SIAM J. COMPUT.
Vol. 10, No. 3, August 1981

OPTIMAL MULTI-WAY SEARCH TREES*

L. GOTLIEB†

Abstract. Given a set of N weighted keys, $N+1$ missing-key weights and a page capacity m, we describe an algorithm for constructing a multi-way lexicographic search tree with minimum cost. The program runs in time $O(N^3m)$ and requires $O(N^2m)$ storage locations. If the missing-key weights are zero, the time can be reduced to $O(N^2m)$. A further refinement enables the factor m in the above costs to be replaced by $\log m$.

Key words. algorithms, optimal weighted search trees, dynamic programming

1. Introduction. In this paper, we consider the problem of constructing a search tree for use in secondary storage. Because the access cost of the external memory is high compared to internal processing speeds, keys are grouped for searching into storage units called *pages*. Given a set of N keys with weights $\{p_i\}$, $N+1$ missing-key weights $\{q_j\}$, and a page capacity m, we show how to construct a multi-way search tree which minimizes the expected number of pages accessed during a search. Section 2 sets up the problem and reviews Knuth's dynamic programming technique for finding optimal binary search trees [1]. Section 3 extends this method to produce an algorithm for constructing optimal multi-way trees. The running time is $O(N^3m)$, and $O(N^2m)$ storage locations required. If the q weights are absent, a "monotonicity" property (expressed by Theorem 2, which is proved in § 6) enables the running time to be reduced to $O(N^2m)$; however a counterexample presented in § 5 shows that this property need not hold if any of the q's are positive. § 4 outlines a refinement whereby the factor m in the above time and storage costs can be replaced by log m.

Variants of the problem considered here have been addressed by Itai [4] and by Wood et al. [6]. Itai shows how to construct optimal multi-way *code* trees, that is, trees in which the weights are confined to appear at the leaves only. Recently, Wood and his colleagues have shown how to construct optimal multi-way search trees with a height constraint, and also how to optimize multi-way trees when the cost of searching is defined as a combination of page accesses and the time required to search within a page. Their results have been derived independently of this work.

2. The problem. We are given keys $K_1<K_2<\cdots<K_N$, a page capacity $m\geqq 1$, and nonnegative weights $\{p_1,\cdots,p_N, q_0,\cdots,q_N\}$, where

p_i/W is the probability that K_i is the search argument,

q_j/W is the probability of a search between K_j and K_{j+1}, $1\leqq j<N$, and

W is the total weight, $p_1+\cdots+p_N+q_0+\cdots+q_N$.

(q_0/W is the probability that the search argument precedes K_1, q_N/W that it follows K_N).

The problem is to construct an $m+1$-way search tree (an example of which is illustrated in Fig. 1), which minimizes the cost

$$\sum_{i=1}^{N} p_i\cdot \text{level}\,(p_i)+\sum_{j=0}^{N} q_j\cdot(\text{level}\,(q_j)-1).$$

Fig. 1 shows an optimal 4-way tree ($m=3$) for the 15 most common names in the Canadian census. The number accompanying each name (the p-weight) is the number of

* Received by the editors November 24, 1977, and in final revised form June 6, 1980.

† Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A7. Present address, Gellman, Hayward and Partners Ltd., Calgary, Canada T2G 0G3. This research was supported in part by the National Research Council of Canada.

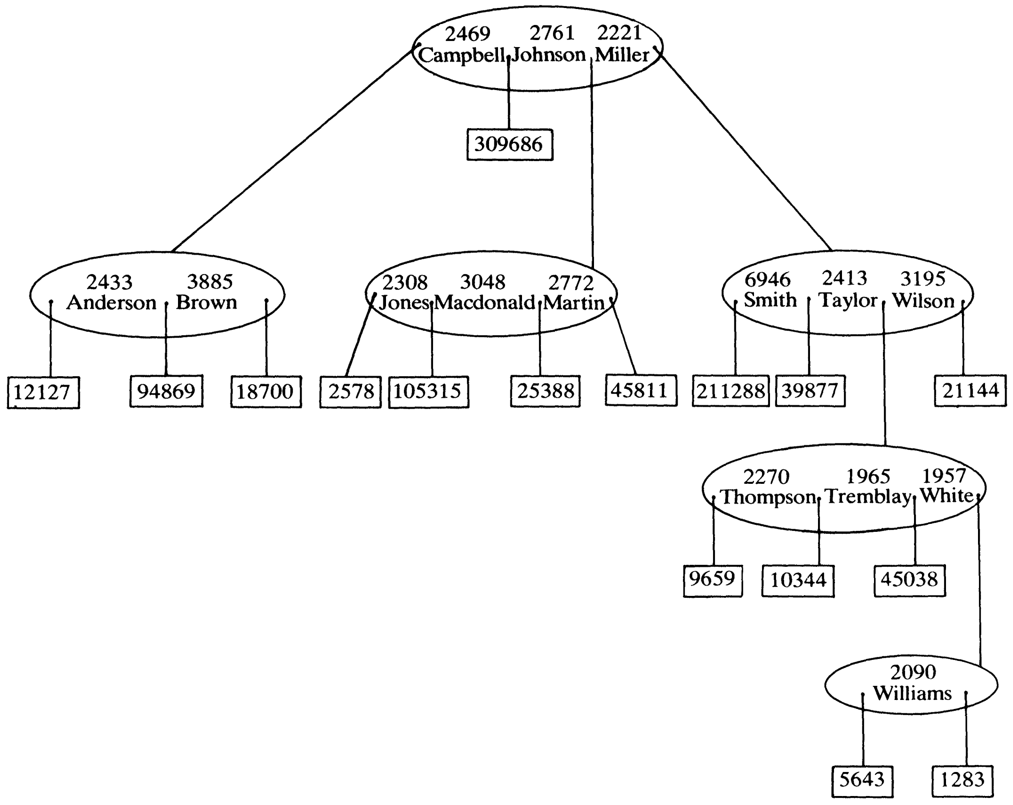

FIG. 1. *Optimal* 4-*way search tree on* 15 *most common names in Canadian census.*

times it occurred in a list of over a million names, while the numbers in the leaves (the q weights) represent those list members falling alphabetically between pairs from the top 15; thus there were 309,686 names (with repetitions) between "Campbell" and "Johnson", and 12, 127 preceeding "Anderson". Thus Level (p_i) is actually the level of K_i, while level (q_j) is the level of a node representing a missing-key range.

