Two-Way Comparison Search Trees
A Generalization of Binary Search Trees and Split Trees

David Spuler
Dept. of Computer Science
James Cook University of North Queensland

Abstract

A generalization of binary search trees and binary split trees is developed that takes advantage of two-way key comparisons — the two-way comparison tree. The two-way comparison tree is shown to have little use for dynamic situations, but to be an improvement over the optimal binary search tree and the optimal binary split tree for static data sets. An $O(n)$ *time and space algorithm is presented for construction of the optimal two-way comparison tree when access probabilities are equal; the construction of the optimal tree for more general probabilities is shown to be more difficult and unlikely to cost less than* $O(n^4)$.

1. Introduction

Binary search trees are a well-known and effective method of searching for data. However, it has been noted that they can be improved upon when a 3-way key comparison costs twice as much as a 2-way key comparison. In that case, a node visitation in a binary search tree involves two 2-way comparisons — an equality comparison and an order comparison. Sheil [1978] noted that an improvement would be to separate these two tests, applying them to different keys, and introduced a new data structure for this purpose: the binary split tree. A node in the binary split tree contains two keys; one is used for the equality comparison and the other for the order comparison.

Given the separation of two types of tests, a generalization of both these types of trees becomes interesting. As motivation, consider the case when two keys have much higher frequencies than all other keys (including unsuccessful search). In this case, the optimal search order is to test both keys for equality before performing any order tests. Neither binary search trees nor split trees can achieve this search order.

2. Definition of the Two-Way Comparison Tree

The generalization is to have a tree where each node contains a single key and a boolean flag to indicate whether a comparison with the key should be for equality or for order. Nodes are called *equality* nodes or *order* nodes. If a node is an equality node, it has at most one subtree (i.e. it has one sub-tree which can be nil), but an order node can have zero, one or two sub-trees (i.e. two sub-trees, both of which can be nil).

The search algorithm for the two-way comparison tree is shown in Figure 1. It is presented as a simple recursive algorithm, but a more efficient iterative version can be simply developed (although as we shall see, the efficiency of this search algorithm is not important).

```
procedure search(node);
begin
 if node = nil then          {*** unsuccessful search when get to nil leaves ***}
       NOTFOUND(searchkey)
 else
  begin
   if node is an equality node then
     begin
       if searchkey = nodeˆ.key then     {*** equality test ***}
            FOUND(searchkey)
       else
            search(nodeˆ.left)     {*** search single descendant ***}
     end
   else
     begin
       if searchkey <= nodeˆ.key then
            search(nodeˆ.left)     {****** search left sub-tree ***}
       else
            search(nodeˆ.right)    {*** search right sub-tree ***}
     end
  end
end
```

FIGURE 1. Search Algorithm for Two-Way Comparison Tree

The search algorithm as given above for a two-way comparison tree is *less* efficient than searching a binary search tree or a split tree because of the extra boolean test to determine whether the node is an equality or order node. This inefficiency means that the two-way comparison tree is not a good data structure for the dynamic situation where an *explicit* tree must be maintained. However, the node-type test can be removed for trees representing *static data sets* by cleverly coding the tree *implicitly* in program code, rather than explicitly in an actual tree data structure. All finite data structures can be converted to execution code using this method, and the two-way comparison tree will be finite under the assumption of static data. In this way, each node of the tree is converted into an if-then-else programming language statement. The tree structure is implicitly represented by program flow, and in fact, the program flow graph for that part of the program becomes a tree. At each node, there is no longer any need to test the boolean flag, because it is clear which type of node is being used, and the test is not needed at run-time. This is rather like unrolling all the recursive calls of the search routine into inline code — it is possible because the recursive depth is finite, as the tree has finite depth. For example, the algorithm to search the two-way comparison tree shown in Figure 2 can be written as inline code as shown in Figure 3.

