References
1. Baer, J. L., and Schwab, B. A comparison of tree-balancing algorithms. *Comm. ACM 20*, 5 (May 1977), 322–330.
2. Foster, C.C. A generalization of AVL trees. *Comm. ACM 16*, 8 (Aug. 1973), 513–517.
3. Hirschberg, D.S. An insertion technique for one-sided height-balanced trees. *Comm. ACM 19*, 8 (Aug. 1976), 471–473.
4. Karlton, P.L., Fuller, S.H., Scroggs, R.E., and Kaehler, E.B. Performance of height-balanced trees. *Comm. ACM 19*, 1 (Jan. 1976), 23–28.
5. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
6. Luccio, F., and Pagli, L. On the height of height-balanced trees. *IEEE Trans. Comptrs. C-25*, 1 (Jan. 1976), 87–90.
7. Luccio, F., and Pagli, L. Polynomial b-balanced trees. Proc. 15th Annual Alberton Conf., Monticello, Ill., Sept. 1977, pp. 8–15.
8. Nievergelt, J. Binary search trees and file organization. *Comptng. Surveys 6*, (Sept. 1974), 195–207.
9. Reingold, E.M., Nievergelt, J., and Deo, N. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1977.

Programming Techniques

S.L. Graham, R.L. Rivest Editors

Median Split Trees: A Fast Lookup Technique for Frequently Occurring Keys

B.A. Sheil
Harvard University

Split trees are a new technique for searching sets of keys with highly skewed frequency distributions. A split tree is a binary search tree each node of which contains two key values—a *node* value which is a maximally frequent key in that subtree, and a *split* value which partitions the remaining keys (with respect to their lexical ordering) between the left and right subtrees. A *median* split tree (MST) uses the lexical median of a node's descendents as its split value to force the search tree to be perfectly balanced, achieving both a space efficient representation of the tree and high search speed. Unlike frequency ordered binary search trees, the cost of a successful search of an MST is log n bounded and very stable around minimal values. Further, an MST can be built for a given key ordering and set of frequencies in time n log n, as opposed to n^2 for an optimum binary search tree. A discussion of the application of MST's to dictionary lookup for English is presented, and the performance obtained is contrasted with that of other techniques.

Key Words and Phrases: tree search, dictionary lookup, binary search, heaps, balanced trees, Zipf's Law, information retrieval
CR Categories: 3.74, 5.25, 5.39

A version of this paper was presented at the Annual Meeting of the Association for Computational Linguistics, San Francisco, 8–9 October 1976.
Author's current address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.
© 1978 ACM 0001-0782/78/1100-0947$00.75

1. Introduction

A central task in many applications is the retrieval, from a table of identifier-information pairs, of the information associated with an identifier (or "key") presented in the input. Many such applications involve sets of keys whose frequencies of occurrence have highly skewed distributions, so that a small fraction of the keys accounts for a large proportion of the requests. Indeed, such distributions are so common that it has been argued that they are an inherent property of human symbol use [9]. Common applications in which such key sets arise include dictionary lookup for natural language processing (e.g. in English, the most frequent 127 words account for about 50 percent of an average text), the recognition of operating system commands, and the symbol table lookup for a compiler (where the language keywords and certain commonly used variable names will be disproportionately frequent).

Lookup techniques which ignore these frequencies tend to scatter the high frequency keys uniformly throughout the table resulting, if the table is large, in an unacceptably high rate of external memory references. It is essential, therefore, to separate out the more frequent keys into a separate table that can be searched without external memory references. We present here a technique, a modification of frequency ordered binary search trees, for searching such tables. We show its search time to be log n bounded[1] and stable around a value that is optimal for any search procedure based on comparisons. We contrast our solution with other techniques, and present an illustrative application to dictionary lookup for English.

2. Median Split Trees

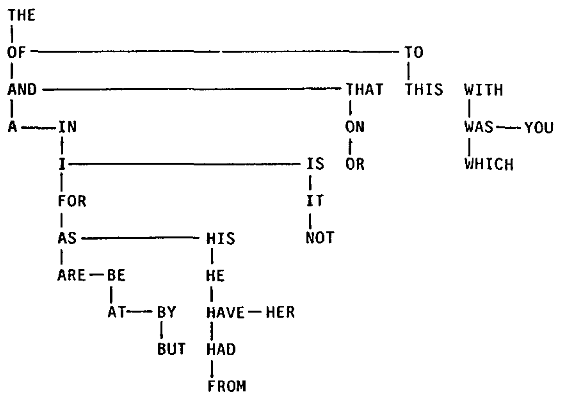
The classical solution to the organization of the high frequency section of a two level lookup system is the frequency ordered binary search (FOBS) tree—a binary search tree formed by inserting keys in decreasing frequency order. Such a tree for the 31 most common words of English, based on data from Knuth [5, p. 432] is presented in Figure 1. Searching is carried out using the usual binary tree search algorithm

```
search (tree, key) =
  IF tree = empty_tree THEN FAIL
                  ELSE
  IF tree.value = key THEN SUCCEED
                  ELSE
  search ((IF tree.value < key THEN tree.right_son
                          ELSE tree.left_son), key)
```

where < indicates comparison between keys in terms of some lexical ordering. We assume that keys occupy unit storage and that their comparison requires unit time. If this is not so (e.g. for variable length character keys),

[1] Unless otherwise indicated, all logarithms are base 2.

Fig. 1. Frequency ordered binary search tree for the 31 most common words of English (using alphabetic ordering on keys). The average cost per successful search is 4.042. Trees are shown in a dominance/succession format in which vertical alignment is used between a node and its left son, and horizontal alignment is used between brother nodes. Thus in this tree, OF and TO are the left and right sons, respectively, of THE. Empty nodes are suppressed to conserve space, but their presence can be inferred from the dominance relations and the lexical ordering. Thus it can be seen that I has only a left son (the subtree rooted at FOR) as the key to FOR's right (IT) is the (right) son of another node (IS).



then, rather than building the tree directly from the actual keys, we may instead build it using some easily computed attribute of them for which these properties hold (e.g. a subsequence or integer encoding), provided that the keys in the tree are uniquely encoded and that the full keys are checked after a successful search.

The problem with the FOBS tree solution is that, like other binary search trees, they can become unbalanced if the order of insertion (in this case, the frequency ordering) is confounded with the lexical ordering. Thus in Figure 1, the use of alphabetic ordering coupled with the accident of "the" being the most common word in English leads to a 24–6 split at the second level. Similar imbalances occur at many other nodes in the tree, resulting in an increase in the search time due to the low information yield of the tests at these nodes. Consequently, rather than inserting the nodes in strict frequency order, FOBS trees are commonly built using an insertion order which is chosen to minimize the average number of nodes of the tree visited per successful search by producing a more balanced tree. Figure 2, also from Knuth [5, p. 433] shows an optimum FOBS tree for the keys of Figure 1.