Two points about Fig. 1 should be noted. First, each page which has pages (i.e., nodes with names) as descendants must be filled, or the cost could be reduced by promoting names up to the unfilled pages. In other words, in an optimal tree, only leaf pages may be unfilled. Second, the search cost of a q-weight is not the level of the leaf labeled with that weight, but rather the level of the page which is the father of that weight; for example, the cost of the leaf weighted 309,686 is 309,686 times the level of the root, since the search for a name between "Campbell" and "Johnson" would stop at the root page. Thus one is subtracted from level (q_j) in the expresssion for the cost of the tree. The level of the root is taken to be one, since we want the level of a page to reflect its access cost.

For $m = 1$ (binary branching), a dynamic programming approach, employing the principle that subtrees of an optimal tree must also be optimal, yields an $O(N^2)$ construction algorithm [1]. Let $t(i, j)$ denote an optimal tree on weights $\langle q_i, p_{i+1}, q_{i+1}, \cdots, p_j, q_j\rangle$ (henceforth abbreviated $\langle i, j\rangle$), $c(i, j)$ be the cost of this tree, and $w(i, j)$ be its *weight*, $p_{i+1}, + \cdots + p_j + q_i + \cdots + q_j$. The idea is to compute $c(0, N)$, starting with

$$c(i, i) = 0, \qquad 0 \leqq i \leqq N,$$

and using

$$(1)\qquad c(i,j)=w(i,j)+\min_{i<h\leqq j}\{c(i,h-1)+c(h,j)\},\qquad 0\leqq i<j\leqq N.$$

Each time (1) is performed, the key corresponding to an h which gives a minimum is selected as $r(i,j)$, the root of $t(i,j)$. The computational cost is proportional to

$$\sum_{j=1}^{N}\sum_{i=0}^{j-1}(j-i),$$

which is roughly $N^3/6$, and N^2 storage locations are needed for the $c(i,j)$'s and $r(i,j)$'s. The running time can be further reduced by using Knuth's observation [1] that

$$r(i,j-1)\leqq r(i,j)\leqq r(i+1,j);$$

that is, one need not move left in the key set, looking for a new root when a key is added following the rightmost key in the tree, and vice-versa. This makes it possible to restrict the range of h in (1), thereby reducing the running time to

$$(2)\qquad \sum_{j=1}^{N}\sum_{i=0}^{j-1}(r(i+1,j)-r(i,j-1)+1)\ .$$

(To make this summation correct when $j-i=1$, we adopt the convention that $r(i,i)=0$, $0\leqq i\leqq N$; in fact, (1) is not needed when $j-i=1$, since by definition, $c(i,i+1)=w(i,i+1)$, and $r(i,i+1)=i+1$.) After cancelling terms, the sum reduces to

$$\frac{N(N+1)}{2}+\sum_{i=1}^{N}(r(i,N)-r(0,N-i))\approx N^2,$$

since $r(i,N)\leqq N$ and $r(0,N-i)>0$, so the time is $O(N^2)$.

The algorithm for optimal $m+1$-way trees, $m>1$, is basically an extension of the above technique. Let

$$i<r(i,j,1)<\cdots<r(i,j,m)\leqq j$$

be the (indices of the) keys on the root page of the optimal $m+1$-way tree $t(i,j)$ with cost $c(i,j)$ and weight $w(i,j)$. We have

$$c(i,i)=0,\qquad 0\leqq i\leqq N,$$
$$c(i,j)=w(i,j),\qquad 1\leqq j-i\leqq m,$$

and for $m<j-i\leqq N$, we must find an assignment for $r(i,j,k)$ such that

$$c(i,j)=w(i,j)+c(i,r(i,j,1)-1)+\sum_{k=1}^{m-1}c(r(i,j,k),r(i,j,k+1)-1)+c(r(i,j,m),j)$$

is a minimum.

The problem is carrying out this minimization. For each $r(i,j)$, there are $\binom{j-i}{m}$ choices for the root page, and when $j-i=d$, there are $N-d+1$ $c(i,j)$'s and $r(i,j)$'s to determine, namely for $(i,j)=\langle(0,d),(1,d+1),\cdots,(N-d,N)\rangle$. Since $j-i$ ranges from $m+1$ to N, a brute force approach which considers each possible candidate for the root page of $t(i,j)$ would take on the order of $\binom{N}{m}$ steps. This cost is too large to be practical, so a way of reducing the number of potential root pages per subtree is needed.

3. A dynamic programming solution. Let $\mathbf{T}(i, j, k)$, $1 \leqq k \leqq m$, denote the set of *optimal k-rooted trees* on weights $\langle i, j \rangle$, that is, optimal trees in which

i) there are *exactly* k keys on the root page,

and

ii) the $k+1$ subtrees of the root page are (optimal) $m+1$-way trees.