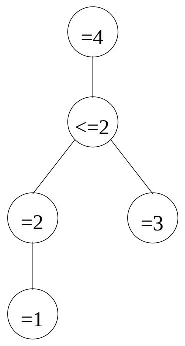

FIGURE 2. Two-way comparison tree for keys 1..4

```
if searchkey = 4 then
    FOUND(4)
else
    begin
        if searchkey <= 2 then
            begin
                if searchkey = 2 then
                    FOUND(2)
                else if searchkey = 1 then
                    FOUND(1)
                else
                    NOTFOUND(searchkey)
            end
        else {*** key > 2 ***}
            begin
                if searchkey = 3 then
                    FOUND(3)
                else
                    NOTFOUND(searchkey)
            end
    end
```

FIGURE 3. if-then-else statements for 2-way comparison tree

3. Properties of the Two-Way Comparison Tree

All keys being searched for must be stored in exactly one equality node — they must appear in a node once so as to be found, and appearing twice would be redundant and inefficient. Hence there are exactly *n* equality nodes.

None of the keys need appear in an order node, although the same keys will often be used in both places. Order nodes need only form an index over the other nodes. One situation where the use of non-key order nodes would be useful is to use smaller keys where the order comparison is cheaper. There can be less than *n* order nodes (e.g. there could be zero order nodes if the first *n* comparisons are all for equality).

At an order node, all the keys in equality nodes in the left subtree will be ≤ the key in the order node, and all keys in equality nodes in the right subtree will be > the key in the order node.

There is no explicit relationship between the key in an equality node and keys in equality nodes of its sub-tree (although this key will usually be the most heavily weighted, so as to make the tree as close as possible to optimal).

4. Binary Search Trees and Split Trees as Special Cases

The binary search tree is a special case of two-way comparison tree where each equality node has as its immediate child an order node containing the same key. This assumes a tree search algorithm that does its equality test first; otherwise the corresponding two-way comparison tree is an order node with the equality node with the same key as the left child. Thus, the tree structure alternates between equality and order nodes.

A split tree is also a special case of the two-way comparison tree which has the same restriction that the tree structure alternates between equality and order nodes as the search moves down the tree. An order node always has an equality node as its parent. However, split trees do not require the keys to be the same in pairs of nodes.

5. Constructing the Optimal Two-Way Comparison Tree For Equal Probabilities

An efficient algorithm can be developed for constructing the optimal two-way comparison tree under the assumption that keys have equal access probabilities for successful search and that unsuccessful search does not occur. These assumptions correspond well to an application in compiler design — code generation for the Pascal case statement — where probabilities of the cases are unknown (and therefore assumed equal) and unsuccessful search is unimportant since it is an exceptional condition that leads to program termination with a diagnostic (refer to [Spuler, 1992]).

For the following discussion we will denote the cost of the optimal tree with n nodes as c_n. Note that this cost is dependent only on n because all access frequencies are equal and, of course it is not dependent on the key values — thus, any optimal tree with n keys has cost c_n. For convenience we will denote the cost of the empty tree with zero keys as $c_0 = 0$.

The crucial point to note in the construction of the optimal two-way comparison tree, which follows from the recursive nature of the tree, is that *subtrees of the optimal tree must also be optimal*.

Hence, when finding the optimal tree for n keys we need only examine all the possible choices of root nodes and determine the costs of the optimal subtrees on either side of the root node. This approach is analogous to that used to build the optimal binary search tree [Knuth, 1973, p436]. The choice of a root node has two main alternatives: an equality node or an order node. Any node can be used as the equality node, as all have equal probabilities. The cost of the resulting tree is 1 comparison for the (equality) root node plus the cost of the subtree containing $n - 1$ keys (an equality node has only one subtree). The cost of the subtree with $n - 1$ keys will be:

$$c_{n-1} + n - 1.$$

because the cost of the subtree with cost c_{n-1} is the sum of the heights of all the *equality* nodes:

$$\sum_{i=1}^{n-1} h_i\, p_i = \sum_{i=1}^{n-1} h_i \text{, because } p_i = 1$$

and hanging this subtree from another node increases the height of each equality node by one, leading to the new cost:

$$\sum_{i=1}^{n-1} h_i + 1 = \sum_{i=1}^{n-1} h_i + \sum_{i=1}^{n-1} 1 = c_{n-1} + n - 1$$