However, it would be surprising if the technique used to produce Figure 2 could not be improved on, as the balance of the tree is being improved at the cost of placing some high frequency nodes lower in the tree than one would choose on the basis of their position in the frequency ordering. This tradeoff reflects the different functions of the two tests against the key in the search procedure. The first is an identity test and should therefore be against the most frequent key in the current subtree. The second partitions the remaining keys on the basis of the lexical order and should therefore be against the key which yields optimal subtrees. A *split tree* is a

Fig. 2. Optimum Frequency Ordered Binary Search Tree for the 31 Most Common Words of English (Using Alphabetic Ordering on Keys). The Average Cost Per Successful Search is 3.437.

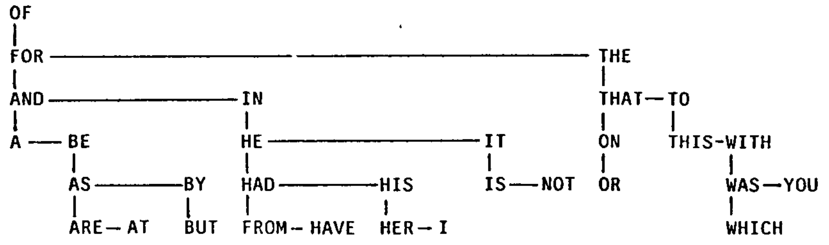
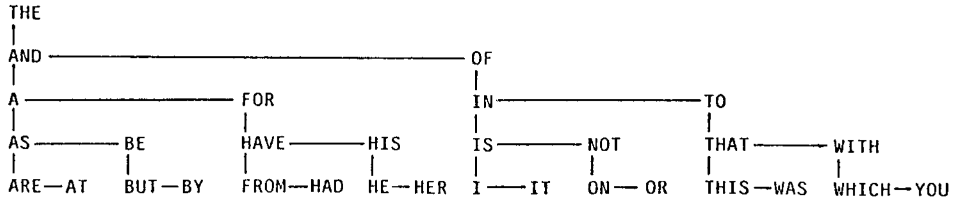


Fig. 3. Median Split Tree for the 31 Most Common Words of English (Using Alphabetic Lexical Ordering). The Average Cost Per Successful Search is 3.127.



binary search tree in which these two functions are represented by two values stored in each node—a *node value*, which identifies the key which resides at the node, and a *split value*, which partitions the remaining nodes between the left and right subtrees. Although we will consistently use the largest key in the left subtree as the split value, any key value on the closed interval between the largest key in the left subtree and the smallest in the right subtree will do. The search procedure for split trees

```
search (tree, key) =
  IF tree = empty_tree THEN FAIL
                   ELSE
  IF tree.node_value = key THEN SUCCEED
                         ELSE
  search ((IF tree.split_value < key THEN tree.right_son
                                   ELSE tree.left_son), key)
```

decides whether to go right or left from a node whose node value does not match the search key by comparing it to the split, rather than the node, value.

Although from an information-theoretic point of view the use of a different target for the second comparison increases the cost of a node inspection, such an increase is not practically significant. In many environments, retrieving information from the first comparison is either impossible or insignificantly faster than a second comparison. On the other hand, if comparisons are non-atomic, so the amount of retrievable state is substantial, an algorithm like digital search which explicitly makes use of that state is more appropriate.

The use of a second key in each node allows the most frequent key in a subtree to be stored at its root, without constraint on the structure of the rest of the subtree. This decoupling makes possible substantial savings in search cost given an appropriate choice of split value. One such choice is the split value such that the number of nodes visited per successful search of the subtree is minimized. While such a choice, by definition, produces an optimal split tree, it has disadvantages. Chief among these is that the determination of this split value seems to be computationally intractable. The removal of the most frequent key into the node value, irrespective of its position in the lexical order, results in the set of keys that has to be partitioned between the left and right subtrees being noncontiguous in the lexical order. This makes it difficult to apply the dynamic programming techniques which make the problem tractable for FOBS trees. Fortunately, an alternative choice which avoids this problem is available.

Median split trees (MST's) are, as their name suggests, split trees in which a node's split value is the median (with respect to the lexical ordering) of the node values of that node's descendents. This choice of split value is attractive as it results in a perfectly balanced tree, and thus optimal information yield from each comparison. Consequently, as any unbalanced tree sacrifices the information yield of its comparisons in order to keep some of the high frequency keys high in the tree, the optimal split tree and the MST for a set of keys are usually identical and invariably of comparable performance. As opposed to the local irregularities that degrade FOBS trees, the lexical and frequency orderings must be nearly completely confounded in order to produce key sets for which the MST is not the optimal split tree. Figure 3 shows the MST (which is also the optimal split tree) for the data of Figure 1 (only the node values are shown, as the split values can be inferred from the subtree they dominate). As can be seen, the MST has significantly better performance than either FOBS tree for this data. Furthermore, as we will demonstrate in greater detail in a later section, this performance is not highly dependent on the lexical ordering used. Different lexical orderings for this data produce search costs ranging from 3.092 to 3.416 nodes per successful search, the average of a large number of randomly chosen orderings being 3.147.

A further pleasant, but nonobvious, consequence of the use of the median as the split value is that, as it results in a complete binary tree, we can use any bijection between the nodes of such a tree and the integers from 1 to n to represent the tree in sequential storage locations, dispensing with the link fields used to locate a node's sons. This allows us to use *fewer* fields for our two valued nodes than was used in the FOBS tree's one valued nodes. Further, if the breadth-first, top-down, left to right enumeration of complete trees is used, the sons of node i will be found at $2i$ and $2i + 1$ [3, p. 401], a fast enough calculation so that this savings in storage can be bought at no cost in execution time. A typical implementation of the MST search procedure uses about six machine instructions per node visited for the main search loop.

Construction of Median Split Trees

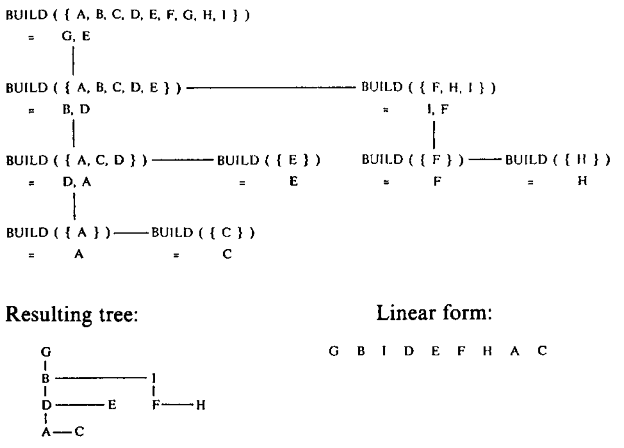
An MST is constructed from two orderings on the set of keys—the *frequency ordering* which is a weak linear order of the keys in terms of their frequencies of occurrence in the input, and the *lexical ordering* which is a strict linear order on their representations. Given two such orderings, an MST may be constructed for a set of keys $\{k_1, k_2, \ldots, k_m\}$ by

```
build ($\{k_1, k_2, \ldots, k_m\}$) =
  IF $m = 0$ THEN empty_tree
        ELSE
  $v \leftarrow$ any $k_i$ such that frequency ($k_i$) $\geq$ frequency ($k_j$) for $1 \leq j \leq m$;
  $s \leftarrow$ lexical_median ($\{k_1, k_2, \ldots, k_m\} - \{v\}$);
  tree OF [node_value: $v$,
          split_value: $s$,
          left_son:   build ($\{k_i | k_i \neq v \;\&\; k_i \leq s\}$),
          right_son:  build ($\{k_i | k_i \neq v \;\&\; k_i > s\}$)]
```

Each subtree is formed by selecting as the root's node value a maximally frequent key from among those to appear in the subtree; selecting the median (with respect to the lexical ordering) of the keys remaining in the subtree as the root's split value; and calling the procedure recursively on the two induced subtrees.

