Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting*

THOMAS N. HIBBARD

System Development Corporation, Santa Monica, California

INTRODUCTION

This paper introduces an abstract entity, the binary search tree, and exhibits some of its properties. The properties exhibited are relevant to processes occurring in stored program computers—in particular, to search processes. The discussion of this relevance is deferred until Section 2.

Section 1 constitutes the body of the paper. Section 1.1 consists of some mathematical formulations which arise in a natural way from the somewhat less formal considerations of Section 2.1. The main results are Theorem 1 (Section 1.2) and Theorem 2 (Section 1.3).

The initial motivation of the paper was an actual computer programming problem. This problem was the need for a list which could be searched efficiently and also changed efficiently. Section 2.1 contains a description of this problem and explains the relevance to its solution of the results of Section 1.

Section 2.2 contains an application to sorting.

The reader who is interested in the programming applications of the results but not in their mathematical content can profit by reading Section 2 and making only those few references to Section 1 which he finds necessary.

1. COMBINATORIAL PROPERTIES OF BINARY SEARCH TREES

1.1 *Preliminary Definitions*

The central notion of this paper is that of a certain type of directed graph undergoing "random" operations. The present section is given to defining the graph and certain quantities associated with the graph. "Expected" values of these quantities will be defined combinatorially in Section 1.2; these "expected" values will then be calculated assuming "random insertions" (Section 1.2) and "random deletions" (Section 1.3).

DEFINITION. A *binary search tree* is a directed graph[1] having the following properties.

* Received March, 1961; revised July, 1961.

[1] A *directed graph* is a set of points together with a set of ordered pairs of these points. The points are called *nodes* and the pairs are called *links*. A *path* in a directed graph is a sequence $(p_i, p_2, \cdots, p_n)$ of nodes such that, for $1 \leqq i < n$, (p_i, p_{i+1}) is a link belonging to the directed graph. (Thus each link is a path of length 2.) Any path $(p, \cdots, q)$ is said to begin with p and end with q.

(i) There is one and only one node, called the *root,* such that for any node p there exists one and only one path which begins with the root and ends with p.
(ii) For each node p, the number of links beginning with p is either two or zero. If the number is two, then p is said to be a *proper* node. If the number is zero, then p is said to be a *blank* node.
(iii) The set of links is partitioned into two sets L and R. Each link belonging to L is called a *left* link. Each link belonging to R is called a *right* link.
(iv) For each proper node p, there is exactly one left link beginning with p and exactly one right link beginning with p.

For brevity, a *search tree* will here be defined as a binary search tree.

The following notational devices will be employed for search trees.

The *length* of a path is the number of proper nodes in the path.

To each node of any search tree we attach a label p_σ, where σ is a sequence of 1's and 0's uniquely defined by the following. Let the root be called $p_\wedge$ (where $\wedge$ denotes the null sequence).[2] If, for any σ, p_σ is a proper node, then $p_{\sigma 0}$ and $p_{\sigma 1}$ are such that $(p_\sigma, p_{\sigma 0})$ is a left link and $(p_\sigma, p_{\sigma 1})$ is a right link.

For a search tree T, the *search subtree* T_σ is defined for each σ as follows. Every node of T_σ belongs to T. A node q of T belongs to T_σ if and only if q belongs to a path in T beginning with p_σ. Every left (right) link of T_σ is a left (right) link of T. A left (right) link (q, r) of T is a left (right) link of T_σ if and only if both q and r belong to T_σ. (Hence, if σ is such that T contains no node p_σ, then T_σ is defined to be empty.)

Throughout this paper, the symbols σ, ρ, and τ will denote sequences of 1's and 0's. Further, any symbol having σ, ρ, or τ as a subscript is to be understood to be a function of σ, ρ, or τ.

In the following definitions, let T be a search tree of n proper nodes and let S be a set of n numbers.

The *list function* f_{ST} is a one-one mapping of the proper nodes of T onto the elements of S, with the following property. If q is a proper node of $T_{\sigma 0}$ then $f_{ST}(q) < f_{ST}(p_\sigma)$, and if q is a proper node of $T_{\sigma 1}$ then $f_{ST}(q) > f_{ST}(p_\sigma)$. (Thus, given any two of (T, S, f_{ST}), the third is uniquely defined.) When S and T are understood, f will be used to denote f_{ST}.

For each σ, the subset S_σ of S is defined as follows. y is in S_σ if and only if there exists a node q of T_σ such that $y = f_{ST}(q)$.

A path in T_σ which begins with p_σ is said to be a *search path* in T_σ. If a search path ends with a proper node then it is said to be an *internal* search path. If a search path ends with a blank node then it is said to be an *open* search path. (Note that if any search tree T has n proper nodes then there exist $n+1$ open search paths in T, one for each blank node. This is easily shown by an induction on n.) The lengths of the search paths in $T_\wedge$ are the subject of the next two sections.

[2] The expressions $p_\wedge$ and p are equivalent. While the use of $\wedge$ is never essential, it sometimes increases clarity. Both notations will be used here.

1.2 *Sequence Binary Search Trees*

We will now formalize, in combinatorial terms, the notion of a "randomly constructed" search tree and of the "expected length" of a search path in such a search tree.

Notation. Throughout this and the following sections, the following notation will be applicable to any sequence R. For each σ, a unique subsequence R_σ of R is defined as follows. For each non-null R_σ, let x_σ be the first term of R_σ. If σ is null then $R_\sigma = R$. Given R_σ for any σ, then $R_{\sigma 0}$ and $R_{\sigma 1}$ are subsequences of R_σ and are obtained as follows. Each term of R_σ, with the exception of x_σ, belongs either to $R_{\sigma 0}$ or to $R_{\sigma 1}$. Every term of $R_{\sigma 0}$ is less than x_σ, and every term of $R_{\sigma 1}$ is greater than x_σ.