The basis for our construction algorithm is an observation similar to that made by Itai for code trees [4]. We note that the cost of a search tree, expressed iteratively as $\sum_i p_i \cdot \text{level}(p_i) + \sum_j q_j \cdot (\text{level}(q_j) - 1)$, can also be expressed recursively, in a form which makes it clear how to carry out minimization. For example, the cost of the tree in Fig. 1 can be written

Cost (2-rooted tree on ⟨"Anderson", · · · , "Martin"⟩)

+ Weight ("Miller") + Weight (Right Subtree of "Miller")

+ Cost (Right Subtree of "Miller").

Now for the tree to be optimal, the right subtree of "Miller" must be optimal, as, must the 2-rooted tree to the left; otherwise the total cost could be reduced by finding better trees on the same key sets. Moreover, the choice of "Miller" as the rightmost key on the root page must produce a cost no greater than any other ⟨optimal 2-rooted tree, rightmost root key, optimal 4-way subtree⟩ combination. Therefore, given all optimal 4-way and 2-rooted trees for proper subsequences of ⟨"Anderson", · · · , "Williams"⟩, the problem of finding the optimal 4-way tree on the whole sequence is reduced to that of choosing the rightmost key of the root page (i.e., that which gives the smallest cost when substituted into the above formula).

More generally, let $c(i, j, k)$ be the cost of $t(i, j, k) \in \mathbf{T}(i, j, k)$. Then for $j - i \geqq m$,

$$c(i, j, m) = \min_{i+m \leqq h \leqq j} \{c(i, h-1, m-1) + p_h + w(h, j) + c(h, j, m)\}.$$

In other words, $t(i, j, m)$ is the smallest cost concatenation of an optimal $m-1$-rooted tree on the left, together with a single root and its (right) $m+1$-way subtree. (The weight of the right subtree, $w(h, j)$, must be added in because the tree starts at level two, and therefore its stand-alone cost, $c(h, j, m)$, does not reflect its access cost as a subtree.) Similarly, $t(i, j, m-1)$ can be determined from optimal m-2-rooted and m-rooted subtrees, and in general, for $2 \leqq k \leqq m$, $j - i \geqq k$,

$$c(i, j, k) = \min_{i+k \leqq h \leqq j} \{c(i, h-1, k-1) + p_h + w(h, j) + c(h, j, m)\}. \tag{3}$$

(Note that $i + k$ is the lower bound, since the set of candidates for the rightmost root of a k-rooted tree starts at the kth key from the left.)

The computation of $c(i, j, 1)$ is slightly different since no concatenation of trees is involved; rather a single root is chosen to minimize the cost of left and right subtrees. We have, for $i < j$,

$$c(i, j, 1) = w(i, j) + \min_{i < h \leqq j} \{c(i, h-1, m) + c(h, j, m)\} \tag{4}$$

which is similar to (1), except that the subtrees are $m+1$-way, rather than binary.

(The equivalence of the recursive formulation of cost, given by (3) and (4), with the iterative expression $\sum_i p_i \cdot \text{level}(p_i) \cdots$ etc.) is easy to show. For trees with maximum height one, (4) is trivially correct, and (3) is established by induction on k. Then, assuming the equivalence for trees with maximum height less than h, (4) is proved for trees with maximum height h, and (3) follows, again by induction on k.)

Assuming then, that $c(i, j, k)$ has been computed for $1 \leqq k \leqq m$ and $j-i<d$, we can compute $c(i, j, k)$ for $1 \leqq k \leqq m$ and $j-i=d$, starting with (4) to get $c(i, j, 1)$, and followed by successive applications of (3) with $k=2, 3, \cdots, m$. (In the final step when $c(0, N, m)$ is to be determined, it is only necessary to use (3) once, since at this point optimal trees with fewer than m keys on the root page are not needed; however the savings are negligible.) The timing analysis is essentially the same as for the binary algorithm: the cost of (4) is proportional to

$$\sum_{j=1}^{N} \sum_{i=0}^{j-1} (j-i) \approx N^3/6,$$

while for $2 \leqq k \leqq m$, (3) takes

$$\sum_{j=k}^{N} \sum_{i=0}^{j-k} (j-i-k+1) \approx \frac{(N-k+1)^3}{6} \text{ steps.}$$

The total cost is therefore approximately

$$\sum_{k=1}^{m} \frac{(N-k+1)^3}{6},$$

which is $O(N^3 m)$.

For $2 \leqq k \leqq m$, let $r(i, j, k)$ be the largest value of h that gives a minimum in (3), and define $r(i, j, 1)$ similarly for (4); $r(i, j, k)$ is simply the largest possible key on a root page of $T(i, j, k)$. In §§ 5 and 6, we show two facts about $r(i, j, k)$:

1) During construction of an optimal k-rooted tree, the rightmost key in the root can move left when a key is added at the right (alphabetically high) end of the key set. In particular, $r(i, j, k)$ may be less than $r(i, j-1, k)$ (§ 5).

2) If the missing-key weights (q's) are all zero, then the following *monotonicity property* holds (Theorem 2, § 6):

$$r(i, j-1, k) \leqq r(i, j, k) \leqq r(i+1, j, k).$$

Point (1) means that, in contrast with the situation for binary trees, $r(i, j, k)$ can lie outside $[r(i, j-1, k), r(i+1, j, k)]$ and therefore, when q's are present, the speed up in running time from N^3 to N^2 cannot be applied. When the q's are absent, (2) restricts the search for the minimum in (3) and (4), making the total cost proportional to

$$\sum_{k=1}^{m} \sum_{j=k}^{N} \sum_{i=0}^{j-k} (r(i+1, j, k) - r(i, j-1, k) + 1).$$

(Again, it is convenient to let $r(i, j, k) = 1$ if $j-i<k$.) For each k, the inner double sum is equivalent to (2) above (in fact, slightly smaller when $k>1$), which is bounded by N^2; hence the running time is $O(N^2 m)$.