Unless m is of the form $2^L - 1$, for some integer L, the resulting tree will be unbalanced and its linear representation will require $(2^{\lfloor \log n \rfloor + 1} - 1) - n$ dummy nodes in its frontier level. A better solution is to choose as the split value, not the median, but the key whose index in the lexical order is $\max(h, n - h)$ where h is the integer of the form $2^L - 1$ closest to $\lceil n/2 \rceil$. This produces an unbalanced, but complete, tree in which the $n - (2^{\lfloor \log n \rfloor} - 1)$ nodes which overflow the balanced complete tree of level $\lfloor \log n \rfloor$ are all in the leftmost positions of the frontier level (cf. Figure 4, which shows the construction of an MST for nine keys). This is desirable because, in the sequential storage system used for MST's, the "missing" keys all occupy contiguous

Fig. 4. Construction of a median split tree for a set of nine keys. Lexical order: A, B, C, D, E, F, G, H, I (ascending). Frequency order: G, I, B, D, E, A, F, H, C (descending).



locations at the extreme right end of the linear array. Consequently, as no search will ever match them, these entries may simply be omitted and the usual search procedure will work without change or wasted storage.

Proposition. *Given a set of n keys, the* MST *construction algorithm requires $O(n \log n)$ time.*

Proof. When called on a set of m keys, the algorithm takes $O(m)$ time before calling itself recursively, as all steps before the recursive call can be carried out in time proportional to m if a linear time median algorithm [e.g. 1] is used. The time for n keys, $T(n)$, is therefore $2T((n - 1)/2) + O(n)$ which is $O(n \log n)$.

This performance is a significant improvement over the $O(n^2)$ required for the construction of an optimal FOBS tree [4]. Further, although the algorithm could clearly be improved (e.g. by keeping the keys sorted by both orders), its execution time is of optimal order. To see this, we observe that when the frequency and lexical orderings are the same, the tree produced has the ith most frequent key as the node value of the ith tree node in a depth-first enumeration of the tree (this is shown in the next section). Consequently, one can sort an arbitrary sequence with respect to some ordering by constructing an MST on its elements using this ordering as both frequency and lexical orderings, and then outputting the node values of the MST in depth-first order. A method for constructing MST's in less than $O(n \log n)$ would, therefore, imply a less than $O(n \log n)$ algorithm for sorting.

Search Performance

The expected cost of searching an MST, like a FOBS tree, depends on both the frequency distribution and the lexical ordering of the keys. As the frequency distribution of a set of keys is both variable across applications and not subject to manipulation, we will concentrate on the impact of different lexical orderings on search performance independently of any specific frequency distribution. We will show that, unlike FOBS trees, MST search time is log n bounded, stable around the optimum value, and, for frequency distributions typical of those encountered in practice, is approximately $(\log n)/2$.

Throughout this section, we will assume that we have a set of n keys (for $n = 2^L - 1$ for some integer L), $k_1, k_2, \ldots, k_n$, arranged in descending frequency order (i.e. $i > j \Rightarrow p_i \le p_j$). The lexical ordering on the keys will be written $k_i > k_j$ if k_i precedes k_j. We choose to regard the keys as being inherently ordered by frequency and only secondarily ordered lexically to emphasize that the lexical ordering is variable, in a way that the frequency ordering is not. The frequencies of occurrence are an attribute of the keys over which we have no control. However, although there is usually some salient lexical ordering defined on a set of keys (e.g. lexicographic ordering for strings), other lexical orderings can easily be generated (e.g. by encoding the keys with a simple integer encoding function and using arithmetic comparison on the encoded values) should the salient lexical ordering produce undesirable behavior. Different encodings will produce different orderings and thus different search behavior.

The result of an MST construction can be regarded as an assignment of each key to the node in which it is stored as node value. This mapping allows one to intermix key and node terminology. For example, we will define the *father* of a key k_i to be the key stored at the node which is the father of the node containing k_i. Note that not all mappings from keys to nodes are possible. Specifically, a complete binary tree is a valid MST (or, more concisely, is *valid*) only if it forms a *heap* [5]—that is, the key stored at each node is at least as frequent as that stored at either of its sons. This property follows directly from the construction algorithm's selection of a maximally frequent key as the node value of the root of any subtree, and is central to the technique's efficiency, although the search procedure itself does not assume it.

The mapping from keys to nodes implicitly assigns each key k_i a *level*, l_i, giving the length of the path from the root to the node containing k_i as node value + 1. As the level of a node is equal to the number of nodes inspected by a successful search which terminates at it, the *average cost of a successful search* (ACSS) will be $\sum^n p_i l_i / \sum^n p_i$. This will be our measure of search performance, as neither the cost (L) nor the probability ($1 - \sum^n p_i$) of an unsuccessful search is affected by the structure of the MST.

Although we will usually assume that the frequency distribution is unknown, the uniform distribution (i.e. $p_i = c$ for all i) is an interesting special case as it gives us an upper bound on the cost of MST search. As an MST is intended to take advantage of a skewed frequency distribution, one might expect it to perform worst when the frequency distribution is flat. That this is indeed the case is shown by the following theorem.

THEOREM. *For given n, an MST on n keys will have maximal ACSS if $p_i = c$ for $1 \le i \le n$.*

PROOF. We begin by noting that any monotonically decreasing set of probabilities can be obtained by starting with a uniform distribution, repeatedly selecting some point k, and "shifting" some proportion of the distribution from those points greater than k to those that are less. That is, for $1 \le k < n$, set $p_i' = p_i + \delta/k$ if $i \le k$ and $p_i = p_i - \delta/(n-k)$ if $i > k$. The change in the ACSS produced by one step of this procedure is

$$\sum^n p_i' l_i \Big/ \sum^n p_i' - \sum^n p_i l_i \Big/ \sum^n p_i$$
$$= \sum^n l_i(p_i' - p_i) \Big/ \sum^n p_i, \quad \text{as} \quad \sum^n p_i' = \sum^n p_i.$$