Definition. Let R be a sequence of distinct numbers. Let S be the set of numbers in R. For each non-null R_σ, let x_σ denote the first term of R_σ. The *sequence* (binary) *search tree* for R is the search tree T such that $f_{ST}(p_\sigma) = x_\sigma$.

Observe that S_σ is the set of numbers in R_σ, and that T_σ is the sequence search tree for R_σ.

Now the notion of a "randomly constructed" search tree is that of the sequence search tree for a "random" sequence. The notion will be formalized combinatorially by means of the family of sequence search trees associated with the $n!$ permutations of a given set of n numbers.

Notation. For each sequence R the sequence of ranks for R, written $r(R)$, is defined as follows. Let $R = y_1, y_2, \cdots, y_n$. For each i, $1 \leqq i \leqq n$, let r_i be the number of integers k, $1 \leqq k \leqq n$, such that $y_k \leqq y_i$. Now $r(R) = r_1, r_2, \cdots, r_n$. Also, r_i is said to be the *rank* of y_i in R.

Clearly, for any two sequences P and Q such that $r(P) = r(Q)$, P and Q have the same sequence search tree.

For a search tree T having n proper nodes, let $\{s_i \mid i = 1, 2, \cdots, n+1\}$ be the set of $n+1$ distinct open search paths in T. Let l_i denote the length of s_i. The function $l(T)$ is now defined by

$$l(T) = \sum_{i=1}^{n+1} l_i .$$

Next let $\{s_i' \mid i = 1, 2, \cdots, n\}$ be the set of n distinct internal search paths in T, and let l_i' be the length of s_i'. Let $l'(T)$ be defined by

$$l'(T) = \sum_{i=1}^{n} l_i'.$$

For a set S of n numbers, let $\Phi = \{R^i \mid i = 1, 2, \cdots, n!\}$ be the set of $n!$ sequences such that each sequence has length n and contains every member of S. For each R^i in Φ, let T^i be the sequence search tree for R^i. Let

$$U(S) = \sum_{i=1}^{n!} l(T^i);$$

$$U'(S) = \sum_{i=1}^{n!} l'(T^i).$$

Since the sequence search tree for the sequence of ranks $r(R^i)$ is identical to the sequence search tree for R^i, it is clear that $U(S)$ and $U'(S)$ depend only on n. Hence, let $S = \{1, 2, \cdots, n\}$ and let

$$V(n) = U(S);$$
$$V'(n) = U'(S).$$

Let the *mean open search length* $\bar{l}(n)$ and the *mean internal search length* $\bar{l}'(n)$ now be defined as follows.

$$\bar{l}(n) = \frac{V(n)}{(n+1)!}.$$
$$\bar{l}'(n) = \frac{V'(n)}{n \cdot n!}.$$

THEOREM 1. *It is true for each n that*

$$\bar{l}(n) = 2\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}\right) \tag{1}$$

and

$$\bar{l}'(n) = \frac{n+1}{n}\bar{l}(n) - 1. \tag{2}$$

PROOF OF (1). Let each tree T^i have search subtrees T_σ^i. For each integer t, $0 \leqq t < n$, let Φ_t be the set of all sequences R^i such that R_0^i has length t. For each sequence R^i in Φ_t, the first element of R^i is $t+1$; and the set of elements in R_0^i is the set of the first t positive integers. Therefore, R_0^i is one of $t!$ sequences. Let P and Q be any two of these sequences. Let m_P and m_Q denote the number of R^i in Φ_t such that $R_0^i = P$ and $R_0^i = Q$, respectively. It is clear that $m_P = m_Q$. Since Φ_t has $(n-1)!$ members, it follows that $m_P = (n-1)!/t!$. Now T_0^i is the sequence search tree for R_0^i. Hence,

$$\sum_{R^i \text{in} \Phi_t} l(T_0^i) = \frac{(n-1)!}{t!} V(t).$$

Hence,

$$\sum_{i=1}^{n!} l(T_0^i) = \sum_{t=0}^{n-1} \frac{(n-1)!}{t!} V(t).$$

Symmetry requires that

$$\sum_{i=1}^{n!} l(T_0^i) = \sum_{i=1}^{n!} l(T_1^i).$$

For each T^i,

$$l(T^i) = l(T_0^i) + l(T_1^i) + n + 1.$$

It now follows that

$$V(n) = (n+1)! + 2(n-1)! \sum_{t=0}^{n-1} \frac{V(t)}{t!}.$$

Writing $V(n-1)$, solving for $\sum_{t=0}^{n-2} (V(t)/t!)$, substituting in $V(n)$ and collecting terms gives

$$V(n) = 2n! + (n + 1)V(n - 1).$$

Since $\bar{l}(n) = V(n)/(n + 1)!$ and $\bar{l}(n - 1) = V(n - 1)/n!$, this gives

$$\bar{l}(n) = \frac{2}{n+1} + \bar{l}(n - 1),$$

from which (1) is easily obtained.

Proof of (2). To prove this, we need only note that if a proper node of T^i belongs to m open search paths in T^i, then it belongs to $m-1$ internal search paths of T^i. Hence, for all i, $l'(T^i) = l(T^i) - n$. From this (2) is easily obtained. Q.E.D.

Note that with the use of $\int (dx/x)$, bounds can be put on $\bar{l}(n)$ thus:

$$2 \log_e \left(\frac{n}{2} + 1\right) < \bar{l}(n) < 2 \log_e(n + 1) \approx 1.4 \log_2(n + 1). \qquad (3)$$

1.3 *Deletion Trees*

In this section the notion of a "random deletion" will be considered.

