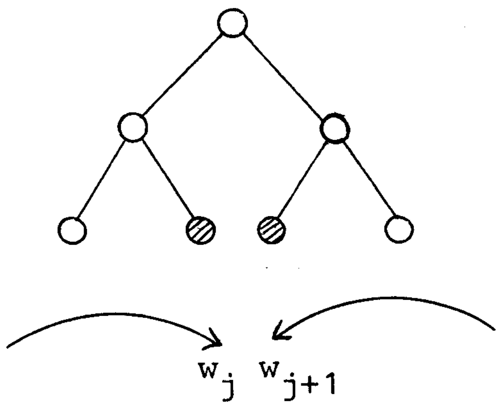
$$w_1+....+w_{j-1} \leq w_{j+1}+...+w_n$$
$$w_1+....+w_j \geq w_{j+2}+...+w_n$$

for some $1 \leq j \leq n-1$.

<u>Lemma</u> 4.2.

Such an index j exists and it is unique (plus or minus 1).

As in Walker & Gotlieb (1972), the position between w_j and w_{j+1} may be called the <u>centre</u>. Observe that the left- and right-subtree of the root are again similar to the start-configuration and the very same procedure of rotation may be used to balance these trees around their root, and so on ...

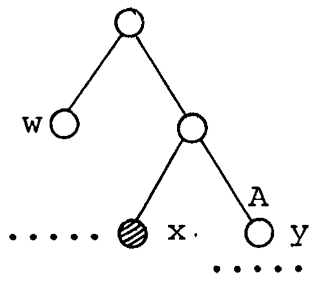


The following observation is essential.

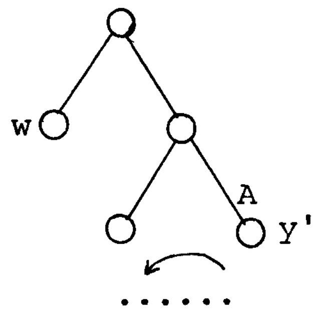
<u>Lemma</u> 4.3.

In balancing the subtrees the configuration at the root remains balanced.

<u>Proof</u>. Denote the original, "balanced" configuration at the root by



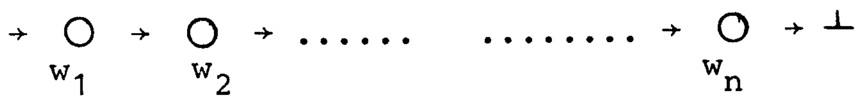
Observe what happens if we begin to rotate in the right subtree



The sum of weights w for the left subtree remains unchanged, but the value of y can decrease. It follows that the original inequality $w \geq y$ cannot go in the "wrong" direction and always $w \geq y'$. The same argument holds for the left subtree and it follows that the root remains in balance. ■

Observe that 4.3 holds only because we insisted on the particular initial organization with a skinny tree. A straight-forward implementation on a general purpose computer will require $O(n^2)$ steps, and mainly so because of all the step-by-step rotations which one may have to perform. Inefficiency could be eliminated with a better method to determine where the centres are.

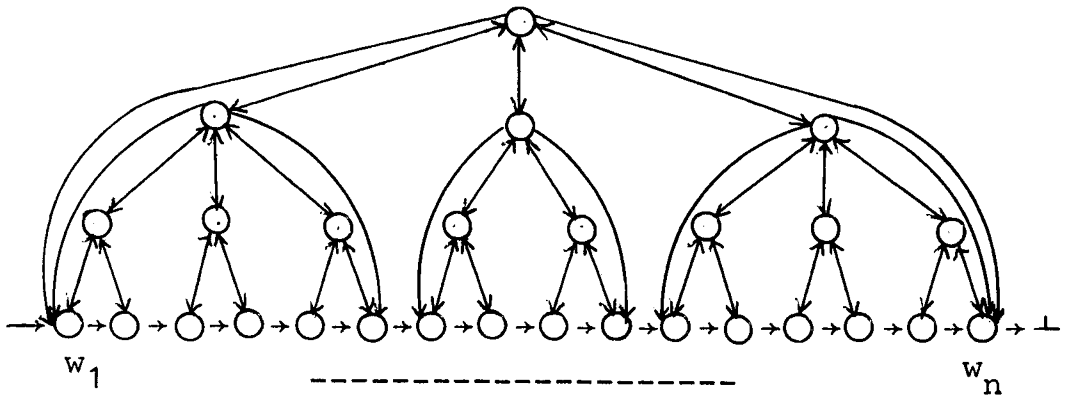
Assume records (and their weights) given in a linear list