Dividing this up into terms to the left and right of k gives us

$$\left[\sum^k l_i\delta/k - \sum_{k+1}^n l_i\delta/(n-k)\right] \Big/ \sum^n p_i$$

which, as both δ and $\sum^n p_i$ are positive, is of the same sign as

$$\left[\sum^k l_i/k - \sum_{k+1}^n l_i/(n-k)\right].$$

LEMMA. *For any* MST, *and $k < n$, $\sum^k l_i/k < \sum_{k+1}^n l_i/(n-k)$.*

PROOF. $\sum^k l_i/k < \sum_{k+1}^n l_i/(n-k)$ iff $\sum^k l_i/k < \sum^n l_i/n$ i.e. iff the average of the first k is less than the average of the whole tree. If this is not so then, as l_1 is the unique minimum level (1) and is therefore overrepresented in the first k, there must exist levels j and j' such that $j' > j$ and j' is overrepresented in the first k, relative to j. That is, if we denote the number of the first k keys at level j by $C_k(j)$, $C_k(j) < C_k(j')/2^{j'-j}$. But, in a valid MST, to each key k_i at level $l_i > 1$ there corresponds a key $k_{i'}$ at level $l_i - 1$, such that $p_{i'} \ge p_i$. In either case, if $i \le k$, $i' \le k$, for if $p_{i'} > p_i$ then $i' < i$ which is no greater than k; whereas, if $p_{i'} = p_i$ then $i' > k$ would imply that two keys on opposite sides of k have equal frequencies. Thus as each node has at most 2 sons, if x of the first k are at level j', at least $\lceil x/2 \rceil$ of the first k must be at level $j' - 1$. By a simple induction argument, as $\lceil \lceil x/a \rceil / b \rceil = \lceil x/ab \rceil$ for any integer b, it follows that $C_k(j) \ge \lceil C_k(j')/2^{j'-j} \rceil$, for any $j' > j$. Therefore there does not exist any level that is underrepresented in the first k, relative to its successors, and the lemma is proven.

As $[\sum^k l_i/k - \sum_{k+1}^n l_i/(n-k)]$ is negative, the cost of each step in the transformation is also negative. Thus as each step in the transformation of a set of uniform probabilities into any other set reduces the associated cost, the result of this sequence of transformations has a lower ACSS than the uniform distribution. But, as any nonuniform distribution is the result of some such sequence of transformations, the uniform distribution must be pessimal.

The importance of the uniform distribution being pessimal is that its performance is quite acceptable. Specifically, the average cost per successful search is

$$\sum^{n} p_i l_i \Big/ \sum^{n} p_i = c \sum^{n} l_i/cn = \sum^{n} (\lfloor \log i \rfloor + 1)/n$$
$$= ((n+1)/n) \log (n+1) - 1.$$

That this is identical to the performance of binary search for n of the form $2^L - 1$ [5, p. 411] is not surprising as it is clear that MST search on a uniformly distributed key set reduces to uniform binary search.

If the frequency distribution is not uniform, the performance of the corresponding MST will depend on the degree to which the frequency and lexical orderings are confounded. We saw previously that a FOBS tree could become unbalanced and inefficient if the lexical ordering and the frequency ordering were confounded. A similar, although much more limited, inefficiency can result in MST's if the high frequency nodes are all clustered on one side of the tree, since that will cause some of them to have longer search paths than one would desire on the basis of their frequency.

As the search cost of an MST is determined by the set of level assignments we will find it convenient to define a canonical form for the set of trees which induce identical level assignments, and which therefore have identical costs for any frequency distribution. The *left canonical form* of an MST is formed by sorting the nodes at each level of the tree so that the key at each node precedes its right neighbor in the frequency ordering (i.e. its index is less, cf. Figure 5). The permissibility of this device is shown by the following.

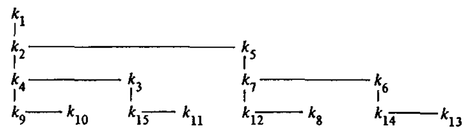
Claim. *The left canonical form of a valid MST is also valid.*

Proof. Consider a process which sorts each level, proceeding down from the root, by exchanging *subtrees* rooted at the same level if the key stored at the root of the subtree on the right precedes that on the left in the frequency ordering. Clearly, this process will terminate with the tree in left canonical form. Consider the tree at some intermediate stage in this process, after l levels have been sorted. Consider the result of exchanging two subtrees rooted at k_x and k_y such that k_x is to the left of k_y in level $l + 1$, and $x > y$. If the father of k_x is also the father of k_y, then there will be no change of parent, and thus no change in validity. Otherwise, as the father of k_x ($k_{f(x)}$) precedes the father of k_y ($k_{f(y)}$) on level l, $f(x) < f(y)$ so $p_{f(x)} \geq p_{f(y)}$. Therefore, as $p_{f(i)} \geq p_i$ for all i in a valid tree, $p_{f(x)} \geq p_y$. Furthermore, as $x > y$, $p_x \leq p_y$ so, as $p_y \leq p_{f(y)}$, $p_{f(y)} \geq p_x$. Consequently, as such an exchange does not invalidate a valid tree, the result of a sequence of such exchanges applied to a valid tree (its left canonical form) is also valid.

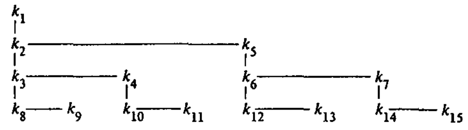
The left canonical form gives us a single representative of all trees which induce the same set of level assignments, each of which corresponds to a large set of different lexical orderings. This allows a much more concise statement of the results which follow.

Given any nonuniform set of frequencies, the cost of the MST will be minimal iff $l_i > l_j \Rightarrow p_i \leq p_j$. This is easily seen to follow from the general result on the minimization of sums of the form $\sum a_i b_i$ [5, p. 403] and the following lemma.

Fig. 5. A median split tree and its left canonical form.



A median split tree on 15 keys.



The median split tree in left canonical form.

Lemma. *Any tree which induces a set of levels such that $l_i > l_j \Rightarrow p_i \leq p_j$ is valid.*

Proof. A tree is valid iff, for all nodes i, $p_i \leq p_{f(i)}$. But, as $l_i = l_{f(i)} + 1$ for all i, $l_i > l_j \Rightarrow p_i \leq p_j$ implies $p_i \leq p_{f(i)}$, which establishes the tree's validity.

If the p_i are all distinct, then $p_i < p_j \rightleftarrows i > j$, so $l_i > l_j \Rightarrow i > j$ which is nothing less than the definition of breadth-first enumeration! Thus we have also shown the following surprising result.

Theorem. *An optimal MST for a set of keys with distinct frequencies assigns keys to nodes in breadth-first order.*

The left canonical form of such a tree is clearly a left to right, breadth-first enumeration of the form shown in Figure 6.

When considering the pessimal MST, a different approach must be used, as the set of level assignments that would maximize the value of the expression $\sum^n p_i l_i$ cannot be induced by a valid MST (e.g. it would put the most frequent node at a leaf of the tree). Instead, we must consider the constraints imposed by the tree's validity. By construction, a maximally frequent key will be at the root of the tree. Similarly, the root of one of the two subtrees rooted at the second level will contain a maximally frequent key amongst those remaining. Consider the other key at the second level; call it k_b.

Claim. $p_b \geq p_{(n+3)/2}$.

Proof. Consider the smallest j such that $p_b \geq p_j$. As k_b is at the root of a subtree of $(n - 1)/2$ nodes, there must be at least $(n - 1)/2$ keys k_i such that $p_b \geq p_i$. Therefore, as there are $((n - 1)/2) - 1$ keys such that $p_b \geq p_i$ and $i > j$, j is no greater than $n - [((n - 1)/2) - 1]$ which is $(n + 3)/2$.