The storage requirement is $O(N^2 m)$ locations, since $r(i, j, k)$ and $c(i, j, k)$ are N by N by m arrays. To see how the desired tree, $t(0, N, m)$, is reconstructed from the information in r, note that the keys on the root page of $t(0, N, m)$ are given by the following sequence:

$$r_1 = r(0, N, m)$$

$$r_k = r(0, r_{k-1}-1, m-k+1), \qquad \text{for } 2 \leqq k \leqq m.$$

In other words, the rightmost key r_1 is $r(0, N, m)$, the key second from the right is the rightmost root key for the weights $\langle 0, r_1-1\rangle$ and so on. The root pages of the subtrees of the root page, and hence ultimately the whole tree, are similarly determined.

An algorithm which uses (3) and (4) to compute optimal $m+1$-way search trees on weights $\langle q_0, p_1, \cdots, p_N, q_N\rangle$ is presented in [5].

4. A refinement. The time and space costs of the above algorithm can be reduced using a technique employed by Itai for the construction of optimal multi-way code trees [4, § 7]. He observed that an optimal m-way code tree is the concatenation of an optimal collection of s m-way trees, together with an optimal collection of m-s trees. Here we cannot concatenate trees in the same way code trees are combined (there would be one subtree too many), but we can join them with a middle root; for example the tree of Fig. 1 can be regarded as the join of two optimal 1-rooted trees (with roots "Campbell", "Miller"), together with the key "Johnson". Thus, for $u+v=k-1$, $k>2$,

$$c(i,j,k)=\min_{i+u<h\leqq j-v}\{c(i,h-1,u)+p_h+c(h,j,v)\}. \tag{5}$$

This makes it possible to determine $c(i,j,m)$ without computing all the intermediate costs, $c(i,j,1), c(i,j,2), \cdots, c(i,j,m-1)$. For example, suppose $c(i,j,k)$ are available for $k=\langle 1,3,7,11,12\rangle$ and $j-i<d$; then for $j-i=d$, $c(i,j,k)$ can be derived for the same set of k's by the following steps:

(1) use (4) to get $c(i,j,1)$,
(2) use (5) with $u=v=1$ to get $c(i,j,3)$,
(3) use (5) with $u=v=3$ to get $c(i,j,7)$,
(4) use (5) with $u=7$, $v=3$ to get $c(i,j,11)$
(5) use (3) with $k=12$ to get $c(i,j,12)$.

More generally, let $s_0, s_1, \cdots, s_l$ be any sequence which satisfies

$$s_0=0,\ s_l=m, \tag{6}$$

and

$$s_h=1+s_i+s_j,\qquad 0\leqq i\leqq j\leqq h\leqq l. \tag{7}$$

For $1\leqq h\leqq l$, let left (h) and right (h) be, respectively, the i and j for which (7) holds, that is, $s_h=s_{\text{left}(h)}+s_{\text{right}(h)}+1$. Finally, suppose $c(i,j,s_h)$ has been computed for $h\in\langle 1,2,\cdots,l\rangle$ and $j-i<d$. Then the following rule is used to decide which of (3), (4) or (5) to apply when computing $c(i,j,s_h)$ for $j-i=d$ and $h=1$ to l:

```
**if** left $(h)=0$ **then** use (4)
**else** if right $(h)=0$ **then** use (3) with $k=s(h)$
     **else** use (5) with $u=$ left $(h)$ and $v=$ right $(h)$.
```

Sequences for which (6) and (7) hold are closely related to addition chains. An *addition chain of length l* for $n>1$ is a sequence of integers $1=a_0, a_1, \cdots, a_l=n$, such that $a_h=a_i+a_j$, $0\leqq i\leqq j<h\leqq l$. Given an addition chain for $n=m+1$, note that the sequence $\langle a_0-1, a_1-1, \cdots, a_l-1\rangle$ satisfies (6) and (7), since

$$a_0-1=0,\qquad a_l-1=m,$$

and

$$a_h-1=(a_i+a_j)-1=1+(a_i-1)+(a_j-1),\qquad 1\leqq h\leqq l.$$

Let $l(n)$ be the length of the shortest possible addition chain for $n>1$. Given such a chain, it is possible to construct an optimal $m+1$-way tree in time $O(N^3 l(m+1))$. As it turns out, neither a minimal chain nor $l(n)$ are easy to compute efficiently for n in general [2, § 4.6.3]; however the *binary* addition chain for n is easily derived from the binary representation of that number, and has at most $2\lfloor \log n \rfloor$ terms ($\langle 1, 2, \cdots, 2^{\lfloor \log n \rfloor}\rangle$ plus a term for each "one" bit except the high order one). Thus the sequence of k's for which $c(i,j,k)$ must be computed need be no longer than $2\lfloor \log m+1 \rfloor$, which makes possible an $O(N^3 \log m)$ construction algorithm (or $O(N^2 \log n)$, when the q's are absent).

5. Failure of monotonicity property in the general case. In this section we show that, during construction of an optimal multi-way search tree, the rightmost key in the root can move left when a key is added at the right. Consider a set of 8 keys, $\{A, B, \cdots, H\}$, each weighted one, together with 9 missing-key weights q, all equal to one except for $q(8)$ (between G and H), which is 4. For convenience, these weights are unnormalized; to get the actual probabilities, it is necessary to divide each weight by the total. Thus the probability of a search for A is $\frac{1}{20}$.

