LINEAR TIME GENERATION OF A NEW FIXED-LENGTH DATA-COMPRESSION CODE

Jan van Leeuwen

RUU-CS-78-3

February 1978

Rijksuniversiteit Utrecht

Vakgroep Informatica

Utrecht 2506
Telefoon 030-53 1454

LINEAR TIME GENERATION OF A NEW FIXED-LENGTH DATA-COMPRESSION CODE

Jan van Leeuwen

Technical Report RUU-CS-78-3

February 1978

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht, the Netherlands

all correspondence to:

Dr. Jan van Leeuwen
Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

LINEAR TIME GENERATION OF A NEW FIXED-LENGTH DATA-COMPRESSION CODE

Jan van Leeuwen
Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

Abstract. Recently J. Verhoeff suggested an interesting data-encoding and compaction technique based on fixed-length codes. Given a permissible code-length, the idea is to assign code-words to variable-length strings in such a way that the average length per bit is maximized. We prove that the method of Verhoeff for constructing a maximal prefix code is essentially unique, and show that the construction can be implemented to run in linear time and "real" space.

1. Introduction.

There is a steadily increasing use of data-encoding techniques in all modern systems for information storage and retrieval, for purposes of both security (ciphers) and data-compression (see e.g. Martin [3], [4]). A typical data-file needs one byte per character, sometimes less if the system has been designed so as to pack and unpack words itself. Wiederhold [7] (Chapter 14) gives a rather complete account of existing data-compression techniques which can bring major savings in tape- or disk-space for both numeric and character data.

It has been known for some time that a considerable compaction of data-files may be achieved by recoding the character-set from standard bytes to a minimum-redundancy Huffman-code, which takes advantage of empirical knowledge about the frequency-distribution of individual characters in the data-file (see Knuth [2], Martin [4], or Wiederhold [7]). A different technique was proposed recently by Verhoeff [6], who observed that in practice there will be a considerable advantage in keeping to a fixed-length code. Given a permissible code-length (typically an integral fraction of a computer's word-size), the idea is to assign code-words to variable-length character-substrings in such a way that the average length per bit is maximized. It should be noted that this approach is dual to Huffman's, which assigns variable-length code-words to fixed-length character-substrings under a similar optimizing criterion.

Let $\Sigma = \{a_1, \ldots, a_k\}$ be an alphabet of k characters, and let p be a known frequency distribution on Σ with $\sum_i p(a_i) = 1$. A prefix-coding over Σ is some

tree T whose internal nodes have degree k and external nodes or leaves r carry the designated 0-1 codeword for the string which describes the unique path from the root to r (with a_i at each internal node indicating that the i^{th} branch should be followed). Let r be reached by the path $a_{i_1} \ldots a_{i_s}$. The path length of r is defined as $\pi(r) = s$, and its weight as $w(r) = p(a_{i_1}) \ldots p(a_{i_s})$. The weight of a leaf, or rather the corresponding character-substring, is to be interpreted as the expected frequency of this string in the data-file. The average weighted pathlength of a prefix-coding T with N leaves is defined as

$$y(T) = \frac{1}{N} \sum_r w(r).\pi(r) \qquad (1.1)$$

Given a permissible code-length b, Verhoeff's idea is to design a prefix-coding T with n internal nodes for some (perhaps the largest) n with

$$b = \lceil \log(N+1) \rceil = \lceil \log(n(k-1)+2) \rceil \qquad (1.2)$$

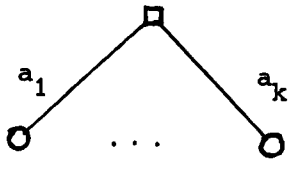
in such a way that $\frac{y(T)}{b}$ is maximum. It means that among all prefix-codings T with n internal nodes we must find one for which in fact y(T) is maximum. Any such tree is called a maximal n-tree.