This result does not depend on the tree being pessimal, and is therefore true for any MST. However, consider the following.

Claim. *In a pessimal tree, $p_b \leq p_{(n+3)/2}$.*

Proof. Consider the left canonical form of a pessimal tree. If $p_b > p_{(n+3)/2}$, then there are *fewer* than $(n + 3)/2$ keys k_i such that $p_i \geq p_b$. The root and k_b itself are two of these. Consequently, at least one of the keys in the subtree of $(n - 1)/2$ nodes rooted at k_b's brother must be

Fig. 6. Left canonical form of the optimal median split tree.

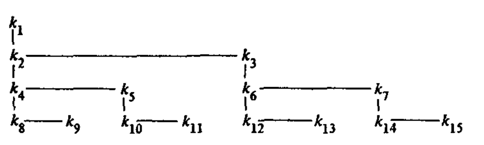
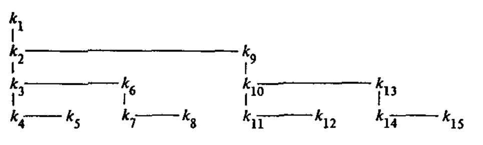


Fig. 7. Left canonical form of the pessimal median split tree.



less frequent than k_b, else there would be $2 + (n - 1)/2 = (n + 3)/2$ keys k_i such that $p_i \geq p_b$, contrary to assumption. Exchange k_b with the shallowest such key, k_x. If k_x is less frequent than the more frequent of the keys at its new sons, exchange k_x and that key and repeat until k_x is at least as frequent than the keys at both its new sons. The new tree will be valid—for k_x by construction; for k_b because its new father is, by selection, more frequent than k_b. The cost of this change to the tree, if k_x was originally at level h and is now at level h', is at most $p_b(h - 2) - [p_x(h - h') + p_b(h' - 2)]$ corresponding to the increase due to k_b having been moved down $h - 2$ levels, and the decrease due to k_x having been moved up $(h - h')$ levels and $h' - 2$ keys with frequency f such that $p_b \geq f > p_x$ being moved up one level each. As $h' < h$ (k_x cannot sink as far down as its old level since, by definition of left canonical form, it is at least as frequent as all keys to the right of its old position, and will therefore find a valid node assignment at or before level $h - 1$) and $p_b > p_x$, this cost is positive, so the original tree was not pessimal.

From the above, the next lemma follows immediately.

Lemma. *In a pessimal tree, $p_b = p_{(n+3)/2}$.*

This may seem to be a very obscure property of the class of pessimal trees. However, in the case where the p_i are distinct it yields a more striking characterization. In that case, it is clear that the keys in the second level of a pessimal tree are $\{k_2, k_{(n+3)/2}\}$, and the difference between their indices is $(n - 1)/2$. However, both of these nodes dominate subtrees containing consecutive keys in the frequency ordering—k_1 the subtree from k_2 to $k_{(n+1)/2}$, and $k_{(n+3)/2}$ the subtree from $k_{(n+3)/2}$ to k_n. Consequently, the above argument may be applied recursively to the set of keys in each subtree to yield a fascinating dual to the optimality result.

Theorem. *A pessimal MST for a set of keys with distinct frequencies assigns keys to nodes in depth-first order.*

Proof. A depth-first enumeration of the nodes of a binary tree is one in which the absolute value of the difference between the indices of brother nodes in the enumeration is equal to the number of nodes in a subtree of which one of them is the root. (Intuitively, this is because any depth-first enumeration must visit all the descendents of one brother before visiting the subtree containing the other brother.) Consider the difference between brothers at level i. As they are the second level of a consecutive subtree rooted at level $i - 1$, the argument above shows that the difference between them is $(S_{i-1} - 1)/2$, where S_l is the number of nodes in a tree rooted at level l. As $S_{i-1} = 2S_i + 1$, the difference is therefore $(2S_i + 1 - 1)/2 = S_i$. As the difference between brothers at a given level equals the number of nodes in (both) trees rooted at that level, we conclude the theorem.

When a pessimal tree is put into left canonical form, we get a prefix walk enumeration like that of Figure 7.

Although the correspondence between optimal and pessimal key orderings and the two common tree walks seems surprising at first, it simply reflects the combined effect of the validity constraint and the minimum $\sum a_i b_i$ theorem. In order to preserve validity, the nodes must be distributed into the tree in a top down fashion. Within that constraint, depth-first enumeration maximizes the confounding of the frequency and lexical orderings, by placing consecutive keys in the frequency ordering on the same side of the tree; breadth-first enumeration minimizes the confounding by distributing consecutive keys across the tree. In the first case, high frequency keys are pushed together, and thus down, in the tree; in the second they are distributed across the top of the tree.

The validity requirement acts to constrain the pessimal case so that it is only slightly worse than the optimal case. Although it is difficult to be more precise without knowledge of the frequency distribution, the similarity between the level assignments of Figures 6 and 7, and the small difference between the search costs for optimal and pessimal trees for the data of Figure 3 are suggestive in this regard. Furthermore, not only is the pessimal result comparable with the optimal, but the result of a random lexical ordering is likely to be much closer to the optimal than to the pessimal value. That is, the behavior of the search performance is stable around optimal values. Once again, our ignorance of the frequency distribution makes it difficult to attack this question directly, as it is not possible to define a metric on differences in search costs. However, some suggestive observations can be made.

We begin by noting the relationships of MST's to the orderings that induce them. Note that the mapping from MST's to lexical orderings is one to many. To see this, note that the split value of each node partitions its descendents. As the aggregate of these partitions constitutes a partial order on the keys, the tree will be compatible with any lexical order which is a topological sort of this partial order. Specifically, as an MST does not determine the position of its root in the lexical ordering, the number of orderings that produce a given MST can be seen to be

$$G(n) = \begin{cases} 1 & \text{if } n = 1 \\ nG((n-1)/2)^2 & \text{otherwise.} \end{cases}$$

A closed form for this recurrence does not concern us as much as the fact that all MST's of a given size correspond to the same number of lexical orderings. Furthermore, as the lexical ordering, viewed as a permutation of the frequency ordering, uniquely determines the MST, the mapping from MST's to lexical orderings partitions the set of orderings. Therefore we can determine the probability of an MST of a given performance given a random lexical ordering by considering the number of MST's of that performance, as each of them corresponds to the same number of lexical orderings. As search performance is determined by the level assignments of the keys, the number of MST's of a given performance is, in turn, given by the number of ways the keys can be permuted within levels without destroying the validity of the tree. The high degree of constraint present in pessimal trees gives us our result.

THEOREM. (a) *The number of MST's of optimal performance exceeds the number of any other given performance.* (b) *The number of MST's of pessimal performance is no greater than the number of any other given performance.*

PROOF. (a) If the tree is optimal, no key at level i is less frequent than any key at level $i + 1$. Therefore, any permutation of the keys at a given level results in a valid tree. However, if the tree is nonoptimal, then there exists at least one key at some level i which is less frequent than some key at level $i + 1$. Thus a permutation which makes the lowest frequency key at level i the father of the highest frequency key at level $i + 1$ is invalid. Consequently, there are more trees with optimal than any other level assignments. (b) If a tree is pessimal, then no keys on the same level can be exchanged between brother subtrees as all sons of one brother are more frequent than all sons of the other. Consequently, the only permutations possible are permutations of entire subtrees (i.e. if you exchange two keys, you must also exchange the subtrees of which they are the roots). But this is possible for any MST.