Thus the total cost of the resulting tree by choosing an equality root node is:

$$c_n = 1 + c_{n-1} + n - 1 = c_{n-1} + n$$

The choice of an *order* root node is more complicated, as different choices of root nodes lead to different costs. There are n different choices of keys for the order node (although some of them are trivially irrelevant), and the cost of the resulting tree for each choice will be the sum of the costs of the two subtrees (the root node has no weight because it is an order node and search cannot terminate there). Letting i denote the number of keys in the left subtree (with $n - i$ keys in the right subtree), the cost of the resulting tree will be:

$$c_n = c_i + c_{n-i} + n$$

where the "+n" term arises for similar reasons to the $n - 1$ term in the equality root node choice — all of the n equality keys in the two optimal subtrees have their height increased by 1, thus adding n to the total cost.

This idea leads to a *dynamic programming* solution whereby the optimal trees for small n are calculated first, stored in a table and then used to build optimal trees for larger n. This algorithm follows the approach to constructing the optimal binary search tree (see [Knuth, 1973, p436]). The choice of root node can be made so that its cost is the minimum of the cost of the single tree resulting from an equality root node, and the formulae for the n possible trees from the n different choices of order root nodes. The computation of the optimal tree directly from this recurrence relation yields an $O(n^2)$ algorithm.

However, it turns out that a much simpler approach can be used to construct the optimal tree for equal probabilities. An exact solution of this recurrence relation is possible. For $n \le 3$ the optimal two-way tree is a sequence of equality nodes (i.e. equivalent to linear search). For $n = 4$ there are two possible optimal trees — either four equality nodes in sequence, or an order node as root with two equality nodes in each subtree. The tree with the order node is preferable, because although it has the same average cost as the one with only equality nodes, its total height is one less and therefore has slightly reduced worst case cost. For $n > 4$ the optimal tree always has an order node as root (i.e. never an equality node), and the choice of order node is simply the $\lfloor n/2 \rfloor$th node.

That the solution for $n > 4$ is so simple appears surprising initially, but it can be explained intuitively from the information-theoretic viewpoint in that testing the middle key gets the maximum "information" from a comparison.

The optimal two-way comparison tree has order nodes in the internal nodes forming an index over subtrees containing equality nodes close to the leaves. Once a subtree has as few as 3 keys, it becomes a sequence of equality nodes. Intuitively, this is similar to a binary search over an array that changes to linear search when the size of the interval to be searched is less than or equal to 3.

By using the above solution to the recurrence, a simple Pascal function can be designed to build the optimal tree. This routine is very similar to that used to build a complete binary search tree from the binary search algorithm [Knuth, 1973, p409].

```
function build_two_way2(keys : ArrayOfKeys; left,right:integer): TreePtr;
var
  n : integer;
  mid : integer;
  temp : TreePtr;
begin
   n := right - left + 1;   {*** number of keys ***}
   if n <= 3 then
        case n of      {*** Build sequence of equality nodes ***}
        0: temp := nil;
        1: temp := new_node(keys[left], equality);
        2: begin
                temp := new_node(keys[left], equality);
                temp^.left := new_node(keys[left+1], equality);
           end;
        3: begin
                temp := new_node(keys[left], equality);
                temp^.left := new_node(keys[left+1], equality);
                temp^.left^.left := new_node(keys[left+2], equality);
           end;
        end
  else
        begin
          mid := (left + right) div 2;   {*** the ⌊n/2⌋th key ***}
           temp := new_node(keys[mid], order);   {*** Order node ***}
           temp^.left := build_two_way2(keys, left, mid);     {*** keys <= key[mid] ***}
           temp^.right := build_two_way2(keys, mid+1, right); {*** keys > key[mid] ***}
        end;
  build_two_way2 := temp;
end;
```

6. Analysis and Proof

The solution of the recurrence for c_n has been presented above and intuitive arguments have supported it. However, a formal proof is necessary to show that the algorithm produces the optimal tree. For $n \le 4$ it is adequate to enumerate all possible trees to show that for $n \le 3$ the root node will be an equality node and that for $n = 4$ either an equality node or an order node (with $k = 2$) is adequate. The general proof that an order node is optimal for $n > 4$ and also that $k = \lfloor n/2 \rfloor$ is more difficult.