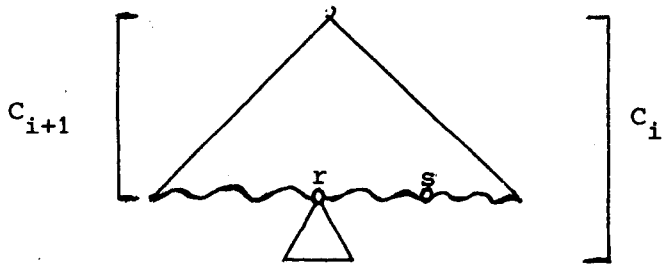
Verhoeff [6] gave a simple algorithm for generating maximal n-trees, based on repeatedly splitting a heaviest leaf of some intermediate tree. In section 2 we shall give a somewhat different proof of the validity of Verhoeff's algorithm, and derive the stronger result that any maximal n-tree must be obtained through the process of splitting a heaviest leaf in some maximal (n-1)-tree. Thus, Verhoeff's algorithm is the only inductive one possible for generating maximal n-trees. In section 3 we show that the algorithm can be implemented in time N and no more space than needed to represent the actual coding tree. Considering that Huffman-trees for N elements require N log N time (see e.g. van Leeuwen [5]), we are able to conclude on quantitative grounds that Verhoeff's codings are strictly easier to generate than Huffman's. We note that the effectiveness of both tends to the Shannon lowerbound for the source entropy when $N \to \infty$.

2. The generation of maximal n-trees.

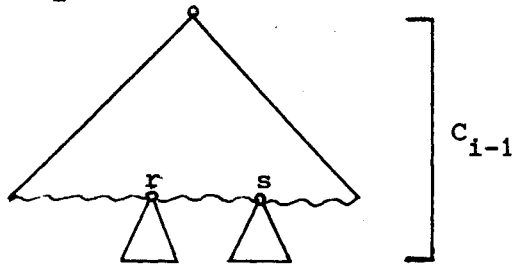
In his paper Verhoeff has given the following construction for a sequence of maximal n-trees $V_1, V_2, \ldots$:

(i) V_1 is the unique 1-tree

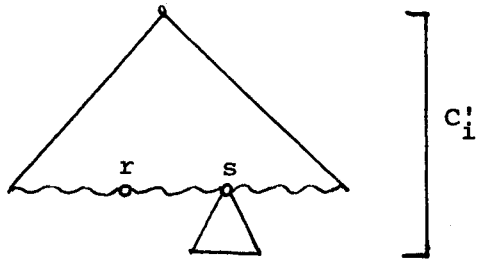




If r ($=r_{i+1}$) was the heaviest leaf, then we are done. Suppose r is <u>not</u> a heaviest leaf of C_{i+1}, but s is. As splitting r only yields leaves of smaller weight than r, the induction hypothesis learns that we may as well assume that $s = r_i$:



Consider C_{i+1}, and suppose that we would construct a tree C_i' instead by splitting s:



It could very well happen that r is a heaviest leaf of the tree this time. Splitting it gives C_{i-1} back, and we have shown that in this case the construction can be <u>reorganized</u> so as to be of the desired form. We shall demonstrate that we can likewise reorganise the construction even when r is not the heaviest leaf at this stage.

If r is not a heaviest leaf of C_i' yet, then splitting a heaviest leaf in it gives a tree C_{i-1}' with

$$C_{i-1}' = C_{i-2} - \{ \overset{r}{\triangle} \}$$

(meaning C_{i-2} with the subtree at r deleted). Following the choices of heaviest leaves as in the original chain for a while gives a chain of trees C_j' with

$$C_j' = C_{j-1} - \{ \overset{r}{\triangle} \}$$

(ii) for $n \geq 1$, V_{n+1} is obtained from V_n by "splitting" a heaviest leaf q_n.

It may very well happen in (ii) that "the" heaviest leaf must be chosen from among several leaves of equal, largest weight. In the construction of the particular sequence $V_1, V_2, \ldots$ we shall always take q_n to be the leftmost qualifying leaf in the highest level of the tree containing heaviest leaves. We shall prove that other algorithms for the inductive generation of maximal n-trees can differ from Verhoeff's only in their choice of an actual heaviest leaf at each stage.

Theorem 2.1. V_n (or any tree obtained by the same algorithm) is a maximal n-tree, and each maximal (n+1)-tree is obtained from a maximal n-tree by splitting one of its heaviest leaves.