Simple counting arguments of this form show that there are $\prod^L (2^{i-1}!)$ optimal trees and $2^{(n-1)/2}$ pessimal trees of size n. The ratio of these two numbers gives some idea of how sharply skewed the performance of MST's is towards the optimal. Small wonder then that the empirical results of Figure 3 show that randomly selected lexical orderings produce results that are close to optimal. It is this attribute of MST's in particular that make them such a useful technique. For even in the unlikely event that the salient lexical ordering produces a near pessimal result, almost any adjustment to it can be expected to produce a near optimal one.

Throughout this section we have been working in assumed ignorance of the frequency distribution, except for our one excursion to derive an upper bound on the search time. We will conclude with a small investigation into the question of the technique's probable performance on real data. To do this we need a model of the frequency distributions encountered in typical key sets. We observe that it is frequently the case that such a distribution satisfies

$$\sum^{j} p_i = \log_{n+1}(j + 1)$$

or, equivalently

$$p_j = \log_{n+1}(j + 1)/j.$$

We will refer to this property as the *cumulative log* property. Frequency distributions of this form give a good fit to those encountered in practice (in particular, a better fit to word frequency data than is given by a Zipf distribution, although the two are clearly closely related), and are easy to manipulate. We have:

THEOREM. *For any key set with the cumulative log property, an optimal MST will have ACSS = (L + 1)/2.*

PROOF. ACSS $= \sum^n p_i l_i / \sum^n p_i$. If we consider only those keys in the MST, so that $\sum^n p_i = 1.0$ (the keys not in the tree do not affect the ACSS), and resum by levels rather than by nodes, we have ACSS $= \sum^L q_j l_j = \sum^L q_j j$ where q_j gives the probability of a key being on the jth level of the tree. As the tree is optimal, a key is on the jth level iff (as all p_i are distinct) $2^{j-1} \le i \le 2^j - 1$. Thus

$$q_j = \sum_{2^{j-1}}^{2^j-1} p_i = \sum^{2^j-1} p_i - \sum^{2^{j-1}-1} p_i$$

which, by the cumulative log property, is

$$\log_{n+1}2^j - \log_{n+1}2^{j-1} = \log_{n+1}2.$$

Thus $q_j = \log_{n+1}2$ for all j, and

$$\begin{aligned} \text{ACSS} &= \sum^L q_j j = (\log_{n+1}2)\sum^L j \\ &= (\log_{n+1}2)L(L + 1)/2 \\ &= (L + 1)/2, \text{ as } L = \log_2(n + 1). \end{aligned}$$

This remarkable result, coupled with the fact that a randomly chosen lexical ordering is unlikely to be far from the optimal, gives us a convenient method of estimating the ACSS for the MST for any key set encountered in practice. It turns out to be a remarkably accurate estimator, and indicates that an MST will normally run about twice as fast as a binary search which does not take the frequencies into account.

Comparison with Other Techniques

The performance of any search method based on nonredundant key comparisons can be expressed in terms of the weighted path length of the corresponding tree. As optimal MST's clearly have minimal weighted internal path length and, therefore, optimal successful search performance for any comparison-based method, the stability of MST's around the optimal MST suggests that they will compare well with other comparison-based methods.

The classical development of comparison-based methods, however, is not only in terms of the cost of successful searches, but in terms of the cost averaged across *all* searches. To this end, an additional set of probabilities which give the probability that a key presented in the input will fall between two lexically adjacent keys in the high frequency set is considered. The shape of the tree is then chosen so that the high frequency "gaps" between keys occur as high in the tree as possible, so as also to reduce the average number of nodes visited during an unsuccessful search. However, as MST's place all terminal nodes (and, therefore, all gaps) on the same level (or, at worst, adjacent levels if n is not of the form $2^L - 1$), such adjustment is not possible for MST's and this might be considered a disadvantage.

There are two reasons why this is doubtful. First, in most applications of tree search, failure to find a key is followed by a search of external memory whose cost is vastly greater than that of internal search. Consequently, there seems to be little advantage in degrading the performance of successful searches in order to obtain a proportionately negligible improvement in the unsuccessful ones. Second, even when external search costs are not an issue, the gap probabilities are usually neither known nor is there any reason to believe that they will be especially skewed. This is important because the case for adjusting the tree to take advantage of these gaps depends on their being sufficiently skewed so that this gain will outweigh the loss due to not being able to use a basically more efficient scheme such as MST search. Empirically, it appears that even on key sets selected to illustrate skewed gap probabilities (e.g. KWIC index term lookup, where the presence of words with long common prefixes leads to sharp differences in interword probabilities) the decoupling of functions in the MST allows it to achieve higher search performance than a FOBS adjusted to take advantage of the interkey probabilities.

The noncomparison-based methods fall into two classes—those based on comparisons of the representations of the keys (e.g. trie or digital search) and those based on key transformations. Of the representation based methods, trie search is by far the most competitive. A straightforward implementation of a trie uses much more space than is practical for internal tables, although the technique is often used for external lookup. However, Maly [7] has recently presented a scheme for compressing tries into an amount of space which is comparable with that used by other internal table methods, making trie search an attractive alternative.

The efficiency of a trie comes from its use of m-way, rather than two-way, branching at each node, based on indexing with an m-valued component of the key (e.g. a 26-way branch using alphabetic characters from a character string key). The use of m-way branching results in an expected search time of $\log_m n$, a speedup of $\log_2 m$ over methods based on binary branching. However, as m is bounded both by restrictions needed for the compressed storage mechanism to work efficiently and also by the range of values of the components used to form the trie (e.g. character components from string keys), the $\log_2 m$ speedup is actually a constant (~ 5) in most contexts. As trie search is degraded by irregularities in the lexical distribution of the keys (e.g. long common prefixes) and as the compressed trie comparison operations are considerably more complex than those used by key comparison schemes, a factor of $\log_2 m$ is only a limited advantage.

Key transformation, or hashing, techniques may also be applied to this problem, but care is required as the limited availability of internal storage implies that the hash tables must be very full. A perfect hashing function (one that takes each key to a different hash value) is, of course, the ideal, but such functions may be difficult to find for moderate size sets ($n > 100$) with high table loadings. Sprugnoli [8] discusses some methods for finding such functions. If collisions are permitted, the best method is separate chaining with the overflow lists ordered by frequency. This method can be used with a hash table of size n (or nearest prime greater than n) with the overflow entries stored in the cells to which none of the high frequency entries hash. The storage cost of this scheme is two entries per key, which is the same as that required for an MST, but its expected ACSS is about 1.5 [5, p. 518] which is independent of n, and better than the corresponding figure for MST's when $n > 2^{(1.5 \times 2)-1} = 4$. However, in the high frequency lookup case, where n is bounded, other (constant) costs, such as the cost of the hashing function itself and its uniformity, which may be suspect over the small key sets which constitute a high frequency table, must also be considered. A further problem is that hashing scatters the highest frequency tokens across the full width of the internal table. In some environments, this can result in paging traffic if system wide memory contention forces some part of the high frequency table out of internal memory, whereas an MST groups the highest frequency items together at the left end of the linearized tree, minimizing this problem.