Lemma 1. The exact solution of c_n has the following form:

$$c_0 = 0,\ c_1 = 1,\ c_2 = 3$$
$$c_n = a(b+4) + (3b+6)2^b, \text{ for } n \ge 3, \text{ where } n = a + 3 * 2^b,\ b \ge 0,\ 0 \le a < 3 * 2^b.$$

or equivalently:

$$c_n = 2a + (b+2)n \qquad , \text{ for } n \ge 3, \text{ where } n = a + 3 * 2^b,\ b \ge 0,\ 0 \le a < 3 * 2^b.$$

Proof: The particular values for $n \le 2$ can be found by simple enumeration of all possible trees. In addition, the values of c_n for $3 \le n \le 5$ can be verified by enumeration of possible trees, direct computation of c_n and calculation of the general formula for c_n for $n \ge 3$. For $n \ge 6$ the proposition can be proved by showing that:

$$c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + n$$

For $n \ge 6$ we can express n as $n = a + 3 * 2^b$ where $b > 0$ and $0 \le a < 3 * 2^b$. Note that $b > 0$ implies that $3 * 2^b$ is even. Therefore, the following identities hold:

$$\lfloor n/2 \rfloor = \left\lfloor a + 3 * 2^b \right\rfloor = \lfloor a/2 \rfloor + 3 * 2^{b-1}$$
$$\lceil n/2 \rceil = \left\lceil a + 3 * 2^b \right\rceil = \lceil a/2 \rceil + 3 * 2^{b-1}$$

Also because of the inequality $a < 3 * 2^b$ the following inequalities apply:

$$\lfloor a/2 \rfloor < 3 * 2^{b-1}$$
$$\lceil a/2 \rceil \le 3 * 2^{b-1}$$

Hence we can express $c_{\lfloor n/2 \rfloor}$ uniquely in terms of $a' = \lfloor a/2 \rfloor$ and $b' = b - 1$ where $a' < 3 * 2^{b'}$. However, there are two cases for $c_{\lceil n/2 \rceil}$: either $\lceil a/2 \rceil < 3 * 2^{b-1}$ or $\lceil a/2 \rceil = 3 * 2^{b-1}$. In the first case we can express $c_{\lceil n/2 \rceil}$ uniquely in terms of $a' = \lceil a/2 \rceil$ and $b' = b - 1$ where $a' < 3 * 2^{b'}$. Therefore:

$$\begin{aligned}
& c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + n \\
&= \lfloor a/2 \rfloor((b-1)+4) + (3(b-1)+6)\, 2^{b-1} + \lceil a/2 \rceil((b-1)+4) + (3(b-1)+6)\, 2^{b-1} + a + 3 * 2^b \\
&= (\lfloor a/2 \rfloor + \lceil a/2 \rceil)(b+3) + (3b+3)\, 2^b + a + 3 * 2^b \\
&= a(b+3) + (3b+6)\, 2^b + a \\
&= a(b+4) + (3b+6)\, 2^b \\
&= c_n
\end{aligned}$$

For the second case when $\lceil a/2 \rceil = 3 * 2^{b-1}$ we can express $c_{\lceil n/2 \rceil}$ uniquely in terms of $a' = 0$ and $b' = b$. Therefore:

$$\begin{aligned}
& c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + n \\
&= \lfloor a/2 \rfloor((b-1)+4) + (3(b-1)+6)\, 2^{b-1} + 0((b-1)+4) + (3b+6)\, 2^b + a + 3 * 2^b \\
&= \lfloor a/2 \rfloor(b+3) + (3b+3)\, 2^{b-1} + (3b+6)\, 2^b + a + 3 * 2^b \\
&= \lfloor a/2 \rfloor(b+3) + ((b+1) + 2(b+2) + 2)3 * 2^{b-1} + a \\
&= \lfloor a/2 \rfloor(b+3) + ((b+1) + 2(b+2) + 2)\lceil a/2 \rceil + a \\
&= (\lfloor a/2 \rfloor + \lceil a/2 \rceil)(b+3) + (2b+4)\lceil a/2 \rceil + a \\
&= a(b+4) + (2b+4)3 * 2^{b-1} \\
&= a(b+4) + (3b+6)2^b \\
&= c_n
\end{aligned}$$