Proof

The assertion for V_n is certainly true for n=1. A maximal 2-tree C_0 must be obtained from the (unique) 1-tree V_1 by splitting some leaf r. One may easily verify that necessarily

$$y(C_0) = y(V_1) + w(r)$$

as a result. If r wasn't a heaviest leaf of V_1, then splitting such a heaviest leaf instead would lead to a tree C_0' with $y(C_o') > y(C_0)$. This would contradict the maximality of C_0.

Let the induction hypothesis be true up to n. The inductive assertion for V_{n+1} follows easily, as the hypotheses imply (through an argument as in 2.2) that in fact each (n+1)-tree obtained by splitting a heaviest leaf of some maximal n-tree (viz. V_n) is maximal. In order to complete the induction step, let us consider an arbitrary maximal (n+2)-tree C_0.

There must be an (n+1)-tree C_1 such that C_0 is obtained from C_1 by splitting some leaf r_1. Clearly

$$y(C_0) = y(C_1) + w(r_1)$$

and as before it is easy to argue that r_1 must be a heaviest leaf of C_1. Likewise analysing C_1 and continuing, we claim that a chain of (n+2-i)-trees C_i can be obtained (for i from 1 to n+1) such that for $0 \leq i < n+2$ C_i is obtained from C_{i+1} by splitting a heaviest leaf r_{i+1}. As a result, clearly

$$y(C_o) = y(C_{i+1}) + w(r_{i+1}) + \ldots + w(r_1)$$

We shall verify the claim by induction in i.

For i=1 we just saw that the claim was valid. Consider some tree C_{i+1} from which C_i is obtained by splitting some leaf r:

and we can continue to do so until j=1 or in the original chain we would have to split a heaviest leaf which is not as heavy as r.

In the latter case we would have to split r first, leading from $C'_j = C_{j-1} - \{$  $\}$ to the ordinary tree C_{j-1} at this stage, before the remaining part of the original construction could be accomplished. Clearly, this reorganization of steps brings us to C_0 by a sequence of steps as desired for the inductive proof of our claim.

In the former case (j=1) we would end up with a tree C'_1 after always splitting heaviest leaves of a weight larger than r's:

$$C'_1 = C_0 - \{$$  $$\}$$

If t is a heaviest leaf of C'_1 (taking t=r if r is heaviest), then splitting t gives a tree C'_0 with

$$y(C'_0) = y(C_0) + w(t) - w(r)$$

(and $C'_0 = C_0$ if t=r). It follows that r must be a heaviest leaf at this stage finally, or the maximality of C_0 gets contradicted. Again the desired reorganization of the construction is achieved.

Having verified the claim we can proceed inductively to a 1-tree C_{n+1}, from which C_0 is derived by repeating a procedure of splitting heaviest leaves. As C_{n+1} must be identical to V_1 (by uniqueness at this level) and therefore be maximal, our induction hypothesis shows that all trees C_j for j from n+1 to 1 must be maximal. In particular we have shown that C_0 is obtained by splitting a heaviest leaf of some maximal (n+1)-tree, namely: C_1.

This completes the induction step.

□

The proof shows that Verhoeff's algorithm is unique up to the choice of a heaviest leaf at each stage and the permutation of occurrences of "independent" splittings. We conclude that V_n is "essentially" unique among the maximal n-trees if we consider its weighted structure only:

Theorem 2.2. For each maximal n-tree C there exists a 1-1 correspondence f from nodes of C to nodes of V_n such that for all nodes x in C:

(i) if x is internal, then f(x) is internal in V_n

(ii) if x is a leaf, then f(x) is a leaf of V_n

(iii) $w(x) = w(f(x))$

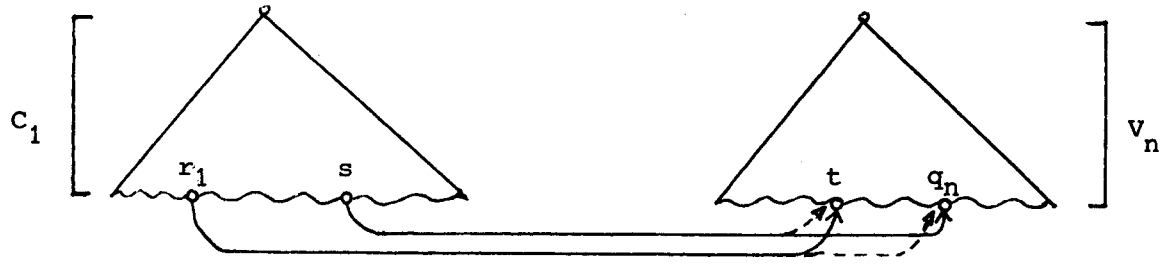
Proof

The result follows directly from the "uniqueness" of Verhoeff's algorithm, but we shall formally prove it by induction in n.