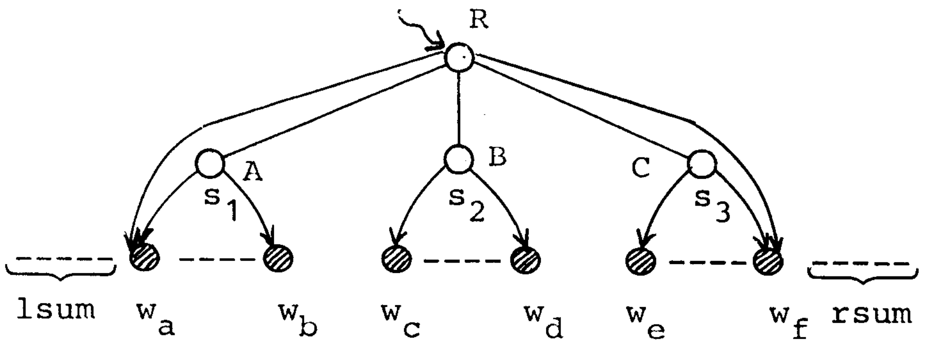
and build a 2-3 tree (see Knuth 1972, p. 468, or Aho, Hopcroft, and Ullman 1974, Ch. 4) on these nodes in which internal nodes are records

father	l.son	m.son	r.son	l.link	r.link	S

containing direct links to the leftmost and rightmost leaves in the attached subtree and the sum (S) of the weights at all leaves in that subtree



Such a linked tree is easily constructed level after level bottom-up in only linear time. One can determine where a centre is by descending down the tree, each time asking whether the centre is in the left, middle, or right subtree before descending further. In order to prove how one can direct the search entirely with local information, consider a typical situation



where we arrived in R, knowing that the centre must be inside its subtree. From the previous steps we know lsum and rsum which gradually accumulated to their present value.

The centre is right between w_b and w_c if and only if

$$\text{lsum} + s_1 - w_b \leq s_2 + s_3 + \text{rsum} \qquad (*)$$

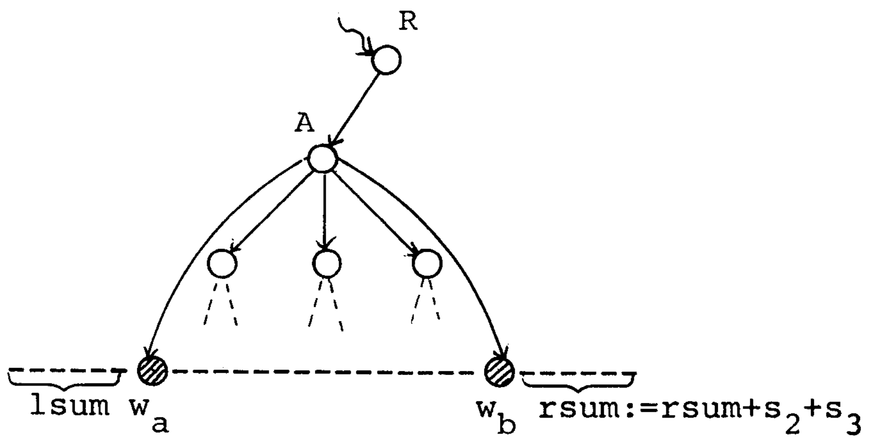
and

$$lsum + s_1 \geq s_2 - w_c + s_3 + rsum \qquad (**)$$

(in which case the search can end), but if any of these inequalities isn't satisfied (one always is) there can only be two cases:

$>$ in (*), and $>$ in (**):

Then we must rotate back, knowing that the centre is in the tree of A. We must move on to A, "update" the local information to



and continue the search in the same way

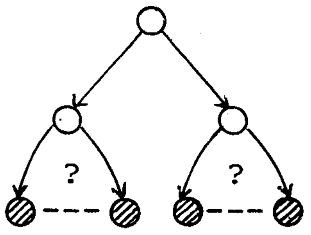
$\leq$ in (*), and $<$ in (**)

Then we know that the centre must lie in B, in between B and C, or in C, and we can resolve where it is precisely in the same way as we did for A.