The second form of the exact expression presented in the Lemma follows easily by arithmetic manipulations from the use of the substitution $n = 3 * 2^b$, as follows:

$$\begin{aligned}
2a + (b+2)n &= 2a + (b+2)(a + 3 * 2^b) \\
&= 2a + a(b+2) + (b+2)(3 * 2^b) \\
&= a(b+4) + (3b+6)2^b
\end{aligned}$$

Lemma 2. The sequence of differences $d_n = c_{n+1} - c_n$ has the following exact form for $n \ge 3$:

$$d_{a+3*2^b} = b + 4,$$

where:

$$n = a + 3 * 2^b,\ b \ge 0,\ 0 \le a < 3 * 2^b.$$

Proof: Let n be expressed as $n = a + 3 * 2^b$ where $b \ge 0$ and $0 \le a < 3 * 2^b$ which allows c_n to be expressed in terms of a and b. $n + 1$ can be expressed as $n = a' + 3 * 2^{b'}$, where $a' = a + 1$ and $b' = b$ and therefore c_{n+1} can be expressed in terms of $a + 1$ and b, provided $a + 1 < 3 * 2^b$. Therefore:

$$\begin{aligned}
d_n &= c_{n+1} - c_n \\
&= (b+2)(n+1) + 2(a+1) - (b+2)n - 2a \\
&= bn + 2n + b + 2 + 2a + 2 - bn - 2n - 2a \\
&= b + 4
\end{aligned}$$

At the boundary case where $a + 1 = 3 * 2^b$ we can express c_{n+1} in terms of $a' = 0$ and $b' = b + 1$ giving:

$$\begin{aligned}
d_n &= c_{n+1} - c_n \\
&= (b+1+2)(n+1) + 2(0) - (b+2)n - 2a \\
&= bn + 3n + b + 3 - bn - 2n - 2a \\
&= b + 4 + (n - 1 - 2a) \\
&= b + 4
\end{aligned}$$

The last step follows because $n - 1 - 2a = 0$. At the boundary we have $a + 1 = 3 * 2^b$ and also the general result $n = a + 3 * 2^b$. Together these two equalities imply that $2a = n - 1$.

The form of d_n shows that it contains long sequences of identical numbers which increment by one at the boundaries of the sub-sequences (i.e. when b increases by one). The result shows that d_n is a non-decreasing sequence, and also that c_n is a strictly increasing sequence for $n \geq 3$ since $d_n > 0$.

Lemma 3. The expression $c_k + c_{n-k}$ is non-increasing for $1 \leq k \leq \lfloor (n/2) \rfloor$ when $n \geq 3$.

Proof: To prove the lemma, consider the following condition on k:

$$\begin{aligned}
& k \leq \lfloor n/2 \rfloor \\
\rightarrow\ & k \leq \lceil n/2 \rceil \\
\rightarrow\ & k \leq (n+1)/2 \\
\rightarrow\ & 2k \leq n + 1 \\
\rightarrow\ & k - 1 \leq n - k
\end{aligned}$$

From Lemma 2 we see that d_n is a non-decreasing sequence, and therefore $k - 1 \leq n - k$ implies:

$$\begin{aligned}
& d_{k-1} \leq d_{n-k} \\
\rightarrow\ & c_k - c_{k-1} \leq c_{n-k+1} - c_{n-k} \\
\rightarrow\ & c_k + c_{n-k} \leq c_{n-k+1} + c_{k-1}
\end{aligned}$$

This shows that $c_k + c_{n-k}$ is non-increasing for $1 \leq k \leq \lfloor n/2 \rfloor$ for $n \geq 3$. By symmetry of k and $n - k$ we need not consider values of k in the range $\lfloor n/2 \rfloor + 1 \leq k \leq n$. Therefore $k = \lfloor n/2 \rfloor$ is the optimal choice of k if an order node is chosen.