Fig. 2(b) shows a 3-way ($m=2$) tree which is optimal for these weights. The rightmost key in the root is F; it is also $r(0,8,2)$, the largest possible rightmost root, since no other 3-way tree costs as little for this data. Now the algorithm outlined in § 3 requires an optimal tree on $\langle A, G\rangle$ before it can determine one on $\langle A, H\rangle$, and such a tree is illustrated in Fig. 2(a). This tree is also uniquely optimal for the weights given, which makes G, the rightmost key in the root, equal to $r(0,7,2)$. Thus the rightmost root key shifts left from G, on keys $\langle A, G\rangle$, to F, on keys $\langle A, H\rangle$, and in particular, $r(0,8,2) \leqq r(0,7,2)$, which shows that $r(i,j,k)$ may lie outside the interval $[r(i,j-1,k), r(i+1,j,k)]$. (The number of keys in the root of the tree in Fig. 2 is not critical to the example, and thus the result is true for optimal k-rooted trees as defined at the beginning of § 3.)

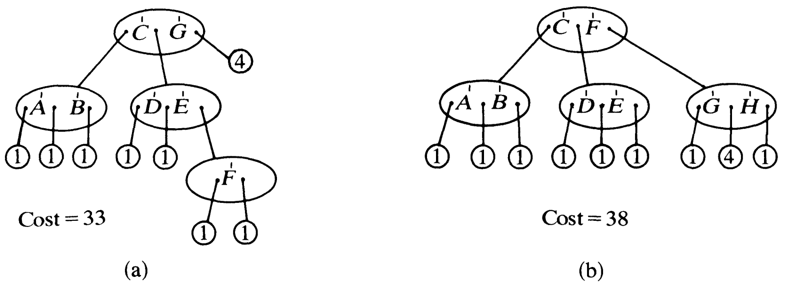

FIG. 2. *Optimal trees on $\langle A, G\rangle$ and $\langle A, H\rangle$.*

The example just presented affects an assumption made by Itai regarding the time needed to construct optimal multi-way code trees [4, § 7]. To see how, observe that if the internal weights are ignored, the trees in Fig. 2 are 3-way code trees. The one in (a) is the join of two 3-way trees together with the leaf weighted "4", while (b) is formed by concatenating two 3-way trees with a rightmost tree consisting of weights $\langle 1, 4, 1\rangle$. The last weight before the rightmost tree in the least-cost concatenation is called the *breakpoint*. For code trees, Itai assumed the equivalent of the monotonicity property, namely that when a weight is added at the right, a breakpoint greater than or equal to the previous one can always be found. But Fig. 2 shows that this need not be true, since the breakpoint in (a) is the 7th leaf weight (following F), while it is the 6th (after E) in (b). Hence the dynamic programming construction procedure which he outlined is $O(N^3 \log m)$, not $O(N^2 \log m)$ as claimed.

In [7] it is shown that, during construction, the rightmost root in a multi-way search tree or code tree can shift the maximum possible number of keys away from the added key. Thus the dependence on key set size of dynamic programming methods for multi-way tree construction presented so far is $O(N^3)$. However our counterexamples to the monotonicity property rely on the existence of nonzero leaf (q) weights; in the next section we show that when these weights are absent, the addition of a key at the right cannot force the rightmost key in the root to move left.

6. Proof of monotonicity property when q's absent. The purpose of this section is to show that when $q_h = 0, 0 \leqq h \leqq N$, $r(i, j, k)$, the largest key on the root page of all trees in $\mathbf{T}(i, j, k)$, satisfies

$$r(i, j-1, k) \leqq r(i, j, k) \leqq r(i+1, j, k), \qquad k < j-i \leqq N, \quad 1 \leqq k \leqq m.$$

The proof is a generalization of the one for binary trees outlined in [3, § 6.2.2]; however it should be noted that for binary trees, the result holds whether the q's are zero or not. Note also that to stay consistent with the general case, we will use $\langle i, j\rangle$ to denote the weights $\langle p_{i+1}, \cdots, p_j\rangle$, even though the absence of the q's makes $\langle i+1, j\rangle$ more natural. The following definitions are needed:

DEFINITION 1. Define $A \lessdot B$ to hold between nonempty sets of integers A, B if $a \in A$, $b \in B$ and $b < a \Rightarrow a \in B$ and $b \in A$. Informally, if A, regarded as a sequence, is "left" of B in terms of position, then $A \lessdot B$ holds if "overlapping" members are common to both sets. The relation has the following property, which we state without proof:

LEMMA 1. *$\lessdot$ is transitive.*

DEFINITION 2. $R(i, j, k)$, $1 \leqq k \leqq m$, is the set of rightmost keys (henceforth called *rightmost roots*) on the root pages of trees in $\mathbf{T}(i, j, k)$; in other words, it is the set of $i+k \leqq h \leqq j$ for which (3) is minimized if $k > 1$, or (4) is minimized if $k = 1$.

DEFINITION 3. $L(i, j, k)$ $1 \leqq k \leqq m$, is the set of leftmost keys (*leftmost roots*) on the root pages of trees in $\mathbf{T}(i, j, k)$. We have $L(i, j, 1) = R(i, j, 1)$ for $1 \leqq j-i \leqq N$, and for $2 \leqq k \leqq m$, $k < j-i \leqq N$,

$$L(i, j, k) = \{i < h \leqq j-k+1 \mid c(i, h-1, m) + w(i, h-1) + p_h + c(h, j, k-1) \text{ is a minimum}\}$$

Proof outline. We first show that except for possible addition of the new key, the set of rightmost roots does not change when the existing key set is augmented at its right (alphabetically high) end by a key with weight zero (Lemma 2). The main theorem (Theorem 1) is proven in two parts. In Lemma 4, weights $\langle i, \cdots, j-1\rangle$ are fixed while p_j is increased from x to y; in this case it is shown that there will always be a rightmost root $\geqq$ any key which is a rightmost root when $p_j = x$. Together with Lemma 2, this shows that $R(i, j-1, k) \lessdot R(i, j, k)$, or, informally, that we need never move left in the key set to look for a new rightmost root when a key is added at the right. The equivalent result for leftmost roots is then used to show that the rightmost root does not have to move right when a key is added at the left, thus completing the proof. Finally, Theorem 1 is used to prove Theorem 2, the desired result.