For n = 1 the result is trivial, as there can be only one tree (i.e. $C=V_1$).

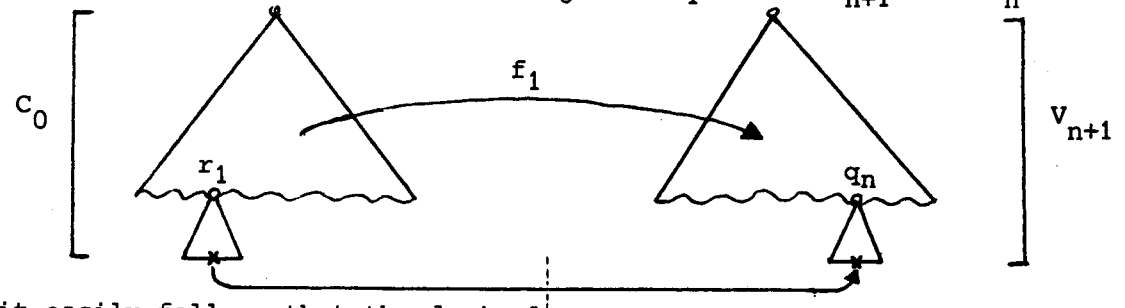
In order to prove the induction step, let us consider some maximal (n+1)-tree C_0. By 2.1. there is a maximal n-tree C_1 such that C_0 is obtained from C_1 by splitting a heaviest leaf r_1. By the induction hypothesis there is a 1-1 correspondence f_1 from C_1 to V_n with the necessary qualifications satisfied. We claim that we may as well assume that $f_1(r_1) = q_n$.

For proving the claim, suppose there was a leaf $s \neq r_1$ of C with $f_1(s) = q_n$ instead. Necessarily s is a heaviest leaf of C also. Let $f_1(r_1) = t$.



We could interchange the f_1-values without harm: $f_1(r_1) = q_n$ and $f_1(s) = t$, and have a correspondence with the desired property.

Considering the construction of C_0 from C_1 and of V_{n+1} from V_n



it easily follows that the desired correspondence f_0 from C_0 to V_{n+1} is obtained by taking

$$f_0(x) = \begin{cases} \text{the } i^{th} \text{ son of } q_n & \text{if } x \text{ is the } i^{th} \text{ son of } r_1 \\ f_1(x) & \text{otherwise} \end{cases}$$

□

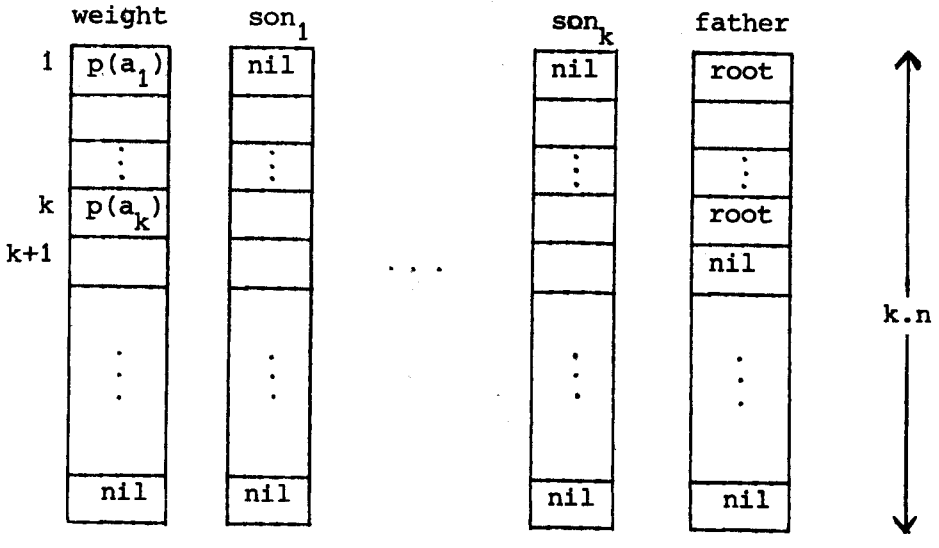
What choice of a heaviest leaf should be made at each stage is considered to be irrelevant for our present purposes, but there may be practical reasons for wanting some very particular strategy. We leave this as an issue for further study.