Definition. Let T be a search tree of n proper nodes, S a set of n numbers, and y a member of S. The *deletion search tree* $D(T, S, y)$ is defined as follows. Let τ be such that $f_{ST}(p_\tau) = y$. Let $S' = S - \{y\}$. Let the nodes of $D(T, S, y)$ be denoted by $p_\sigma{}^D$. For each σ, let D_σ denote the search subtree of $D(T, S, y)$ having $p_\sigma{}^D$ as its root. Now $D(T, S, y)$ is given by (i) and (ii) below, viz:

(i) If $S_{\tau 1}$ is empty, then:
$D_\tau = T_{\tau 0}$;
for each node p_σ not belonging to T_τ, $p_\sigma{}^D = p_\sigma$.

(ii) If $S_{\tau 1}$ is not empty, then let ρ be such that $f_{ST}(p_\rho)$ is the smallest member of $S_{\tau 1}$. (Observe that $S_{\rho 0}$ must then be empty.) Now:
$D_\rho = T_{\rho 1}$;
for each node p_σ not belonging to T_ρ, $p_\sigma{}^D = p_\sigma$.

Note the following. In case (i), for every proper node q of T such that $q \neq p_\tau$, it is true that q belongs to $D(T, S, y)$ and $f_{S'D}(q) = f_{ST}(q)$. In case (ii): for every proper node q of T such that $q \neq p_\rho$, q belongs to $D(T, S, y)$; and for every proper node q of $D(T, S, y)$ such that $q \neq p_\tau$, it is true that $f_{S'D}(q) = f_{ST}(q)$; finally, $f_{S'D}(p_\tau) = f_{ST}(p_\rho)$.

For the trees T^i of the previous section, let D^{iy} denote, for each R^i in Φ and for each y in S, the deletion tree $D(T^i, S, y)$.

Before considering the search properties of the trees D^{iy}, we consider briefly the nature of the search paths t^{iy}, where t^{iy} is defined as follows. Let τ and ρ be as in the definition of D. For case (i) of this definition (where $p^i_{\tau 1}$ is a blank node of T^i), t^{iy} is the search path $p^i \cdots p^i_{\tau 1}$. For case (ii) ($p^i_{\tau 1}$ a proper node), t^{iy} is $p^i \cdots p^i_{\rho 0}$. Let $s_0{}^i$ be an open search path, defined for each R^i in Φ as follows.

If $s_0^i = q_1, q_2, \cdots, q_m$ then (q_j, q_{j+1}) is a left link of T^i for $1 \leqq j \leqq m$. Now it can be easily verified that the search paths t^{iy} are all the open search paths in the trees T^i excepting the search paths s_0^i. Let $V_0(n)$ denote the sum of the search lengths of the paths s_0^i. Let $\bar{l}_d(n)$ be defined by

$$\bar{l}_d(n) = \frac{V(n) - V_0(n)}{n \cdot n!}.$$

Since we are considering $n \cdot n!$ deletion trees, $\bar{l}_d(n)$ represents the mean search length of t^{iy}. $V_0(n)$ is found by a method exactly similar to that used for $V(n)$ in the previous section. The result is $V_0(n) = n!(1 + \frac{1}{2} + \cdots + (1/n))$. (Thus $V_0(n)/n!$, which represents the mean search length of s_0^i, is only about half as large as $\bar{l}(n)$.) Hence,

$$\bar{l}_d(n) = \bar{l}(n) + \frac{1}{n}\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} - 1 + \frac{2}{n+1}\right). \tag{4}$$

Hence, $\bar{l}_d(n)$ is approximately equal to $\bar{l}(n)$.

We now consider the search properties of the search trees D^{iy}.

There are $n \cdot n!$ pairs iy, and each corresponding D^{iy} has n open search paths and $n-1$ internal search paths. Hence, for the trees D^{iy} we define the mean open search length $\bar{l}_D(n)$ and the mean internal search length $\bar{l}_{D'}(n)$ as follows.

$$\bar{l}_D(n) = \frac{1}{n^2 \cdot n!} \sum_{i=1}^{n!} \sum_{y \text{ in } S} l(D^{iy})$$

and

$$\bar{l}_{D'}(n) = \frac{1}{(n-1)n \cdot n!} \sum_{i=1}^{n!} \sum_{y \text{ in } S} l'(D^{iy}).$$

Observe that, as before, the assertion that these quantities depend only on n is justified because two sequences having the same sequence of ranks have the same sequence search tree. Therefore, we take S to be the set of the first n positive integers.

THEOREM 2. *It is true for each n that*

$$\bar{l}_D(n) = \bar{l}(n-1) \tag{5}$$

and

$$\bar{l}_{D'}(n) = \bar{l}'(n-1). \tag{6}$$

PROOF. To show this, we will construct for each pair iy a sequence R^{iy}. R^{iy} will be such that D^{iy} is the sequence search tree for R^{iy}. Then, letting Q be any sequence equal to at least one of these sequences R^{iy}, we will show that there are exactly n^2 distinct pairs iy such that $R^{iy} = Q$. It will then follow that the set of pairs iy can be partitioned into n^2 sets such that Theorem 1 is directly applicable to each set. That is, if A is any one of these sets, then

$$\sum_{iy \text{ in } A} l(D^{iy}) = n!\bar{l}(n-1),$$

$$\sum_{iy \text{ in } A} l'(D^{iy}) = (n-1)(n-1)!\bar{l}'(n-1).$$

It will also follow that

$$n^2 \sum_{iy \text{ in } A} l(D^{iy}) = \sum_{i=1}^{n!} \sum_{y \varepsilon S} l(D^{iy}),$$

$$n^2 \sum_{iy \text{ in } A} l'(D^{iy}) = \sum_{i=1}^{n!} \sum_{y \varepsilon S} l'(D^{iy}).$$

Substitution of these values into the defining equations for $\bar{l}_D(n)$ and $\bar{l}_D'(n)$ will then yield (5) and (6).

Therefore, to prove the theorem, it only remains to find the sequences R^{iy}.