Lemma 4.4.

The algorithm finds a centre in O(log n) steps.
Proof. The path-length in a 2-3 tree on n records is O(log n), and the amount of work needed at each node in order to determine the next step downwards is bounded by a constant. ■

Once a centre is found we can create the "top-most" part of the weakly stable tree



and repeat the process to find the balanced versions of the left and right subtree and so on, slowly working downwards (see lemma 4.3).

In order to continue in a subtree one need not start all over constructing another 2-3 tree (requiring, possibly, another O(n) steps again), but one can split the original 2-3 tree around the centre that we found and slice it into two pieces (which are again 2-3 trees) which we can use to continue the construction at the left and at the right. In the splitting algorithm as described in Aho, Hopcroft, and Ullman (1974, sect. 4.12) it is no problem to modify the links and sum values for the nodes right on the edge. Thus the SPLIT can indeed be achieved in O(log n) steps.

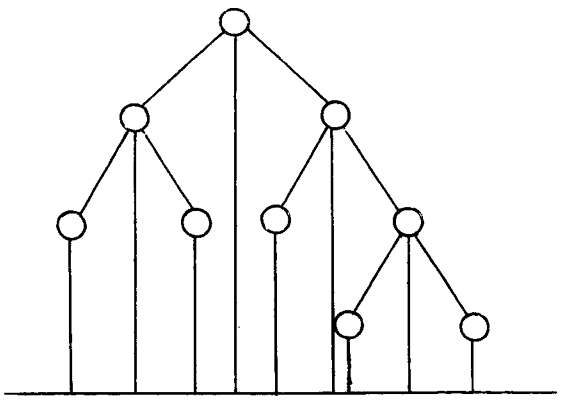
Theorem 4.5.

There is an algorithm for producing a weakly stable alphabetic Huffman-tree on n records in O(n log n) steps.

The conclusion is that with a more convenient data-structure (illustrating the use of 2-3 trees in programming) we could reduce the complexity to O(n log n), saving substantially over explicit rotations.

Assume that a weakly stable tree has been build, and that there is a "small" change in weight at one of the leaves. One is tempted to restore the balance by a bottom-up progression of local rotations, but this may not always be the most efficient way of doing it and we shall argue why top - down rebalancing is preferred.

The previous construction does not only give a weakly stable tree, but at the same time it yields a second tree which shows abstractly what the location of the centres is



and for each centre what the "left-sum" and the "right-sum" locally are. If we change a weight, then all centres move a little, and way down in the lower levels some of the centres may collapse into one while others may have to be split a few times more. Transmitting the changes down the tree it isn't hard to see that a top-down algorithm can re-compute the centres using the information from the old tree, and one may re-build the weakly stable, alphabetic Huffman tree accordingly. It follows that if the changes aren't radical the centres can move only a bounded distance and updating the Huffman-tree takes at most $O(n)$ steps. The same holds for insertion or deletion of a record, and on the average the behavior may be even better if we keep a rather homogeneous set of records. For most practical situations the trees exhibit good dynamic behavior, thus making the present direction for a combinatorial characterization of near-optimality in alphabetic Huffman trees into a subject worthy of further investigation.

REFERENCES

Aho, A.V., J.E. Hopcroft, and J.D. Ullman. The design and analysis of computer algorithms. Addison-Wesley, Reading, Mass. (1974).

Bruno, J., and E.G. Coffman Jr. Nearly optimal binary search trees. Proc. IFIP Congress 71, pp. 99-103, North Holland Publ. Comp, Amsterdam (1972).

Byrne, J.G. Hu - Tucker minimum redundancy alphabetic coding method. CACM 16 (1973) 490.

Fredman, M.L. Two applications of a probabilistic search technique: sorting X+Y and building balanced search trees. Proc. 7th Annual ACM Symposium Theory of Computing, Albuquerque (1975) 240-244.

Gilbert, E.N., and E.F. Moore. Variable length binary encodings. Bell Systems Techn. J. 38 (1959) 933-968.