Lemma 4. The value of c_{n-1} is strictly greater than $c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil}$ for $n > 4$.

Proof: This proposition is proved by showing first that c_{n-1} is strictly greater than $c_2 + c_{n-2}$ for $n \geq 5$ as follows.

$$\begin{aligned}
& c_{n-1} > c_2 + c_{n-2} \\
\text{iff } & c_{n-1} - c_{n-2} - c_2 > 0 \\
\text{iff } & d_{n-2} - 3 > 0 \\
\text{iff } & b + 4 - 3 = b + 1 > 0,
\end{aligned}$$

The last step follows because when $n \geq 5$, we can express $n - 2$ in the form $a + 3 * 2^b$ where $b \geq 0$ and $0 \leq a < 3 * 2^b$. Since $b \geq 0$, the condition $b + 1 > 0$ is always true, and the intermediate result has been proven.

The above has shown that c_{n-1} is larger than the choice of $k = 2$ from the second case of the recurrence, but has not related it to the optimal choice of $k = \lfloor n/2 \rfloor$. However, since $n \geq 5$ we have $\lfloor n/2 \rfloor \geq 2$ and therefore by Lemma 3 we have the relation:

$$c_2 + c_{n-2} \geq c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil}, \text{ for } n \geq 5.$$

Therefore:

$$c_{n-1} > c_2 + c_{n-2} \geq c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil}, \text{ for } n \geq 5.$$

This has shown that the optimal choice of node for $n > 4$ is an order node with $k = \lfloor n/2 \rfloor$.

Theorem 1. The optimal two-way comparison tree for equal probabilities for n nodes has an equality node as the root node for $n \leq 3$, either an equality node or order node with $k = 2$ when $n = 4$, and an order node as the root node for $n > 4$, with the $\lfloor n/2 \rfloor$th key being stored in the order node.

Proof: The proposition is easily proved for $n \leq 4$ by enumeration of all possible trees or by calculation of c_n. Lemma 1 has proved an exact solution to c_n which follows the conditions specified by this theorem. Lemma 4 has proved that for $n > 4$ the value of c_n using an order node is always better than that for an equality node. Lemma 2 has proved that the optimal choice of order node will be $k = \lfloor n/2 \rfloor$.

Theorem 2. The asymptotic behaviour of c_n is $O(n \log n)$.

Proof: The exact form of c_n for $n \geq 3$ can be presented in terms of n by calculating a and b from $n = a + 3 * 2^b$. By the definition of a we have

$$a < 3 * 2^b$$

and also:

$$n = a + 3 * 2^b$$

Therefore:

$$a \leq n/2$$

Also, b is bounded by:

$$b = \lfloor \log_2 (n - a)/3 \rfloor \leq \lfloor \log_2 n/3 \rfloor \leq \log_2 n$$

Together, these bounds on a and b imply:

$$\begin{aligned} c_n &= 2a + (2 + b)n \\ &\leq 2(n/2) + (2 + \log_2 n)n \end{aligned}$$

The asymptotic behaviour of this function is obviously $O(n \log n)$. Therefore the average cost of searching for a key in the optimal two-way comparison tree with equal access probabilities is $O(\log n)$ two-way comparisons.

7. The Optimal Two-Way Comparison Tree For Unequal Probabilities

Let us now consider the problem of computing the optimal two-way comparison tree in the general situation where access frequencies for successful and unsuccessful search are known and are not all identical. This problem has already been studied for the special cases of the binary search tree and the binary split tree, and the research there gives us some insight into the difficulty of the problem.