Let P be any sequence of length m, $m > 0$, and containing m distinct numbers. Let y be any term of P. For each such pair (P, y) we define the sequences $d(P, y)$ and $g(P, y)$ as follows.

Let $P = w_1, w_2, \cdots, w_m$. Let $y = w_j$ where $1 \leqq j \leqq m$. If there exists a number in P which is greater than y, let w_k be equal to the smallest such number. Let $d(P, y)$ be a sequence of length $m-1$ defined as follows.

(i) If y is the greatest number in P, or if y is not the greatest number in P and $j > k$, then $d(P, y)$ is a subsequence of P and contains every term of P except w_j.

(ii) If y is not the greatest number in P and $j < k$, then let Q' be a subsequence of P containing every term of P except w_k. Replace the jth term of Q' with w_k. The sequence so obtained is $d(P, y)$.

It can be verified easily that if S is the set of numbers in P, and if T is the sequence search tree for P, then $D(T, S, y)$ is the sequence search tree for $d(P, y)$.

$g(P, y)$ is defined to be the sequence of ranks for $d(P, y)$. That is, $g(P, y) = r[d(P, y)]$. Clearly, $D(T, S, y)$ is the sequence search tree for $g(P, y)$.

Now, let $R^{iy} = g(R^i, y)$. It follows from the foregoing that D^{iy} is the sequence search tree for R^{iy}.

Now, let Q be any sequence equal to at least one of the sequences R^{iy}. That is, Q is any sequence of the first $n - 1$ positive integers. That $Q = R^{iy}$ for n^2 distinct pairs iy will be proved by induction on n. The case $n = 1$ is trivial. The case $n = 3$ shows the nature of the problem and is exhibited in Table I.

TABLE I

Sequences R^{iy} for $n = 3$

i	R^i	R^{i1}	R^{i2}	R^{i3}
1	123	12	12	12
2	132	12	12	12
3	213	12	21	21
4	231	12	21	21
5	312	21	21	12
6	321	21	21	21

Consider arbitrary n. Let $Q = r_1, r_2, \cdots, r_{n-1}$. We define and consider in turn the subsets A_1, A_2, A_3 of the set of pairs iy.

Set A_1. iy is in A_1 if and only if $y = n$ is the first term of R^i. It is clear that there is exactly one pair iy in A_1 such that $R^{iy} = Q$.

Set A_2. A_2 is that set of pairs iy such that $y = r_1$. This condition implies $y \neq n$, thus excluding members of A_1. It also implies that the first element of R^i is either y or $y+1$. For each j, $2 \leqq j \leqq n$, there are exactly two pairs iy belonging to A_2 and satisfying both of the following.

(i) The jth element of R^i is y or $y+1$.

(ii) $R^{iy} = Q$. Thus, A_2 contains $2n-2$ pairs iy for which $R^{iy} = Q$.

Set A_3. A_3 is the set of all pairs iy which are not in A_1 or A_2 and for which the first term of R^{iy} is r_1. Clearly, each pair iy which we have yet to count belongs to A_3. A necessary and sufficient condition for a pair iy to belong to A_3 is given by the following (i) and (ii).

(i) Neither y nor $y+1$ is the first element of R^i (or else iy belongs to A_1 or A_2).

(ii) Let x^i be the first element of R^i. If $x^i > y$ then $x^i = r_1 + 1$. If $x^i < y$, then $x^i = r_1$.

Therefore, there are $n-1$ allowable values of y in A_3. These are all of the integers $1, 2, \cdots, n$ excepting r_1. $y < r_1$ if and only if $x^i = r_1 + 1$. $y > r_1$ if and only if $x^i = r_1$.

Now for each i let P^i be defined by $R^i = x^i P^i$. That is, P^i is the subsequence of R^i containing all but the first term. It can be verified that for iy in A_3, R^{iy} can be constructed as follows. Let t_y be the rank of y in P^i. Denote by w_j^{iy} the jth term of the sequence $g[r(P^i), t_y]$. Denote by v_j^{iy} the jth term of R^{iy}. By the definition of A_3, $v_1^{iy} = r_1$. v_j^{iy} for $1 < j \leqq n-1$ is as follows. If $w_j^{iy} < r_1$ then $v_j^{iy} = w_j^{iy}$. If $w_j^{iy} \geqq r_1$ then $v_j^{iy} = w_j^{iy} + 1$. Thus the sequence $g[r(P^i), t_y]$ uniquely defines R^{iy} for iy in A_3. Thus also if iy and kz are in A_3 and $R^{iy} = R^{kz}$ then $g[r(P^i), t_y] = g[r(P^k), t_z]$.

The induction assumption can be applied to the sequence $g[r(P^i), t_y]$. For, let Q' be any sequence of length $n-1$ containing each of the first $n-1$ positive integers. For each y it is true that there is exactly one pair iy in A_3 such that $r(P^i) = Q'$. This is verified as follows. It is clear that for each z, $1 \leqq z \leqq n$, there is one and only one pair iy such that $x^i = z$ and $r(P^i) = Q'$. But, as observed above, y uniquely defines x^i when iy is in A_3. Also, if $y < r_1$ then $x^i = r_1 + 1$ and hence $t_y = y$; and if $y > r_1$ then $x^i = r_1$ and hence $t_y = y - 1$. $y \neq r_1$ for any iy in A_3. It follows that t_y uniquely defines x^i for iy in A_3. Hence for a given t_y there is exactly one pair iy in A_3 such that $r(P^i) = Q'$. The induction assumption therefore applies to the sequences $g[r(P^i), t_y]$.

Now, let Q'' be the sequence such that if $g[r(P^i), t_y] = Q''$ then $R^{iy} = Q$. The induction assumption asserts that there are $(n - 1)^2$ pairs iy in A_3 such that $g[r(P^i), t_y] = Q''$. This would imply exactly $(n - 1)^2$ pairs iy in A_3 such that $R^{iy} = Q$. These, together with the one found in A_1 and the $2n-2$ found in A_2, complete the induction. Q.E.D.