The existence of several techniques which are competitive for this application suggests that the choice among them will probably be determined by how the technique interacts with other aspects of the specific problem environment. To illustrate how these might affect the use of MST's, we will now consider their application to the organization of the internal tables of a dictionary lookup system for English.

3. Application to Dictionary Lookup for English

Dictionary lookup is one of the more common applications of the techniques described here. The task, an essential component of any nontrivial language processing, consists of the retrieval of the dictionary entries associated with the words found in the input stream by the lexical analyzer. These latter are referred to as "tokens" both to indicate their tentative "wordhood" and to distinguish an occurrence of a word in the input from its representation in the dictionary ("types"). We will concentrate on English here, for the sake of specificity, but similar considerations would obtain with any language for which lexical analysis can be done independently of morphology and syntax.

The two primary concerns in this problem are the size of the internal tables, determined by the maximum tolerable fault rate, and the speed of a successful search. As the cost of a fault is very much greater than the cost of internal searching, the mild increase in the cost of the internal search with expanded internal tables has little effect on their size. Instead, it is bounded (usually somewhere between 100–200 types) by the following considerations. First, if the size of the internal tables becomes too large they become too expensive to keep in internal memory. The exact point at which this happens depends on the size of the dictionary entries and the rest of the system in which the tables are being used. Second, even were internal memory availability not a factor, other search methods may be more desirable on the lower frequency keys. For example, although it could be argued that in a pure virtual storage system in which the whole dictionary was stored in an MST, the root pages of the tree would be kept in the working set by the high frequency of accesses to them, thus dynamically adjusting the section of the dictionary to be kept in internal memory, this search method is not particularly appropriate for the low frequency tokens. The reason for this is that the MST technique is designed to take advantage of sharply skewed distributions of frequency of occurrence. However, as the token frequencies drop, this distribution becomes much more nearly uniform. As was shown above, MST's on a uniformly distributed key set reduce to binary search. Therefore, one could obtain the same performance on the low frequency keys by using uniform binary search (ignoring the frequencies) while saving the storage space of the split cells.

Another reason for shifting search techniques after about the first 150 types which is specific to the case of natural language is the existence of a systematic morphology. The number of types for an English dictionary can be decreased by a factor of somewhere between 6 and 10 if inflected forms are analyzed in terms of their components instead of being separately entered into the dictionary. However, in order to capitalize on this source of efficiency, the external tables must be organized so that failure of a lookup for an inflected word is rapid and also facilitates looking for the inflected form. Thus a uniform hashing algorithm that would be suitable for a compiler symbol table is inappropriate here, as the hash value of the root and an inflected form are unlikely to be the same, leading to multiple external accesses for a single search. Similarly, MST's are inappropriate as a failed search is both expensive and likely to reference several pages of the external file that are not referenced by the root form. Instead, some form of trie search seems most suitable for the external form of the dictionary. This type of search has the property that inflected forms often fail very high "up" in the tree (thus at low cost) and the point of failure is normally on the same page as the correct continuation of the root form. Kay and Kaplan [2] give a detailed description of a lookup scheme which combines trie search and systematic morphological analysis.

For the rest of this section, we will assume, for the sake of specificity, that the internal tables must contain the 166 most frequent words. Our choice of this figure is based on the fact that the 166 most frequent words include exactly those words whose frequencies of occurrence exceed 5 in 10^4.[2]

Although the ACSS is the most useful measure for contrasting different MST's, other measures, including the probability of an external search and the cost of those items that are searched for unsuccessfully in the MST are useful. We will use a three-part measure of the performance of the MST sections of the dictionary. The first part is the probability of an external reference (E); the second is the ACSS, familiar from the section on theoretical analyses; and the other is the average cost per *token* (ACT)—which includes the cost of searching the MST unsuccessfully. The formulae for the probability of an external search and the average number of nodes visited per token can easily be seen to be

$$\mathrm{E} = 1 - p_m,$$
$$\mathrm{ACT} = p_m.\mathrm{ACSS} + (1 - p_m)L = L - p_m(L - \mathrm{ACSS})$$

where $p_m = \sum^n p_i$ is the probability of a token being in the MST.

As English token frequencies are well approximated by the cumulative log model of the previous section, we know that the ACSS for an MST with optimal level assignments will be approximately $(L + 1)/2$. As one would expect, given the stability of MST level assignments around the optimal values, the level assignments of alphabetic lexical ordering differ only slightly from those assigned by an optimal ordering, so this approximation is valid for trees based on alphabetic lexical order trees also. The actual values of the ACSS for trees of various sizes are shown in Table I, along with the predicted value, illustrating how remarkably accurate this approximation is. Using this estimate, it can be seen that the ACSS for a tree of the first 166 types will be between 4 and 4.5; the actual value is 4.38.

The other two aspects of performance, ACT and the external reference probability, are shown plotted against each other for $n = 2^L - 1$ for L from 8 to 0 in Figure 8. Because of the cumulative log property, this is almost exactly a lateral reflection of the plot of ACT against log n, and were the graph completed for a typical complete dictionary of English, the cumulative log property implies it would cross the vertical axis at ACT $\simeq$ (log(5 $\times$ 10^4))/2 $\simeq$ 8.

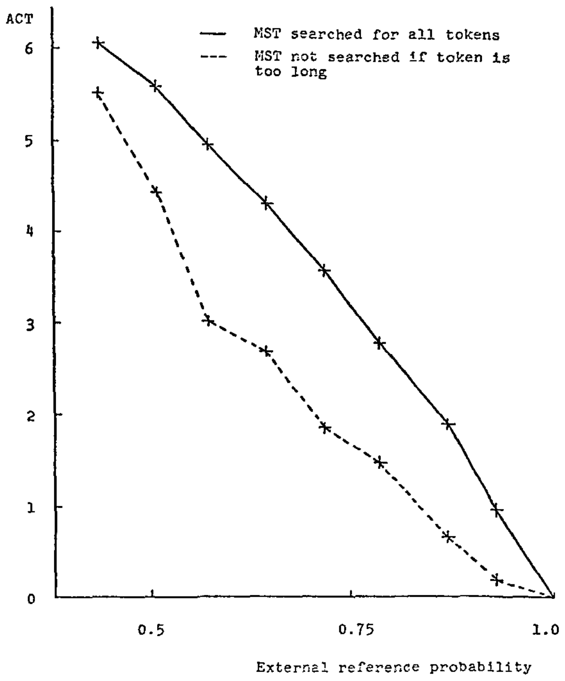
The values shown in this plot are unfortunately high. As the size of the MST is increased to drive down the fault rate, the increasing depth of the tree exacts a penalty

[2] All word frequency data in this section comes from the Brown corpus [6].