LEMMA 2. *Let $q_h = 0$, $0 \leqq h \leqq N$. If $p_j = 0$, then $R(i, j-1, k) = R(i, j, k) - \{j\}$, $k < j-i \leqq N$.*

Proof. Since $p_j = 0$, adding it to a tree in $\mathbf{T}(i, j-1, k)$ cannot increase the cost, but cannot lead to a better rearrangement either, or removing it would then leave a better tree than the original, which was optimal. Therefore any tree in $\mathbf{T}(i, j-1, k)$ can be converted to one in $\mathbf{T}(i, j, k)$ by straight insertion of p_j, which does not change the rightmost key in the root. A similar argument shows that any tree in $\mathbf{T}(i, j, k)$ in which p_j is not the rightmost root must also be in $\mathbf{T}(i, j-1, k)$ when p_j is removed.

The main theorem to be proved is

THEOREM 1. *If* $q_h = 0$, $0 < h \leqq N$, *then*

$$R(i, j-1, k) \lessdot R(i, j, k) \lessdot R(i+1, j, k),$$

and

$$L(i, j-1, k) \lessdot L(i, j, k) \lessdot L(i+1, j, k),$$

for

$$k < j-i \leqq N, 1 \leqq k \leqq m.$$

Proof. By induction on $j - i > k$.

Basis. If $j-i=k+1$, then $R(i, j-1, k)=\{j-1\}$, $R(i+1, j, k)=\{j\}$, and $R(i, j, k) \subseteq \{j-1, j\}$. Similarly $L(i, j-1, k)=\{i+1\}$, $L(i+1, j, k)=\{i+2\}$, and $L(i, j, k) \subseteq \{i+1, i+2\}$; hence the theorem holds trivially.

Induction. Assuming that the theorem holds for $k < j-i < d < N$, $1 \leqq k \leqq m$, we will show that it remains true for $j - i = d$.

LEMMA 3. *If* $u - w < d$, *and* $u < v < w$, *then* $R(u, w, k) \lessdot R(v, w, k)$, *and* $L(u, v, k) \lessdot L\{u, w, k)$.

Proof. By the induction hypothesis and Lemma 1.

LEMMA 4. *Let* $R_x(i, j, k)$ *denote the set of rightmost roots when* $p_j = x \geqq 0$, *and* $R_y(i, j, k)$ *be the corresponding set when* $p_j = y > x$, *while weights* $\langle i, \cdots, j-1\rangle$ *remain fixed; then* $R_x(i, j, k) \lessdot R_y(i, j, k)$.

Proof. Let $r \in R_x(i, j, k)$, T_x be an optimal k-rooted tree with rightmost root r and $p_j = x$, and l_x be the level of p_j in T_x; similarly for $s \in R_y(i, j, k)$, T_y, y and l_y. Suppose $s < r$. The lemma will be proved if we can modify T_x without changing its rightmost root so that it is optimal for y, and similarly for T_y with respect to x.

The cost of T_x can be expressed as a linear function of p_j, mamely $a + l_x \bullet p_j$, and similarly, cost $(T_y) = b + l_y \bullet p_j$. Of the nine possible outcomes for a compared to b, l_x compared to l_y, all contradict the definitions of T_x and T_y except $a = b$, $l_x = l_y$, in which case the Lemma is proved, and $a < b$, $l_x > l_y$.

When $a < b$ and $l_x > l_y$, cost (T_x) meets cost (T_y) at $z > 0$, and Fig. 3 shows the two basic situations possible at this point. In 3(a), T_x is optimal for $p_j \leqq z$, while T_y is optimal for $p_j \geqq z$. In 3(b), neither is optimal at z, because for $x < p_j < y$, there are intermediate trees (two in the figure) which are optimal and cost less than either T_x or T_y. We consider the case of (a) first.

In tree T_x, let $r_1, r_2, \cdots, r_{l_x}$ be the sequence of rightmost roots along the path from the root page down to the page containing p_j, and let $s_1, \cdots, s_{l_y}$ be the corresponding sequence for T_y. Now $s = s_1 < r_1 = r$ by assumption, and $r_{l_y} < s_{l_y} = j$, since $l_x > l_y$, and $r_{l_x} = j$, the index of the last key; thus there must be a level h such that $r_h > s_h$ and $r_{h+1} < s_{h+1}$. Both T_x and T_y are optimal at z, so $r_{h+1} \in R_z(r_h, j, m)$ and $s_{h+1} \in R_z(s_h, j, m)$. Since $s_h < r_h < j$, and $j - s_h < j - i = d$, we can apply Lemma 3 to get $R_z(s_h, j, m) \lessdot R_z(r_h, j, m)$. But $r_{h+1} < s_{h+1}$, so by definition of $\lessdot$, it follows that $s_{h+1} \in R_z(r_h), j, m)$ and $r_{h+1} \in R_z(s_h, j, m)$.