2. APPLICATIONS

2.1 *Searching*

Computer programmers are familiar with the fact that lists which can be searched efficiently tend to be difficult to change efficiently, and vice versa. Some lists are never changed, and some are never searched. If a list needs to be both searched and changed, the conflict can often be resolved by devoting more storage to the list. But when not enough storage is available to resolve the conflict, a problem arises. This problem is to effect with the limited storage, a workable compromise between a search-oriented list design and change-oriented list design. In the next two sections we consider two well-known kinds of lists which illustrate this conflict. In Section 2.1.3 we consider a list which offers, at a reasonable cost in storage, a compromise between searching and changing; and we apply to this list the main results of Section 1.

2.1.1 *Sequence Lists*

The classic example of a search-oriented list is the following. Let $z_1, z_2, \cdots, z_n$ be an ascending sequence of length n. For each i, $1 \leqq i \leqq n$, let z_i occupy location $A+b_i$ in a computer. The well-known binary search algorithm is applicable to the list. This algorithm makes its first comparison with the number in location $A+b[n/2]$ (or thereabouts), thus eliminating from the search one of the sequences $\{Z_i\}$, $b \leqq i < b[n/2]$, or $\{Z_i\}$, $b[n/2] < i \leqq b_n$. The algorithm then applies itself recursively to the remaining sequence.

The search length, defined as the number of comparisons in a search, has an expected value of approximately $\log_2 n$. However, to make an insertion or a deletion in the list entails a lot of work. For, on the average, assuming random choice of the number to be inserted or deleted, $n/2$ numbers must be given new memory locations.

Hence, a sequence list is search-oriented but is not change-oriented.

Note that there is a binary search tree associated with the binary search algorithm. Each location corresponds to a proper node of the search tree. The tree has the property that it is balanced. That is, letting m_σ denote for each σ the number of nodes in T_σ, it is true for all σ that $| m_{\sigma 0} - m_{\sigma 1} | \leqq 1$.

2.1.2 *Singly Linked Lists*

The leading example of a change-oriented list is the singly linked or "push-down" list [3, 4]. Let $z_1, z_2, \cdots, z_n$ be a sequence. Let a singly linked list consist of a number l_0 together with a set $\{d_i \mid 1 \leqq i \leqq n\}$ of ordered pairs $d_i = (z_i, l_i)$. l_i, $0 \leqq i < n$, is the location[3] of d_{i+1}. l_n is any number not belonging to the set of possible locations.

Clearly, an insertion or deletion is accomplished in a singly linked list simply by changing one of the l_i and, for an insertion, adding a new pair to the set of pairs. Note, however, that searching in a singly linked list is a long process. Expected search time is proportional to the number n of entries in the list.

[3] The term *location* as used here is applicable to any set of memory components to which the program has ready access. Thus z_i and l_i, as well as d_i have locations.

TABLE II

A Singly Linked List

i	Location	x_i	l_i
6	5	200	ϕ
2	10	600	28
1	14	100	10
4	17	400	23
5	23	500	5
3	28	300	17

$l_0 = 14$

2.1.3 *A Doubly Linked List*

The results of Section 1 imply that a doubly linked list can be designed to give, under random conditions, expected search time and expected insertion/deletion time both proportional to log n.

Notation. Let I be a set of integers and let $L = \{(x_i, l_i, r_i) \mid i \text{ in } I\}$ be a set of triples. The *successor* relation is defined among the numbers x_i by the following two statements.

(1) If l_i (respectively r_i) is in I, then x_{l_i} (respectively x_{r_i}) is a successor of x_i.

(2) If x_j is a successor of x_i, then every successor of x_j is a successor of x_i.

DEFINITION. Let I be a set of storage locations. Let ϕ be a number which is not a storage location. A *doubly linked list* is a number E together with a set $L = \{(x_i, l_i, r_i) \mid i \text{ in } I\}$ of triples, having the following properties.

(1) Each triple (x_i, l_i, r_i) is in location i.

(2) If L is empty, then $E = \phi$; otherwise, E is in I.

(3) If $E \neq \phi$, then x_E together with the successors of x_E is equal to $\{x_i \mid i \text{ in } I\}$.

(4) For each i in I, if $l_i \neq \phi$ then x_i is greater than x_{l_i} and every successor of x_{l_i}.

(5) For each i in I, if $r_i \neq \phi$ then x_i is less than x_{r_i} and every successor of x_{r_i}.

An example of a doubly linked list is given in Table III. The same list is shown

TABLE III

A Doubly Linked List

i (location)	x_i	l_i	r_i
5	600	28	ϕ
10	300	ϕ	ϕ
14	500	ϕ	ϕ
17	200	23	5
23	100	ϕ	ϕ
28	400	10	14