3. Linear time generation of maximal n-trees.

We shall now develop an actual implementation of Verhoeff's algorithm, without paying much attention to the particular selection-strategy for heaviest leaves at each stage. Thus, the sequence of maximal trees we generate here will not be the very same as $V_1, V_2, \ldots$ but was chosen so as to achieve greatest efficiency in our present algorithmic approach.

A straightforward implementation of Verhoeff's algorithm would make use of a general priority queue (see Knuth [2] or Aho, Hopcroft and Ullman [1]), which administers the current leaves by weight in such a way that there is always a currently heaviest item "on top". When new elements (obtained from splitting the most recently deleted top-element) are inserted, it typically takes O(log m) steps per element for an efficient priority queue containing m elements at the time to "rearrange" itself. As we must do n-1 splits on an initial 1-tree to obtain a maximal n-tree and thus perform k(n-1) or about N insertions in the priority queue in total, the straightforward implementation would run in time proportional to about N log N. In order to improve on this, we shall present a different implementation based on the crucial observation that in this particular algorithm one can permanently decompose the required priority queue into k ordinary queues $Q_1,\ldots,Q_k$. We note that a similar decomposition was shown for Huffman's algorithm acting on an initially ordered sequence (see van Leeuwen [5]).

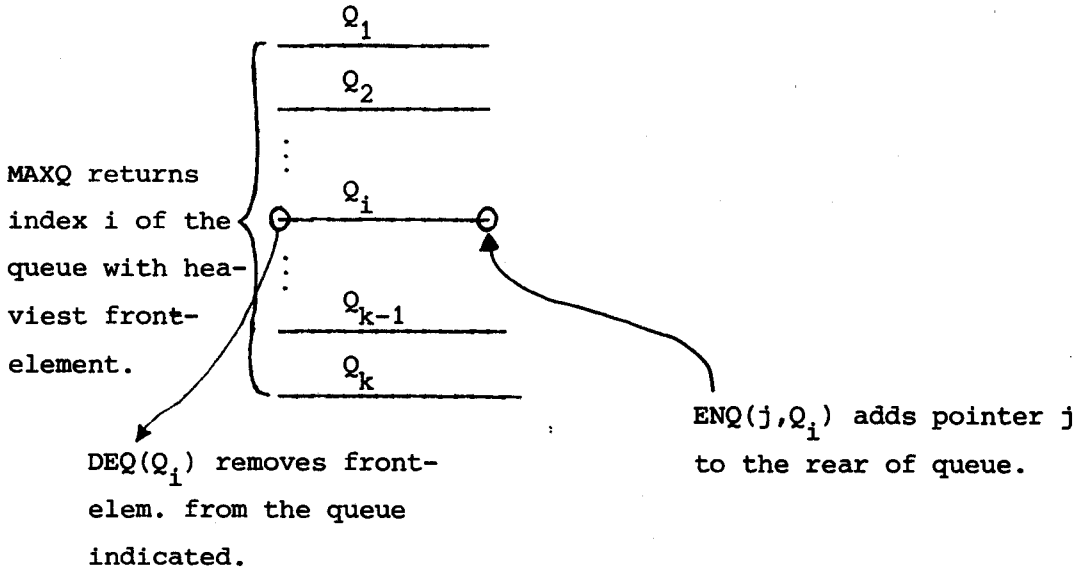
The algorithm we develop shall be generating a maximal n-tree according to Verhoeff's algorithm, for some arbitrary n. In order to represent the resulting tree we must adopt an appropriate record-structure for nodes. To keep it rather language-independent we shall assume that each node-record is a row in the following structure of k+2 arrays, with k.n positions each:



We initialize the structure by entering "records" for the leaves of the unique 1-tree in positions 1 through k. Obviously, k.n rows are precisely sufficient to store all internal and external node-records of the final n-tree. Rows ("record-slots") are allocated one at a time when needed, in direct sequential order. We shall use a function routine NEW which always returns the index of the next available row when called.