Hu, T.C. A comment on the double-chained tree. CACM 15 (1972) 276.

Hu, T.C. A new proof of the T-C algorithm. SIAM J. Appl. Math 25 (1973) 83-94.

Hu, T.C., and K.C. Tan. Path length of binary search trees. SIAM J. Appl. Math. 22 (1972) 225-234.

Hu, T.C., and A.C. Tucker. Optimal computer search trees and variable length alphabetic codes. SIAM J. Appl. Math. 21 (1971) 514-532.

Huffman, D.A. A method for the construction of minimum-redundancy codes. Proc. I.R.E. 40 (1952) 1098-1101.

Karp, R.M. Minimum redundancy coding for the discrete noiseless channel. I.R.E. Trans. IT-7 (1961) 27-35.

Knuth, D.E. The art of computer programming, Vol. I: Fundamental algorithms. Addison-Wesley, Reading, Mass. (1968).

Knuth, D.E. The art of computer programming, Vol. III: Sorting and searching. Addison-Wesley, Reading, Mass. (1972).

Mehlhorn, K. Nearly optimal binary search trees. Acta Informatica 5(1975) 287-295.

Nievergelt, J., and E.M. Reingold. Binary search trees of bounded balance. Proc. IVth Ann. ACM Symp. Theory of Comp., Denver, Colorado (1972), pp. 137-142.

Patt, Y. Variable length tree structures having minimum average search time. CACM 12 (1969) 72-76.

Schwartz, E.S. An optimum encoding with minimum longest code and total number of digits. Inform & Control 7 (1964) 37-44.

Schwartz, E.S., and B. Kallick. Generating a canonical prefix encoding. CACM 7 (1964) 166-169.

Sedgewick, R. Quicksort. Ph.D. Thesis, STAN-CS-75-492, Stanford University, Stanford, California (1975).

Sussenguth, E.H. Use of tree structures for processing files. CACM 6 (1963) 272-279.

Walker, W.A., and C.C. Gotlieb. A top-down algorithm for constructing nearly optimal lexicographic trees, in: R.C. Read (Ed.), Graph theory and computing, Acad. Press, New York (1972), pp. 303-323.

Yohe, J.M. Hu-Tucker minimum redundancy alphabetic coding method, Algorithm 428. CACM 15 (1972) 360-362.

Zimmerman, S. An optimal search procedure. Amer. Math. Monthly 66 (1959) 690-693.

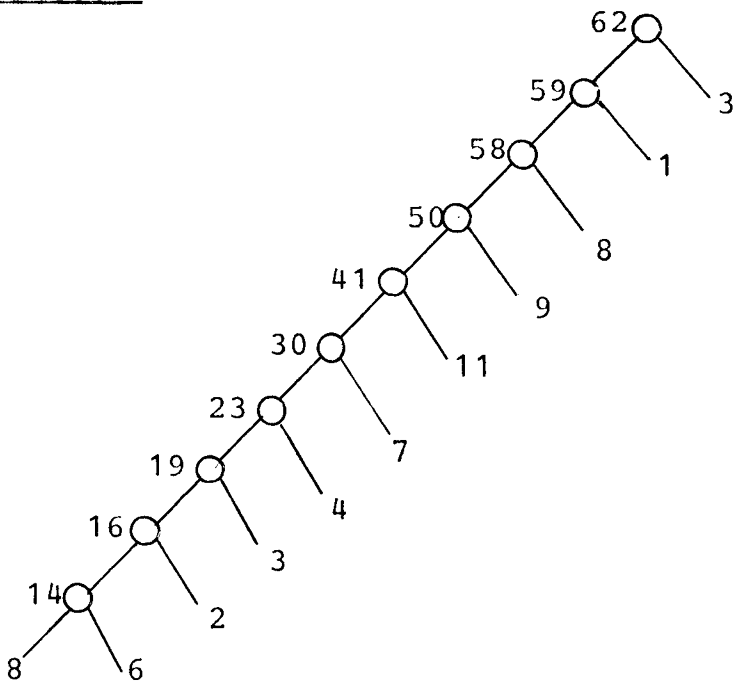
*Note added in proof: at the time of submitting the revised version of this manuscript another paper appeared which is relevant to the present study:

Itai, A. Optimal alphabetic trees. SIAM J. Computing 5 (March 1976) 9-18.