$E = 17$

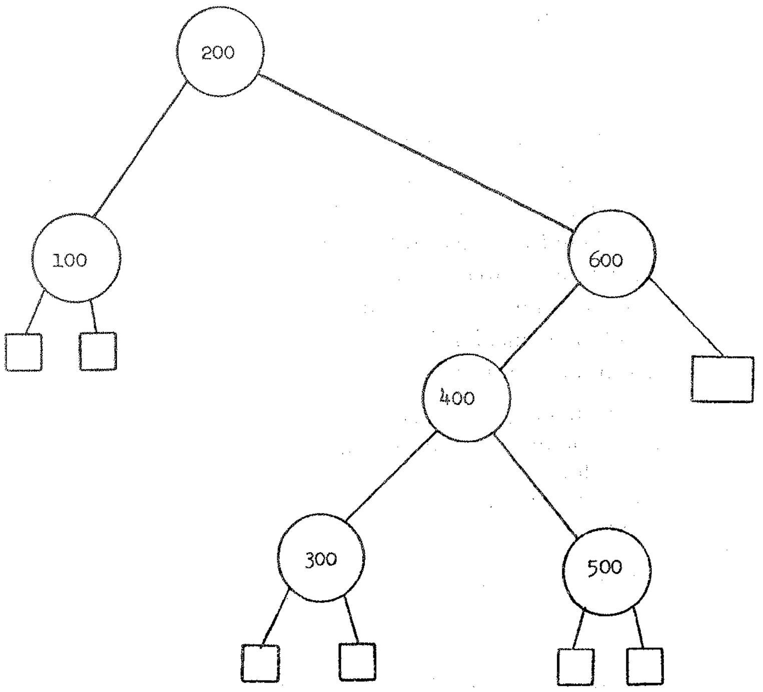

FIG. 1. The binary search tree corresponding to the list of Table III. Circles denote proper nodes, rectangles denote blank nodes. The numbers are the $f_{ST}(p_\sigma)$. The tree is the sequence search tree for the sequence (200, 600, 400, 100, 500, 300).

schematically in Figure 1. Observe that Figure 1 is also a schematic diagram of a search tree. The set of proper nodes of the tree is the set I of locations. The location E is the root. For two locations (i.e. nodes) i and j, if $j = l_i$ (respectively r_i) then (i, j) is a left (respectively right) link of the tree. Let T denote the tree and let $S = \{x_i \mid i \text{ in } I\}$. Then the list function is given by $f_{ST}(i) = x_i$ for each i in I. The root p of the tree is $p = E$. For any node $p_\sigma = i$, if $l_i \neq \phi$ then $p_{\sigma 0} = l_i$ and if $r_i \neq \phi$ then $p_{\sigma 1} = r_i$. If l_i (respectively r_i) $= \phi$ then $p_{\sigma 0}$ (respectively $p_{\sigma 1}$) is a blank node.

Let E be denoted by r_ϕ. A doubly linked list exists in storage in the form of three arrays, one array for each of the sets $\{x_i \mid i \text{ in } I\}$, $\{l_i \mid i \text{ in } I\}$, $\{r_i \mid i \text{ in } I \cup \{\phi\}\}$. The order within these arrays is unimportant.

Consider the following search algorithm. (The notation used in this and in the algorithms which follow it, is that of ALGOL 60 [8].)

```
procedure search (y) doubly linked list: (x,ℓ,r) main output: (z) secondary outputs: (i,j,w);
real array x;  integer array ℓ, r;  real y;
boolean z;  integer w, i, j;
comment  z is "true" if and only if y is in x. The secondary outputs are for the use of
  the "insert" and "delete" procedures defined below;
begin i := r_φ;  w := 1;  j := φ;
S1: if i = φ then z := false
else if y = x_i then z := true
else begin j := i;  if $y > x_i$ then begin i := $r_i$;  w := 1 end
else begin i := $\ell_i$;  w := 0 end;
go to S1 end
end search
```

A measure of the length of a search process is the number of comparisons performed. In the above algorithm, y is compared with each number in a sequence $x_{i_1} \cdots x_{i_w}$. The sequence $i_1 \cdots i_w$ is a search path in the search tree associated with the list. If y is found by the search, the path is an internal one, otherwise it is open. The number of comparisons in a search is equal to the length of the associated search path. The expected value of this quantity depends upon how the list is constructed. Consider the following "insert" algorithm, which uses the search algorithm as a subroutine.

```
procedure insert (y) doubly linked list: (x, ℓ, r) next available location: (k);
comment the array r contains not only the right links but a pushdown list of available
  storage beginning with $r_k$;
begin integer i, j, w;  boolean z;
search (y,x,ℓ,r,z,i,j,w);
if ¬ z then
  begin if w = 1 then $r_j$ := k else $\ell_j$ := k;  j := k;  k := $r_k$;  $x_j$ := y;
    $\ell_j$ := $r_j$ := $\phi$ end
end insert
```

Note that if the arrays x, l, r constitute a doubly linked list at the beginning of the execution of "insert" then they will do so at the end. Observe that the insertion is accomplished by changing four numbers in the list proper and one in the available storage list. Thus the only portion of the process which depends on the size of the list is that performed by the search algorithm.

Let $y_1 \cdots y_n$ be a sequence of n distinct numbers chosen and ordered randomly. Let $r_\phi = \phi$ initially (that is, let the list be essentially empty), and let "insert (y_i, x, l, r, k)" be performed for $i = 1, 2, \cdots, n$. The resulting list will have associated with it the sequence search tree for $y_1 \cdots y_n$. Therefore, Theorem 1 gives the expected number of comparisons in a search. That is, the expected number t of comparisons when y is not found is $t = \bar{l}(n)$, and the expected number t' when y is found is $t' = \bar{l}'(n)$. The overall expected number of comparisons depends on the probability that y will be found.

Consider now the following deletion algorithm.

```
procedure delete (y) doubly linked list: (x,ℓ,r) next available location: (k);
begin integer h, i, j, w;  boolean z;
search (y,x,ℓ,r,z,i,j,w);
if z then
  begin if $r_i = \phi$ then
    begin h := j;  j := i;  if w = 1 then $r_h$ := $\ell_i$ else $\ell_h$ := $\ell_i$ end else
    begin j := $r_i$;  if $\ell_j = \phi$ then
      begin $x_i$ := $x_j$;  $r_i$ := $r_j$;  go to C end;
    B:  h := j;  j := $\ell_j$;  if $\ell_j \neq \phi$ then go to B;  $x_i$ := $x_j$;  $\ell_h$ := $r_j$
    end;
    C:  $r_j$ := k;  comment see "insert" comment;  k := j
  end
end delete
```

It is directly verifiable that the list resulting from "delete" has associated with it the tree $D(T, S, y)$ of Section 1.3. Therefore, if y is selected at random from among the x_i then Theorem 2 is applicable. That is, the functional relationship between the expected search lengths t, t' and the number n of entries in the list, is preserved.