The best known algorithm for constructing the optimal binary search tree has $O(n^2)$ time complexity [Knuth, 1973, p436]. The best known algorithm for the optimal binary split tree has $O(n^4)$ complexity with the restriction that the access probabilities for each key are all different; without this restriction, the best known algorithm is $O(n^5)$ [Hester, Hirschberg, Huang and Wong, 1986]. For both these situations, the space complexity is $O(n^3)$. The reason for this apparently strange restriction of distinct access frequencies is that if access frequencies are distinct, they can be placed into a strict ordering which simplifies the dynamic programming algorithm. Since the key with maximum weight is always placed as the value key in the root of a split tree, for a given interval of keys with a number of keys "missing" (i.e. as value keys in higher internal nodes), the k missing keys are known to be the k keys with greatest weights and a value can be ascribed to the optimal split tree from that interval with a particular number of keys missing. The details can be found in [Huang and Wong, 1984 (b)], [Perl, 1984], and [Hester, Hirschberg, Huang and Wong, 1986].

Huang and Wong [1984a] have also demonstrated a result about split trees that is not intuitively obvious — that having the key with maximum weight as the value key in the root node does not necessarily give the best tree. They have defined "generalized split trees" which lift the restriction that the maximally weighted key need be in the root node. Huang and Wong [1984a] give an $O(n^5)$ algorithm for constructing optimal generalized split trees.

Since two-way comparison trees are a generalization of split trees (and of generalized split trees) we would assume that the best algorithm for the optimal two-way comparison tree would have at least $O(n^4)$ cost, possibly under the restriction of distinct access probabilities. The discovery by Huang and Wong [1984a] and [1984b] that generalized split trees can be better than split trees has implications for designing an algorithm for the optimal two-way split tree. In particular, when the root node of the optimal two-way tree is an *equality* node it does not necessarily follow that choosing the key with the largest weight will produce the optimal tree; it might be true but there is as yet no proof nor counter-example.

It may be fruitful to define a class of restricted two-way comparison trees where an equality node must contain the maximally weighted key. For this class of two-way trees we could use similar dynamic programming techniques to those for split trees, particularly if we assume distinct access probabilities.

8. Conclusions

The two-way comparison tree is a generalization of binary search trees and binary split trees, which takes full advantage of the breakdown of a 3-way key comparison into two 2-way key comparisons. It is a practical alternative for searching static sets of data, but is not useful for dynamic data sets (i.e. it is not practical if insertion or deletion are required). Explicit representation of a two-way comparison tree is inefficient because of the need to test a boolean flag during search, but an *implicit* two-way comparison tree represented by *if-then-else* program statements can be very efficient. The conversion of an explicit two-way comparison tree to *if-then-else* statements (or to lower-level assembly language) is a simple procedure and can be automated.

A linear time algorithm has been presented for building the optimal two-way comparison tree when access frequencies are equal (or assumed equal when they are unknown). The problem of building the optimal tree under more general weights is examined and it is conjectured that it will cost at least $O(n^4)$. The author is currently examining this more difficult problem.

9. Acknowledgments

Special thanks to colleagues on the `sci.math` international news group, especially David Larue and Laurent Alonso, for their help in solving the recurrence.

10. References

Hester, J.H., Hirschberg, D.S., Huang, S-H.S. and Wong, C.K., "Faster Construction of Optimal Binary Split Trees", *J of Algorithms*, 7(3):412-424, (Sep 1986).

Hester, J.H., Hirschberg, D.S. and Larmore, L.L., "Construction of Optimal Binary Split Trees in the Presence of Bounded Access Probabilities", *J of Algorithms*, 9(22):245-253, (June 1988).

Hester, J.H. and Hirschberg, D.S., "Faster Construction of Optimal Binary Split Trees", *Journal of Algorithms*, Vol 7, 412-424, 1986.

Huang, S-H.S. and Wong, C.K., "Generalized Binary Split Trees", *Acta Informatica*, 21(1):113-123, (1984).

Huang, S-H.S. and Wong, C.K., "Optimal Binary Split Trees", *J of Algorithms*, 5(1):6579, (Mar 1984).

Knuth, D.E., *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, Mass, 1973.

Perl, Y., "Optimum split trees", *J of Algorithms*, 5(3):367-374, (Sep 1984).

Sheil, B.A., "Median Split Trees: A Fast Lookup Technique for Frequently Occurring Keys", *C.ACM*, Vol. 21, No. 11, p947-958, Nov 1978.

Spuler, D.A., "Code Generation for the Pascal Case Statement", *in preparation*, 1992.