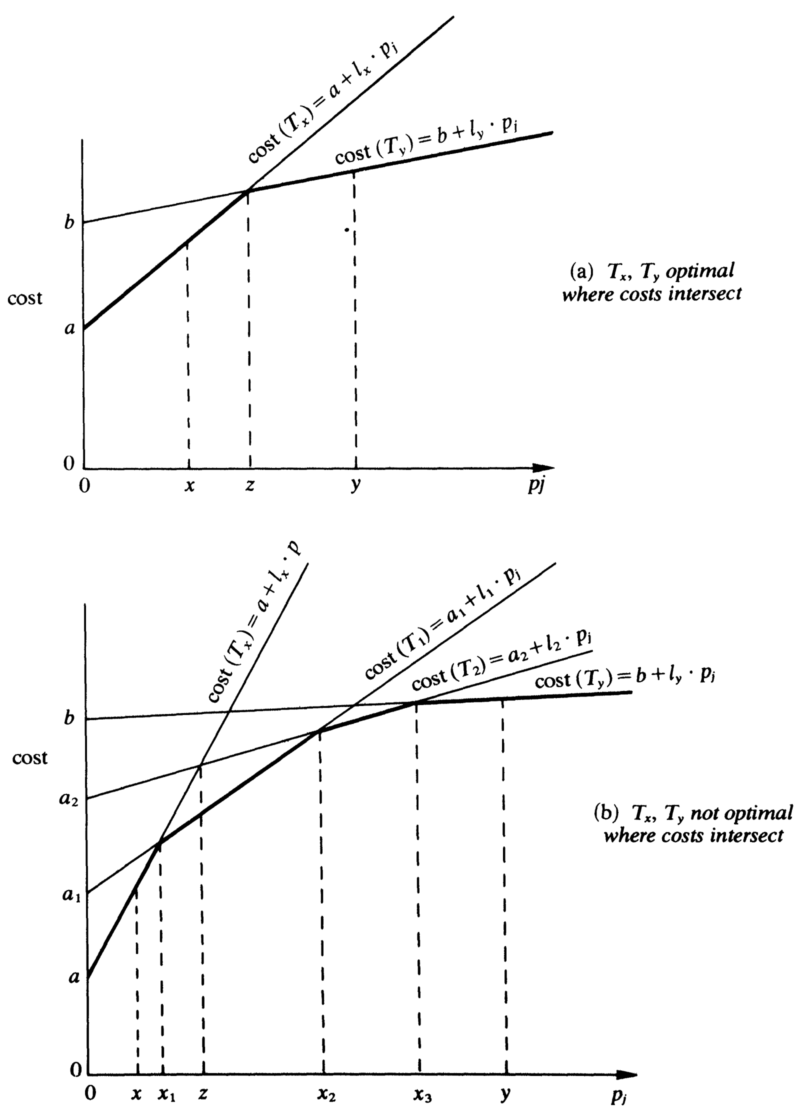

FIG. 3. *Cost of T_x, T_y as functions of p_j.*

In other words, there is a rearrangement of T_x, call it T'_x, which is optimal for $p_j = z$, and has s_{h+1} (at level $h+1$) as the rightmost root of the subtree for keys to the right of r_h. Now the subtree to the right of s_{h+1}, containing p_j, can remain unchanged from T_y, since T_y is also optimal at z, and this subtree starts at the same level as it did in T_y. Thus the level of p_j in T'_x is l_y, and cost $(T'_x) = a' + l_y \bullet p_j$. But cost $(Ty) = b + l_y \bullet p_j$ and both agree at z; therefore $a' = b$, and T'_x, with rightmost root r, is optimal for y. Similarly we can rearrange T_y, without changing its rightmost root s, to T'_y, which has the same cost at T_x. Thus $r \in R_y(i, j, k)$ and $s \in R_x(i, j, k)$.

If cost (T_x) meets cost (T_y) at a point where neither is optimal, then there is a sequence of trees $T_x = T_0, T_1, \cdots, T_l = T_y$, and values $x = x_1, \cdots, x_{l+1} = y$, such that T_I is optimal on $[x_I, x_{I+1}]$, $0 \leqq I \leqq l$. (Fig. 3(b) illustrates the situation for $l = 3$.) Let $rt(I)$ be the rightmost root of T_I; for the sequence of roots $\langle rt(0), \cdots, rt(l)\rangle$, we have $r = rt(0)$ and $rt(l) = s$. If $s < r$, an induction argument shows that the sequence can be transformed so that s is the first term (i.e., $s \in R_x(i, j, k)$), and r is the last ($r \in R_y(i, j, k)$). The basis for sequences of length two was established above, and the induction step is straightforward. $\square$

From Lemma 2, $R(i, j-1, k) = R_x(i, j, k) - \{j\}$ if $x = 0$, and from Lemma 4, $R_x(i, j, k) \prec R_y(i, j, k)$ for $x < y$; hence $R(i, j-1, k) \prec R(i, j, k)$, as desired. By symmetry, the equivalent of Lemmas 2 and 4 for leftmost roots can be used to show that $L(i, j, k) \prec L(i+1, j, k)$, $1 \leqq k \leqq m$. Thus it remains to show

$$R(i, j, k) \prec R(i+1, j, k) \quad \text{and} \quad L(i, j-1, k) \prec L(i, j, k).$$

For $k = 1$ these are immediate, since $R(i, j, 1) = L(i, j, 1)$. We will show $R(i, j, k) \prec R(i+1, j, k)$ for $k > 1$.