In general, therefore, we assert that, starting with an empty list and performing m_1 random insertions and m_2 random deletions in any order, if $m_1 - m_2 = n$ then

$$t = 2\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}\right) \approx 1.4 \log_2 n$$

and

$$t' = \frac{n+1}{n} t - 1 \approx t - 1.$$

The first approximation is by (3), Section 1.2.

Consider the efficiency of the deletion algorithm. The algorithm first locates the number y in the list. It then finds the smallest successor of y. These two processes together correspond to the paths t^{iy} discussed in Section 1.3. The expected number of list elements handled is the average length $\bar{l}_d(n)$ of t^{iy} given in (4). Thus, the searching portion of the deletion algorithm has an expected length only slightly larger than t.

The deletion process modifies three to five numbers in the list, regardless of the size of the list.

The storage required is, in a typical application, about double the required storage of a sequenced list and about half again that of a singly linked list.

In return for this moderate amount of storage, both search time and insertion/deletion time are proportional to log n.

2.1.4 *An Unsolved Problem*

It is a consequence of the results of [1] that the condition for optimal searching in a binary search tree T is as follows. We assume random choice of the search operand. Let m_σ denote the number of proper nodes in T_σ. Let r_σ denote the integral power of 2 such that $r_\sigma \leqq \max(m_{\sigma 0}, m_{\sigma 1}) < 2r_\sigma$. The condition for optimal searching is that, for each σ, $\min(m_{\sigma 0}, m_{\sigma 1}) \geqq r_\sigma - 1$.

Now if the numbers m_σ are adjoined to the doubly linked list of the previous section, then an insertion algorithm which ensures the above condition can be designed. While such an algorithm would tend generally to be inefficient, it would clearly be more efficient than the insertion algorithm for the sequence list (Section 2.1.1). Note that the complexity of the list transformation would depend on the proximity of $m_\wedge$ to a power of 2. A quantitative analysis of this insertion algorithm is still being sought.

2.2 *A Sorting Algorithm*

Constructing the list of Section 2.1.3 is, in a sense, a sorting operation. However, it is slower than other known sorting methods which use a comparable amount of storage. But when sorting is to be done with a minimum of storage, the results of Section 1.2 suggest a method which does have some advantages. The method turns out to be very similar to Hoare's "Quicksort" [9] but with two important differences which are discussed below. The advantages are

(i) an expected number of comparisons approximately equal to $1.4\ n \log_2 n$ (in close agreement with Hoare's reported average of $2n \log_e n$);
(ii) auxiliary storage proportional to $\log_2 n$;
(iii) insensitivity to distribution.

Any pair of these properties can be obtained, or improved upon, in other sorting schemes: but the scheme defined below is the only one known by the author to have all three properties. Quicksort has properties (i) and (iii) but not (ii), as is shown below.

The expected number of comparisons required to construct the sequence search tree for a given sequence is calculated as follows. Let $\{l_i' \mid i = 1, 2, \cdots, n\}$ be the lengths of the internal search paths, as in Section 3. Then $\sum_{i=1}^{n} (l_i' - 1)$ is the number of comparisons required to construct the tree. Hence the expected number $\bar{c}$ of comparisons to construct a sequence search tree is $[V'(n) - n \cdot n!]/n!$ Since $V'(n)/n! = n\bar{l}'(n)$, we have

$$\bar{c} = n\bar{l}'(n) - n \approx 1.4(n + 1) \log_2 n - 2n. \tag{7}$$

Note that the derivation of (7) requires the assumption that the sequence contains n distinct numbers.

The sorting algorithm which we wish to consider, while it does not construct an explicit search tree,[4] performs a sequence of comparisons which is equivalent to constructing a search tree. The method first compares a_1 with each of a_2, $\cdots$, a_n, thus calculating the rank of a_1. a_1 is then stored in the location corresponding to its rank. In the process, all numbers less than a_1 get stored in locations below that of a_1, and the rest of the numbers get stored above. As will be seen in the algorithm, this is accomplished with just one extra storage location. At this point, the task is reduced to that of sorting two sequences: the sequence stored below a_1, and the sequence stored above a_1. Each of the sequences is sorted by a recursive application of the algorithm, one of the two sequences remaining untouched until the other is completely sorted. It is necessary to store the terminal locations of the sequence which is not sorted first; thus there will be a table of such locations. By sorting first the smaller of the two sequences flanking a_1, the length of the table is forced not to exceed $\log_2 n$.

The method bears a strong resemblance to the radix exchange method [7], the principal difference being that the present method employs a comparison, instead of a digit inspection, as its basic operation.

Quicksort, instead of calculating the rank of a_1, calculates the rank of a "randomly" chosen member of the sequence. Quicksort appears at first glance to have no table like the one mentioned above. These differences are evaluated

[4] (Added in proof). It has been brought to my attention that Windley [WINDLEY, P. F. Trees, forests and rearranging, *Comput. J. 3* (1960), 84–88] has examined the explicit construction of the tree as a sorting technique and has derived an expression equivalent to (7). Windley has also calculated a mean deviation associated with (7).

below. First let us define the present algorithm more precisely by means of ALGOL 60 as follows.

```
procedure  P(a, n);  value n;  array a;  integer n;
begin  integer e,h,i,j,k;  integer array f,g(1: $\log_2$ n);  real d;
e := 1;  h := n;  k := 1;
B:  if e $\geqq$ h then go to C;  d := $a_e$ ;  i := e;  j := h;
E:  if $a_j$ < d then
    begin $a_i$ := $a_j$ ;  i := i + 1;  if i = j then go to D end
    else begin j := j − 1;  if i = j then go to D;  go to E end;
F:  if $a_i$ > d then
    begin $a_j$ := $a_i$ ;  j := j − 1;  if i = j then go to D;  go to E end
    else begin i := i + 1;  if i = j then go to D;  go to F end;
D:  $a_i$ := d;
    if i − e < h − i then
    begin $f_k$ := i + 1;  $g_k$ := h;  h := i − 1 end
    else begin $f_k$ := e;  $g_k$ := i − 1;  e := i + 1 end;
    k := k + 1;  go to B;
C:  k := k − 1;  if k > 0 then begin e := $f_k$ ;  h := $g_k$ ;  go to B end
end
```