Let us assume without loss of generality that $p(a_1) \geq \ldots \geq p(a_k)$.

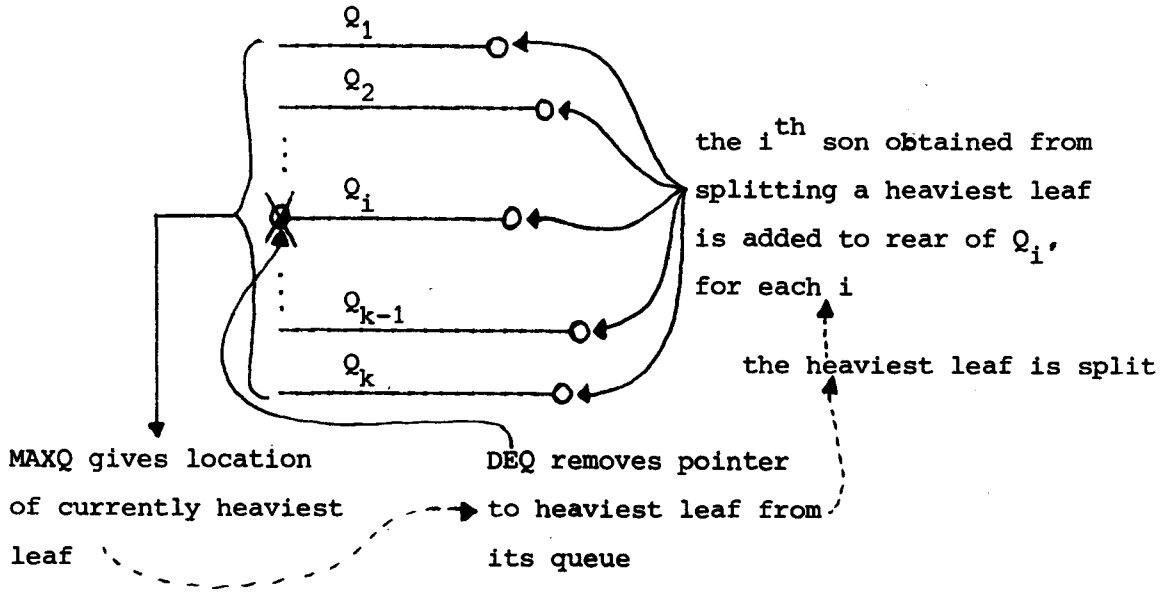
Our algorithm shall be using k queues $Q_1,\ldots,Q_k$ which we leave unspecified as a detailed datastructure and merely manipulate as an abstract datatype. (Implementations are straightforward, but would only confuse the issue here.) Standard operations are:



Each queue contains zero or more pointers to leaf-records in the array-structure. Initially each queue Q_i contains a single pointer i, pointing to the record of the i^{th} leaf in the initial 1-tree. We shall see that, as the algorithm proceeds, the pointers in each queue "automatically" appear in an order $j_1,j_2,\ldots$ such that

$$\text{weight}(j_1) \geq \text{weight}(j_2) \geq \ldots$$

We shall say that the queues have pointers listed in weight-decreasing order. Assuming that this property is maintained (which we shall have to prove), the algorithm cycles through the following transactions until a maximal n-tree is obtained:



Assuming the required initializations have occurred, the algorithm may be formulated as follows:

```
Algorithm 3.1.
     {initializations done}
     count:=1;
     {we have a maximal 1-tree}
     {now cycle through Verhoeff's method n-1 more times}
L1:repeat
     M:=MAXQ;
     i:=DEQ(Q_M);
L2:{claim : i is pointer to some heaviest leaf}
   {split it, and enqueue resulting leaves onto proper Q}
     t:=1;
     repeat
       {allocate record for t^th son}
       j:=NEW;
       son_t(i):=j;
       weight(j):=p(a_t)*weight(i);
       father(j):=i
       ENQ(j,Q_t);
       t:=t+1
     until t>k;
   count:=count+1;
   {at this stage we have a maximal count-tree}
until count=n;
```

Provided the claim at L_2 is right, the algorithm will finish with the representation of a maximal n-tree in store.