Table I. ACSS (alphabetic key ordering) and cumulative log prediction for the N most frequent words of English. (N from 1 to 255.)

N	L	$\frac{1}{2}(L+1)$	ACSS
1	1	1.0	1.0000
3	2	1.5	1.4826
7	3	2.0	2.0524
15	4	2.5	2.5243
31	5	3.0	3.0558
63	6	3.5	3.5891
127	7	4.0	4.0874
255	8	4.5	4.4345

Fig. 8. Average cost per token plotted against the probability of making an external reference. Points plotted (from left to right): N = 255, 127, 63, 31, 15, 7, 3, 1, 0.



in terms of those tokens that are searched for unsuccessfully. In order to avoid this, it would be desirable if there were some aspect of the tokens which would enable us to avoid searching the table if a token will not be found there. If one were able to reduce the fraction of the tokens that were searched for in the tree to p_w then

$$\begin{aligned}\text{ACT} &= p_w(\text{ACSS}\, p_m/p_w + (1 - p_m/p_w)L)\\ &= p_wL - p_m(L - \text{ACSS})\end{aligned}$$

giving, as one might expect, a reduction of $(1 - p_w)L$ corresponding to those tokens not in the tree which are no longer searched for there.

Although a variety of ad hoc heuristics can be used to reduce p_w for any given MST, by far the most powerful in English is the use of word length. Also plotted on Figure 8, is the modified ACT if only words of lengths found in the MST are searched for therein. As can be seen, a dramatic improvement on the order of 50 percent

Table II. Performance of length stratified MST's for English dictionary lookup.

Word length	Types	% Tokens	Depth	ACSS	ACT
1	2	2.81	1.5	1.18	0.037
2	23	16.57	4.5	2.77	0.476
3	40	18.96	5.3	2.67	0.616
4	54	9.13	5.7	3.87	0.732
5	30	3.44	4.9	3.47	0.487
6	7	0.53	3.0	2.30	0.253
7	9	0.61	3.2	2.66	0.248
8	1	0.06	1.0	1.00	0.056
Totals	166	52.10		2.88	2.906

is obtained in the performance of the smaller MST's, but the improvement tails off as the MST's get larger and p_w goes to 1.0. For the 166 word tree, the longest word used is "american" of length 8, so the improvement is

$$\begin{aligned}&(1 - \text{probability word length} < 9)L\\ &= (1 - 0.897)7.305 = 0.752\end{aligned}$$

yielding a performance of [0.479, 4.376, 5.026].

Although this is an improvement, the ACT is still uncomfortably large. A dramatic improvement can be obtained if, rather than using one MST and bypassing the search if the length of the input precludes its being in the high frequency set, one splits the search set into several MST's—each containing only elements of a given length. This strategy pays off for two reasons. First, there is a clear gain in information theoretic terms if one avoids searching through elements of the wrong length. Second, as there are fewer elements in each tree, the trees are shallower and thus an unsuccessful search will cost less. Note that this stratification by length costs almost nothing: Each word length can have an associated MST which is found by indexing a pointer table; and the MST's for different lengths can be laid end to end in exactly the same amount of storage that the complete tree would take. One *advantage* of this approach, however, is that a different amount of storage per node can be used in trees for different word lengths. Thus a word oriented machine which can store five characters per word can use one word per field in trees for words of length five or less, and two words per cell for the others, avoiding either length counts or wasted storage.

When such a set of MST's is constructed for the 166 most frequent words of English, eight trees, whose performance is summarized in Table II, are obtained. Collectively, they result in a performance of [0.479, 2.877, 2.906]. While other heuristics (e.g. maintaining a length by first letter table which locates the first possible match node within an MST) are plausible, it should be pointed out that the simple length stratified MST's are avoiding 52 percent of the external searches at a cost of about 16 machine instructions per token. It seems doubtful that there is enough scope for improvement to justify further embellishment.

4. Conclusions

The results presented here have shown that the storing of two key values in the nodes of a static binary search tree can significantly speed searches of that tree by decoupling the potentially conflicting functions of frequency ordering and subtree structuring. Median split trees have been shown to be good estimates of the corresponding optimal split trees, to have log n bounded execution time, require less storage than FOBS trees, and to be stable around a performance which is optimal for any method based on key comparisons.

It is hoped that they will find application in any searching problem characterized by a static set of keys with a highly skewed distribution of frequencies of occurrence. One such application, dictionary lookup for English, has been discussed in detail and shown to be well suited to this technique. This success provides encouragement for the investigation of other applications of this type.

Received February 1977; revised January 1978

References
1. Blum, M., Floyd, R., Pratt, V., Rivest, R., and Tarjan, R. Time bounds for selection. *JCSS 7*, 4 (Aug. 1973), 448–461.
2. Kay, M., and Kaplan, R. Word recognition. Xerox Palo Alto Res. Ctr., Palo Alto, Calif., 1976.
3. Knuth, D. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
4. Knuth, D. Optimum binary search trees. *Acta Informatica 1*, 1 (1971), 14–25.
5. Knuth, D. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
6. Kucera, H., and Francis, W. *Computational Analysis of Present-day American English.* Brown U. Press, Providence, R.I., 1967.
7. Maly, K. Compressed tries. *Comm. ACM 19*, 7 (July 1976), 409–415.
8. Sprugnoli, R. Perfect hashing functions: A single probe retrieving method for static sets. *Comm. ACM 20*, 11 (Nov. 1977), 841–849.
9. Zipf, G. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, Mass., 1949.

Corrigendum. Programming Languages
C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM 21*, 8 (August 1978), 666-677.

The technical department for this paper was incorrectly listed as Programming Techniques, S.L. Graham and R.L. Rivest, Editors. The correct department is Programming Languages, J.J. Horning, Editor, with Ben Wegbreit having completed the editorial consideration.

Programming Techniques — S.L. Graham, R.L. Rivest Editors

Synthesizing Constraint Expressions

Eugene C. Freuder
University of New Hampshire

A constraint network representation is presented for a combinatorial search problem: finding values for a set of variables subject to a set of constraints. A theory of consistency levels in such networks is formulated, which is related to problems of backtrack tree search efficiency. An algorithm is developed that can achieve any level of consistency desired, in order to preprocess the problem for subsequent backtrack search, or to function as an alternative to backtrack search by explicitly determining all solutions.

Key Words and Phrases: backtrack, combinatorial algorithms, constraint networks, constraint satisfaction, graph coloring, network consistency, relaxation, scene labeling, search

CR Categories: 3.63, 3.64, 5.25, 5.30, 5.32

1. Satisfying Simultaneous Constraints: Problem and Applications

We are given a set of variables $X_1, \ldots, X_n$ and constraints on subsets of these variables limiting the values they can take on. These constraints taken together constitute a global constraint which specifies which sets of values $a_1, \ldots, a_n$ for $X_1, \ldots, X_n$ can simultaneously satisfy all the given constraints. In other words, the constraints define an n-ary relation. Our problem is to synthesize this relation, i.e. to determine those sets of values which simultaneously satisfy the set of constraints.

Author's address: Department of Mathematics and Computer Science, University of New Hampshire, Durham, NH 03824.

Research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.