Let $T1 \in \mathbf{T}(i, j, k)$ with rightmost root $r1$ and leftmost root $l1$, and similarly for $T2 \in \mathbf{T}(i+1, j, k)$, $r2$ and $l2$. Assume $r2 < r1$; $l1$ may be equal to, less than, or greater than $l2$, and Fig. 4 illustrates the three cases.

$$\begin{array}{ll}
\langle i+1, \ldots, l2, \ldots\ldots\ldots\ldots, r2, \ldots\ldots\ldots, j\rangle & T2 \\
\langle i, i+1, \ldots, l1, \ldots\ldots\ldots\ldots\ldots\ldots, r1, \ldots, j\rangle & T1, l1 = l2 \\
\langle i, i+1, \ldots\ldots\ldots, l1, \ldots\ldots\ldots\ldots\ldots, r1, \ldots, j\rangle & T1, l1 > l2 \\
\langle i, i+1, \ldots, l1, \ldots\ldots\ldots\ldots\ldots\ldots\ldots, r1, \ldots, j\rangle & T1, l1 < l2
\end{array}$$

FIG. 4. *Proving* $R(i, j, k) \prec R(i+1, j, k)$.

If $l1 = l2$, then since $r2$ and $r1$ are both in $R(l1, j, k-1)$, the subtrees on $\langle l1, j\rangle$ in $T1$ and $T2$ can be switched; thus $r2 \in R(i, j, k)$ and $r1 \in R(i+1, j, k)$. Suppose $l1 > l2$. Since from above, $L(i, j, k) \prec L(i+1, j, k)$, there is a re-arrangement of $T2$ which is optimal but has $l1$ as its leftmost root. In this tree, the keys to the right of $l1$ can be arranged as they are in $T1$, giving $r1$ as the rightmost root. Thus $r1 \in R(i+1, j, k)$ and a similar re-arrangement of $T1$ gives $r2 \in R(i, j, k)$. Finally, suppose $l1 < l2$; then $R(l1, j, k-1) \prec R(l2, j, k-1)$ by Lemma 3, and since $r2 < r1$, $r2$ must also be in $R(l1, j, k-1)$ and $r1$ in $R(l2, j, k-1)$ by definition of $\prec$. Therefore the $k-1$ rooted tree to the right of $l1$, with rightmost root $r1$, can be replaced by one with the same cost and rightmost root $r2$, i.e., $r2 \in R(i, j, k)$; a similar replacement in $T2$ gives $r1 \in R(i+1, j, k)$.

A symmetric argument shows that $L(i, j-1, k) \prec L(i, j, k)$, $k > 1$, and thus completes the proof of the theorem. $\square$

The needed result is expressed by:

THEOREM 2. *If* $q_h = 0$, $0 \leqq h \leqq N$, *then* $r(i, j, k)$, *the largest possible rightmost root, satisfies*

$$r(i, j-1, k) \leqq r(i, j, k) \leqq r(i+1, j, k), \qquad k < j - i \leqq N, \quad 1 \leqq k \leqq m.$$

Proof. By induction on $j - i > k$.

If $j - i = k + 1$, then $r(i, j-1, k) = j - 1$, $r(i+1, j, k) = j$, and the only two possibilities for $r(i, j, k)$ are $j - 1$ or j, so the theorem holds. Assuming that the theorem holds for $k < j - i < d < N$, $1 \leqq k \leqq m$, we will show that it remains true for $j - i = d$.

First, observe that from the induction hypothesis, $r(i, j-1, k) \leqq r(i+1, j-1, k) < r(i+1, j, k)$. To show the left inequality, suppose $r \in R(i, j, k) < r(i, j-1, k)$. Then since $r(i, j-1, k) \in R(i, j-1, k)$ and $R(i, j-1, k) \prec R(i, j, k)$ by Theorem 1, $r(i, j-1, k) \in R(i, j, k)$, that is, the rightmost possible root in $T(i, j, k)$ is $\geqq r(i, j-1, k)$.

To show $r(i, j, k) \leqq r(i+1, j, k)$, suppose $r \in R(i, j, k) > r(i+1, j, k) \in R(i+1, j, k)$. By Theorem 1 we have $R(i, j, k) \leqslant R(i+1, j, k)$, so $r \in R(i+1, j, k)$. But r is greater than $r(i+1, j, k)$, which by definition is the largest rightmost root, a contradiction. Therefore any member of $R(i, j, k)$, and in particular, $r(i, j, k)$, is $\leqq r(i+1, j, k)$. $\square$

7. Conclusion. The algorithm outlined in § 3 employs "double" dynamic programming—building optimal trees on increasingly larger key sets, and at the same time building trees with successively more keys on the root page. The technique is powerful: even when the monotonicity property does not hold, it reduces the number of possible root pages from $\binom{N}{m}$ to N, which is independent of page capacity. In [5], the method is used to construct optimal weighted B-trees. Since these have the property that a page may have anywhere from the minimum number of keys allowable to the maximum, the basic $O(N^3 m)$ algorithm is not completely supplanted by the refinement of § 4.

Acknowledgments. I would like to thank my advisor, Prof. John D. Lipson, for advice and encouragement.

REFERENCES

[1] D. E. Knuth, *Optimum binary search trees*, Acta Informat., 1 (1971), pp. 14–25; *Errata*, 1 (1972), p. 270.

[2] ———, *The Art of Computer Programming, vol.* 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA., 1969, Section 4.6.3.

[3] ———, *The Art of Computer Programming, vol.* 3, *Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

[4] Alon Itai, *Optimal alphabetic trees*, this Journal, 5 (1976), pp. 9–18.

[5] L. Gotlieb, *Optimal Multi-way Search Trees*, TR 128/78, Department of Computer Science, University of Toronto, 1978.

[6] V. K. Vaishnavi, H. P. Kriegel and D. Wood, *Optimal Multi-way Search Trees*, TR 78-CS-13, Dept. of Applied Mathematics, McMaster University, Hamilton, Ontario, 1978.

[7] L. Gotlieb and D. Wood, *The construction of multiway search trees and the monotonicity principle*, Internat. J. Comput. Math., to appear.