Lemma 3.2. Each queue Q_i $(1 \le i \le k)$ has pointers listed in weight-decreasing order, at any time during the execution of algorithm 3.1.

Proof

Use induction on the numbers of steps performed by algorithm 3.1. The assertion certainly holds when L_1 is reached for the first time, because at that time each Q_i contains just one item. Go around the loop once, splitting some leaf of weight $p(a_i)$. The i^{th} queue is emptied, but receives a new leaf of weight $p(a_i) * p(a_i)$, less than $p(a_i)$. The t^{th} queue (for $t \neq i$) keeps its item of weight $p(a_t)$, but a new item of weight $p(a_t) * p(a_i)$ is added to its rear. As $p(a_t) * p(a_i) \le p(a_t)$, we see that the t^{th} queue remains ordered (and the i^{th} queue trivially was).

To prove the induction step, suppose that we have reached L_1 for the u^{th} time (some $2 \leq u < n$) and

(i) the queues are still ordered by decreasing weight (and remaining items are less than or equal to all elements ever removed),

(ii) we have just split a heaviest front-element of weight v (which must have been a heaviest leaf therefore), which was listed in some queue Q_i.

Thus, in particular, the rear-element of each queue Q_t must have weight $p(a_t) * v$ at this stage.

Now suppose that a next heaviest front-element is located at Q_j, having some weight w. If $i = j$, then our assumption on the ordering of the queues shows that necessarily $w \leq v$. If $i \neq j$, then we observe that in the previous round v was decided to be heaviest and necessarily $w \leq v$ also. If the front-element is removed from Q_j and split, then a new item of weight $p(a_t) * w$ is added to the rear of each queue Q_t. When $t = j$ the new item is conceivably added to a queue that just got emptied, otherwise it is listed after an item of weight $p(a_t) * w$. As $p(a_t) * w \leq w$ in the former case and $p(a_t) * w \leq p(a_t) * v$ in the latter, we conclude that in either case the queues remain ordered by decreasing weights (and all remaining items are less than or equal to the last one removed, i.e. w).

□

The lemma shows that the listing of pointers in weight-decreasing order is a loop-invariant for each queue throughout the entire execution of algorithm 3.1. It means that the choice of a heaviest leaf can always be made by looking at front-elements only, and we have verified the claim at L_2 as correct.

Theorem 3.3. Algorithm 3.1 correctly generates a maximal n-tree (for any n), and runs in time O(N).

Proof

Correctness follows from 3.2. The time-bound is derived by noticing that MAXQ works in time O(k), that leaf-splitting and enqueueing new records takes O(k) also, and that the outer repeat-loop is executed precisely n-1 times. We conclude that the algorithm runs through $O(k.n) = O(N)$ steps for the construction of a maximal n-tree.

□

Algorithm 3.1 doesn't use any more additional space than needed for representing the final tree, if we ignore the additional O(N) memory-locations needed for implementing the queues. An interesting problem might be how one

should let MAXQ locate a heaviest front-element if there is a choice between several (more than one). It seems unlikely that algorithm 3.1 can be modified without loss of efficiency to implement just any splitting strategy for heaviest leaves, although it does have considerable flexibility.

4. References

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

[2] Knuth, D.E., The art of computer programming, vol 3: sorting and searching, Addison-Wesley, Reading, Mass. (1973).

[3] Martin, J., Systems analysis for data-transmission, Prentice-Hall Inc., Englewood Cliffs, N.J. (1972).

[4] Martin, J., Computer data-base organization, Prentice-Hall Inc., Englewood Cliffs, NJ. (1975).

[5] van Leeuwen, J., On the construction of Huffman-trees, in S. Michaelson & R. Milner (eds.): Automata, Languages and Programming (Proc. 3rd Colloquium, Edinburgh, July 20-23, 1976), Edinburgh Univ. Press (1976) 382-410.

[6] Verhoeff, J., A new data-compression technique, Annals of Systems Research (1978) (to appear).

[7] Wiederhold, G., Database design, McGraw Hill Inc., New York, NY (1977).