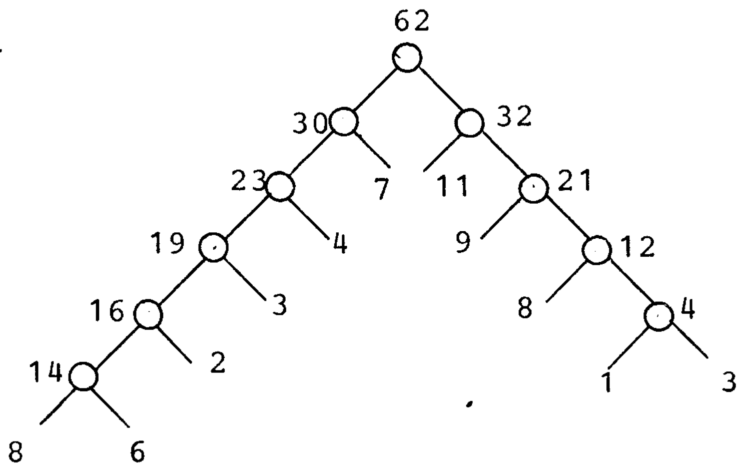
APPENDIX

We shall here apply the algorithm of section 4 to produce a weakly stable alphabetic Huffman tree for the same example used by Hu & Tucker (1971).

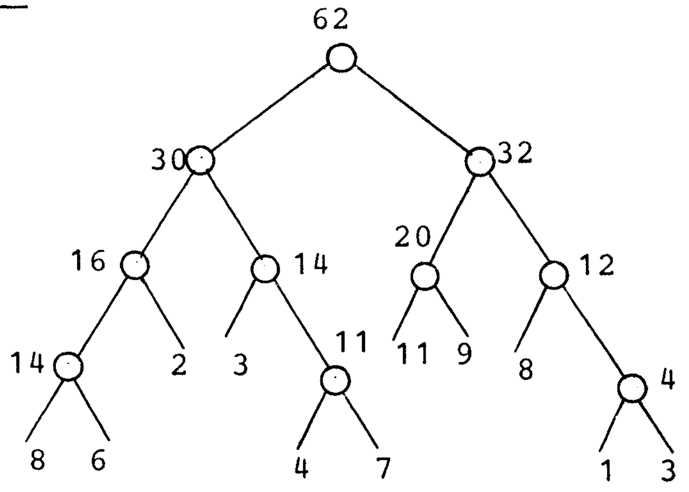
Stage 1



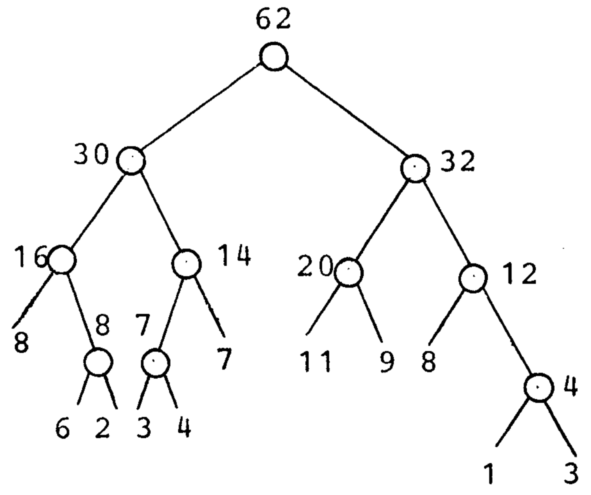
Stage 2



Stage 3



Stage 4



It so happens that this tree is weakly stable, but not optimal: moving "6" up and combining "2" and "3" one level lower yields a tree with a weight slightly less. Observe that we could have arrived at it had we resolved the 14-versus-14 clash in stage 3 in favor of the left subtree! (It suggests an explicit strategy is needed for resolving such situations in the algorithm.)

AUTOMATA LANGUAGES AND PROGRAMMING

Third
International
Colloquium
at the
University
of
Edinburgh
edited
by
S.Michaelson
and
R.Milner

20.21.22.23
July
1976

*

Edinburgh
University Press