From the following considerations, it can easily be verified that each execution of this algorithm has a search tree associated with it. Let the execution of the algorithm start at time 0. Let $A(t) = a_1(t), \cdots, a_n(t)$ denote the value of A at any time t. Let α denote the time of the completion of the first execution of statement D. At time α, $a_1(0)$ has been compared with every other number in A, and $a_i(\alpha) = a_1(0)$. Moreover, $a_m(\alpha) < a_1(0)$ for all $m < i$, and $a_m(\alpha) > a_1(0)$ for all $m > i$. That is, $a_1(0)$ occupies its proper position in the sequence at time α, and therefore undergoes no more comparisons. Clearly, $a_1(0)$ thus corresponds to the root of the search tree. The algorithm is then applied recursively to the sequences $a_1(\alpha), \cdots, a_{i-1}(\alpha)$ and $a_{i+1}(\alpha), \cdots, a_n(\alpha)$ (not necessarily in that order). $a_1(\alpha)$ will therefore correspond to the node p_0 of the search tree and $a_{i+1}(\alpha)$ will correspond to the node p_1. Or, in terms of the list function f for the search tree and the set of elements in A, we have $f(p) = a_1(0)$, $f(p_0) = a_1(\alpha)$ and $f(p_1) = a_{i+1}(\alpha)$. Continuing in this way will, clearly, define a list function, and therefore a search tree.

It can also be verified that if A is a sequence of the first n integers, then the $n!$ possible values of $A(0)$ correspond in a 1-to-1 way with the sequences R^i of Section 1.3. That is, if T is the search tree corresponding to the execution of the algorithm for $A(0)$, then the sequence R^i corresponding to $A(0)$ is such that T is the sequence search tree for R^i.

It follows that (7) gives the expected number of comparisons in an execution of the algorithm for a randomly chosen $A(0)$.

Similar considerations show that (7) is also applicable to Quicksort, thus verifying Hoare's reported average of $2n \log_e n$ comparisons.

The only storage requirement, beyond that necessary for the program itself and for the sequence A, is the storage necessary for the numbers f_k, g_k. The maximum value of k during the algorithm does not exceed $\log_2 n$. This is ensured because, at each completion of statement D, the algorithm chooses the smaller of the two subsequences open to it.

The numbers f_k, g_k do not appear in Quicksort because of the recursive way in which that algorithm is defined. But it is clear that at each new level of recursion some information about the previous level must be retained; that is, the f_k, g_k are needed by Quicksort too. Moreover, k is there allowed to attain the value n; thus Quicksort does not have property (ii) above. It can, of course, be given that property by the same means used in the present algorithm.

Equation (7) holds for the present algorithm if the sequence of ranks is random. Since this implies nothing about the distribution of the numbers in the sequence, property (iii) holds. If the sequence of ranks is not random then the algorithm might be slowed down considerably. The worst case, $n(n - 1)/2$ comparisons, occurs with either an ascending or a descending sequence.

"Quicksort's" worst case is also $n(n - 1)/2$ comparisons, but the question of which sequence gives rise to it depends on the nature of the "random" choice of an element from the subsequence $a_e, \cdots, a_h$. Although this choice requires an amount of extra work proportional to n, it has an advantage if the sequence of ranks is biased. For, even though it is possible for a bias toward Quicksort's worst case to exist (since nothing is really random on a computer), this kind of bias seems much less likely to exist than a bias toward ascending or descending order.

If the present algorithm were preceded by a "random" scrambling of the sequence (a process with duration proportional to n) it would be equivalent to Quicksort.[5]

A search of the literature shows that Shell's method [5, 6] is the fastest sorting algorithm having the property of being insensitive to distribution and having a storage requirement comparable to that of the present algorithm. The expected number of comparisons for Shell's method, given a randomly ordered sequence, is given approximately[6] by $n(.194(\log_2 n)^2 - .77 \log_2 n + 8.4)$. Hence the present algorithm is, for large enough n, superior to Shell's method. It will be superior to the radix exchange method [7] only in cases of uneven distribution.

REFERENCES

1. Huffman, D. A. A method for the construction of minimum redundancy codes. *Proc. I.R.E. 40* (1952), 1098–1101.
2. Burge, W. H. Sorting, trees and measures of order. *Informat. Contr. 1* (1958), 181–197.
3. Carr, John W., III. Recursive subscripting compilers and list-type memories. *Comm. ACM 2* (1959), 4–6.
4. Newell, A., and Shaw, J. C. Programming the logic theory machine. Proc. Western Joint Comput. Conf. (1957), 230–240.
5. Shell, D. L. A high speed sorting procedure. *Comm. ACM 2*, No. 7 (1959), 30–32.
6. Frank, R. M., and Lazarus, R. B. A high speed sorting procedure. *Comm. ACM 3* (1960), 20–22.
7. Hildebrandt, P., and Isbitz, H. Radix exchange—an internal sorting method for digital computers. *J. ACM 6* (1959), 156–163.
8. Nauer, P. (ed.); Backus, J. W., et al. Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (1960), 299–314.
9. Hoare, C. A. R. Algorithms 63 and 64. *Comm. ACM 4* (1961), 321.

[5] Except when equal elements are allowed, in which case a slight modification of the present algorithm is desirable.

[6] Experimentally determined by the author on an IBM 709 computer; holds for $n \geqq 200$.